

Motor Control Firmware

Generated by Doxygen 1.9.8

1 Hierarchical Index	1
1.1 Class Hierarchy	1
2 Class Index	3
2.1 Class List	3
3 File Index	5
3.1 File List	5
4 Class Documentation	7
4.1 CommandType Class Reference	7
4.1.1 Detailed Description	7
4.2 ControllerState Class Reference	7
4.2.1 Detailed Description	8
4.2.2 Member Function Documentation	8
4.2.2.1 enter()	8
4.2.2.2 leave()	8
4.2.2.3 setController()	8
4.2.2.4 update()	9
4.3 CurrentSense Class Reference	9
4.3.1 Detailed Description	9
4.3.2 Member Function Documentation	10
4.3.2.1 getCurrent()	10
4.3.2.2 initialize()	10
4.4 Direction Class Reference	10
4.4.1 Detailed Description	10
4.5 EMA< K, uint_t > Class Template Reference	11
4.5.1 Detailed Description	11
4.6 Motor Class Reference	11
4.6.1 Detailed Description	13
4.6.2 Constructor & Destructor Documentation	13
4.6.2.1 Motor()	13
4.6.3 Member Function Documentation	14
4.6.3.1 displayInfo()	14
4.6.3.2 drive()	14
4.6.3.3 getCurrent()	14
4.6.3.4 getNormalizedPos()	15
4.6.3.5 hitBottom()	15
4.6.3.6 isStopped()	15
4.6.3.7 setSpeed()	15
4.6.3.8 topReached()	16
4.6.3.9 update()	16
4.6.4 Member Data Documentation	16

4.6.4.1 homing	16
4.6.4.2 lastPos	16
4.6.4.3 pos	16
4.6.4.4 speed	17
4.6.4.5 totalPulseCount	17
4.7 MotorController Class Reference	17
4.7.1 Detailed Description	20
4.7.2 Constructor & Destructor Documentation	20
4.7.2.1 MotorController()	20
4.7.3 Member Function Documentation	21
4.7.3.1 currentAlarmTriggered()	21
4.7.3.2 disableMotors()	21
4.7.3.3 displayCurrents()	21
4.7.3.4 extend()	21
4.7.3.5 home()	22
4.7.3.6 immediateHalt()	22
4.7.3.7 initialize()	22
4.7.3.8 isStopped()	23
4.7.3.9 motorsCloseToEndOfRange()	23
4.7.3.10 motorsDesynced()	23
4.7.3.11 motorsStopped()	23
4.7.3.12 printCurrent()	24
4.7.3.13 report()	24
4.7.3.14 resetCurrentInformation()	24
4.7.3.15 resetMotorCurrentAlarms()	24
4.7.3.16 resetPid()	25
4.7.3.17 resetSoftMovement()	25
4.7.3.18 retract()	26
4.7.3.19 sampleCurrents()	26
4.7.3.20 savePosition()	26
4.7.3.21 setCurrentLimit()	26
4.7.3.22 setPos()	27
4.7.3.23 setSpeed()	27
4.7.3.24 startMotion()	27
4.7.3.25 stop()	28
4.7.3.26 update()	28
4.7.3.27 updateCurrentReadings()	28
4.7.3.28 updateLeadingAndLaggingIndicies()	29
4.7.3.29 updateMotors()	29
4.7.3.30 updateSoftMovement()	29
4.7.3.31 zero()	30
4.8 MotorPins Class Reference	30

4.8.1 Detailed Description	30
4.9 MotorsSoftMovementState Class Reference	31
4.9.1 Detailed Description	31
4.9.2 Member Function Documentation	32
4.9.2.1 enter()	32
4.9.2.2 leave()	32
4.9.2.3 update()	32
4.10 MotorsStartingState Class Reference	33
4.10.1 Detailed Description	33
4.10.2 Member Function Documentation	34
4.10.2.1 enter()	34
4.10.2.2 leave()	34
4.10.2.3 update()	34
4.11 MotorsStoppedState Class Reference	35
4.11.1 Detailed Description	35
4.11.2 Member Function Documentation	36
4.11.2.1 enter()	36
4.11.2.2 leave()	36
4.11.2.3 update()	36
4.12 PIDController Class Reference	36
4.12.1 Detailed Description	37
4.12.2 Constructor & Destructor Documentation	37
4.12.2.1 PIDController()	37
4.12.3 Member Function Documentation	38
4.12.3.1 adjustSpeed()	38
4.12.3.2 report()	38
4.12.3.3 reset()	38
4.12.3.4 setParams()	39
4.13 StateController Class Reference	39
4.13.1 Member Function Documentation	39
4.13.1.1 setController()	39
4.13.1.2 setState()	40
4.13.1.3 update()	40
5 File Documentation	41
5.1 Commands.hpp File Reference	41
5.1.1 Enumeration Type Documentation	41
5.1.1.1 Command	41
5.2 Commands.hpp	42
5.3 ControllerState.hpp File Reference	43
5.4 ControllerState.hpp	43
5.5 ControlPins.hpp File Reference	43

5.6 ControlPins.hpp	44
5.7 CurrentSense.hpp File Reference	44
5.8 CurrentSense.hpp	44
5.9 defs.hpp File Reference	45
5.9.1 Variable Documentation	48
5.9.1.1 ADC_LOGIC_VOLTAGE	48
5.9.1.2 CURRENT_INCREASE_MULTIPLIER	48
5.9.1.3 MOTOR_END_OF_RANGE_SPEED_DELTA	48
5.9.1.4 save_position_slot_names	48
5.9.1.5 SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS	48
5.9.1.6 SOFT_MOVEMENT_UPDATE_STEPS	49
5.10 defs.hpp	49
5.11 Direction.hpp File Reference	50
5.11.1 Enumeration Type Documentation	50
5.11.1.1 Direction	50
5.12 Direction.hpp	51
5.13 EMA.hpp	51
5.14 Motor.hpp File Reference	51
5.14.1 Macro Definition Documentation	52
5.14.1.1 MOVE_TO_POS	52
5.15 Motor.hpp	52
5.16 MotorController.hpp File Reference	56
5.16.1 Macro Definition Documentation	56
5.16.1.1 ALL_MOTORS	56
5.16.1.2 SERIAL_SAVE_POSITION	56
5.17 MotorController.hpp	57
5.18 MotorPins.hpp File Reference	67
5.18.1 Enumeration Type Documentation	67
5.18.1.1 MotorPin	67
5.19 MotorPins.hpp	68
5.20 MotorsSoftMovementState.hpp File Reference	68
5.21 MotorsSoftMovementState.hpp	69
5.22 MotorsStartingState.hpp File Reference	69
5.23 MotorsStartingState.hpp	70
5.24 MotorsStoppedState.hpp File Reference	70
5.25 MotorsStoppedState.hpp	71
5.26 PIDController.hpp	71
5.27 PinMacros.hpp File Reference	73
5.27.1 Function Documentation	74
5.27.1.1 fmap()	74
5.28 PinMacros.hpp	74
5.29 PotentiometerPins.hpp File Reference	75

5.29.1 Enumeration Type Documentation	75
5.29.1.1 PotentiometerPins	75
5.30 PotentiometerPins.hpp	75
5.31 RouteMacros.hpp File Reference	76
5.31.1 Macro Definition Documentation	76
5.31.1.1 LOAD_SAVED_POSITION	76
5.31.1.2 MOTOR_COMMAND	76
5.31.1.3 SET_POS_HANDLER	76
5.31.1.4 SET_TILT	77
5.32 RouteMacros.hpp	77
5.33 StateController.hpp	77
Index	79

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

CommandType	7
ControllerState	7
MotorsSoftMovementState	31
MotorsStartingState	33
MotorsStoppedState	35
CurrentSense	9
Direction	10
EMA< K, uint_t >	11
EMA< 1 >	11
Motor	11
MotorController	17
MotorPins	30
PIDController	36
StateController	39

Chapter 2

Class Index

2.1 Class List

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

CommandType	Represents the types of commands recognized by the firmware	7
ControllerState	Base class for controller state	7
CurrentSense	This implements the current sense functionality for the motors	9
Direction	Direction values for the direction indicator for the motor controller and the motors themselves	10
EMA< K, uint_t >	Calculates a fast exponential moving average	11
Motor	This class represents the motor controlled by the microcontroller	11
MotorController	This is the controller of the motors	17
MotorPins	Pin number definitions for the motor	30
MotorsSoftMovementState	Represents the motors making a soft movement	31
MotorsStartingState	Handles the starting state for motor control system	33
MotorsStoppedState	Handles the stopped state for motor control system	35
PIDController	This is the PID controller for motor synchronization	36
StateController	39

Chapter 3

File Index

3.1 File List

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

Commands.hpp	41
ControllerState.hpp	43
ControlPins.hpp	43
CurrentSense.hpp	44
defs.hpp	45
Direction.hpp	50
EMA.hpp	51
Motor.hpp	51
MotorController.hpp	56
MotorPins.hpp	67
MotorsSoftMovementState.hpp	68
MotorsStartingState.hpp	69
MotorsStoppedState.hpp	70
PIDController.hpp	71
PinMacros.hpp	73
PotentiometerPins.hpp	75
RouteMacros.hpp	76
StateController.hpp	77

Chapter 4

Class Documentation

4.1 CommandType Class Reference

Represents the types of commands recognized by the firmware.

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

4.1.1 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:

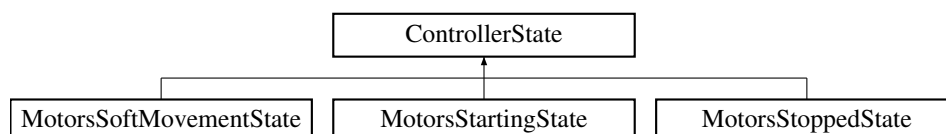
- [Commands.hpp](#)

4.2 ControllerState Class Reference

Base class for controller state.

```
#include <ControllerState.hpp>
```

Inheritance diagram for ControllerState:



Public Member Functions

- **ControllerState** ([MotorController](#) *pMotorController=nullptr)
- void [setController](#) ([MotorController](#) *pMotorController)
- virtual void [enter](#) ()=0
- virtual void [update](#) ()=0
- virtual void [leave](#) ()=0

Protected Attributes

- [MotorController](#) * **controller**

4.2.1 Detailed Description

Base class for controller state.

Author

Terry Paul Ferguson
terry@terryferguson.us

Version

0.1

4.2.2 Member Function Documentation

4.2.2.1 enter()

```
virtual void ControllerState::enter ( ) [pure virtual]
```

Implemented in [MotorsSoftMovementState](#), [MotorsStartingState](#), and [MotorsStoppedState](#).

4.2.2.2 leave()

```
virtual void ControllerState::leave ( ) [pure virtual]
```

Implemented in [MotorsSoftMovementState](#), [MotorsStartingState](#), and [MotorsStoppedState](#).

4.2.2.3 setController()

```
void ControllerState::setController (
    MotorController * pMotorController ) [inline]
```

Sets the [MotorController](#) for the object.

Parameters

<i>pMotorController</i>	The MotorController to set.
-------------------------	---

Exceptions

<i>None</i>	
-------------	--

4.2.2.4 update()

```
virtual void ControllerState::update ( ) [pure virtual]
```

Implemented in [MotorsSoftMovementState](#), [MotorsStartingState](#), and [MotorsStoppedState](#).

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

- [ControllerState.hpp](#)

4.3 CurrentSense Class Reference

This implements the current sense functionality for the motors.

```
#include <CurrentSense.hpp>
```

Public Member Functions

- **CurrentSense** (const adc1_channel_t pCurrentSensePin=ADC1_CHANNEL_0, const double pLogicVoltage=[ADC_LOGIC_VOLTAGE](#), const int32_t pMaxAdcValue=[MAX_ADC_VALUE](#))
- void [initialize](#) (const adc1_channel_t pCurrentSensePin=ADC1_CHANNEL_0)
Initialize the current sensing pin and calibrate the ACS offset.
- int [getCurrent](#) () const
Calculates the average current.

4.3.1 Detailed Description

This implements the current sense functionality for the motors.

Author

Terry Paul Ferguson
terry@terryferguson.us

Version

0.1

4.3.2 Member Function Documentation

4.3.2.1 `getCurrent()`

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

Calculates the average current.

Returns

the average current of the sampling

Exceptions

<i>None</i>	
-------------	--

4.3.2.2 `initialize()`

```
void CurrentSense::initialize (
    const adc1_channel_t pCurrentSensePin = ADC1_CHANNEL_0 ) [inline]
```

Initialize the current sensing pin and calibrate the ACS offset.

Parameters

<i>pCurrentSensePin</i>	The pin used for current sensing. Defaults to ADC1_CHANNEL_0.
-------------------------	---

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

- [CurrentSense.hpp](#)

4.4 Direction Class Reference

[Direction](#) values for the direction indicator for the motor controller and the motors themselves.

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

4.4.1 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](#)

4.5 EMA< K, uint_t > Class Template Reference

Calculates a fast exponential moving average.

```
#include <EMA.hpp>
```

Public Member Functions

- `uint_t operator() (uint_t input)`
Update the filter with the given input and return the filtered output.
- `void reset ()`

Static Public Attributes

- `static constexpr uint_t half = 1 << (K - 1)`
Fixed point representation of one half, used for rounding.

4.5.1 Detailed Description

```
template<uint8_t K, class uint_t = uint_fast32_t>
class EMA< K, uint_t >
```

Calculates a fast exponential moving average.

Template Parameters

<i>K</i>	The amount of bits to shift by. This determines the location of the pole in the EMA transfer function, and therefore the cut-off frequency. The higher this number, the more filtering takes place. The pole location is $1 - 2^{-K}$.
----------	---

Author

Terry Paul Ferguson
terry@terryferguson.us

Version

0.1

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

- `EMA.hpp`

4.6 Motor Class Reference

This class represents the motor controlled by the microcontroller.

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

Public Member Functions

- **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 adc1_channel_t currentSensePin, const int totalPulses, const int freq=[PWM_FREQUENCY](#), const int defSpeed=[MIN_MOTOR_TRAVEL_SPEED](#), const int pwmRes=[PWM_RESOLUTION_BITS](#), const int bottomLimitPin=-1, const int currentLimit=-1, const int alarmCurrentLimit=-1, const int stopBuffer=0)
The constructor for the motor controlled by the microcontroller.
- void **initialize** ()
Initialize motor.
- void **drive** (const [Direction](#) motorDirection, const int specifiedSpeed=0)
Drives the motor in the specified direction at the specified speed.
- 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 **disable** ()
Disables the motor. If the motor is already stopped, this function does nothing. Otherwise, it stops the motor, sets the speed to 0, and logs a message.
- void **zero** ()
Reset the distance sensor count and position variables for the motor to zero.
- bool **hitBottom** () const
Checks if the current position has reached or exceeded the current limit and if the direction is set to retract.
- bool **topReached** () const
Check if the top has been reached.
- void **update** (const int newSpeed=([MAX_SPEED](#)+1))
Update the state of the motor.
- void **readPos** ()
Read the position of the motor.
- float **getNormalizedPos** ()
Get a normalized indicaton of the position of this motor based on its total range.
- void **displayInfo** ()
Displays information about the motor.
- int **getCurrent** () const
Retrieve the current value as milliamps.
- bool **isStopped** () const
Checks if the current motor is in a stopped state.
- void **setSpeed** (int newSpeed)
Set the speed to the specified value.

Public Attributes

- int **pos** = 0
- int **lastPos** = 0
- int **speed** = [DEFAULT_MOTOR_SPEED](#)
- int **totalPulseCount** = 0
- int **stopBuffer** = 0
- int **bottomLimitPin** = -1
- int **bottomCurrentLimit** = -1
- int **currentAlarmLimit** = -1
- bool **outOfRange** = false
- [Direction](#) **dir** = [Direction::STOP](#)
- bool **homing** = false

4.6.1 Detailed Description

This class represents the motor controlled by the microcontroller.

Author

Terry Paul Ferguson

Version

0.1

4.6.2 Constructor & Destructor Documentation

4.6.2.1 Motor()

```
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 adcl_channel_t currentSensePin,
    const int totalPulses,
    const int freq = PWM_FREQUENCY,
    const int defSpeed = MIN_MOTOR_TRAVEL_SPEED,
    const int pwmRes = PWM_RESOLUTION_BITS,
    const int bottomLimitPin = -1,
    const int currentLimit = -1,
    const int alarmCurrentLimit = -1,
    const int stopBuffer = 0 ) [inline]
```

The constructor for the motor controlled by the microcontroller.

Parameters

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

Copy name of linear actuator into ID field

4.6.3 Member Function Documentation

4.6.3.1 displayInfo()

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

Displays information about the motor.

Parameters

None	
------	--

Returns

None

Exceptions

None	
------	--

4.6.3.2 drive()

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

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

Parameters

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

4.6.3.3 getCurrent()

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

Retrieve the current value as milliamps.

Returns

The current value as milliamps

4.6.3.4 getNormalizedPos()

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

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

Returns

The fraction that represents how much of total extension we are currently at as a float value. If the total pulse count is 0, it returns 0.0f.

Exceptions

None	
------	--

4.6.3.5 hitBottom()

```
bool Motor::hitBottom ( ) const [inline]
```

Checks if the current position has reached or exceeded the current limit and if the direction is set to retract.

Returns

true if the current position has reached or exceeded the current limit and the direction is set to retract, false otherwise.

4.6.3.6 isStopped()

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

Checks if the current motor is in a stopped state.

Returns

true if the motor is in a stopped state, false otherwise.

4.6.3.7 setSpeed()

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

Set the speed to the specified value.

Parameters

<i>newSpeed</i>	The new speed value.
-----------------	----------------------

4.6.3.8 topReached()

```
bool Motor::topReached ( ) const [inline]
```

Check if the top has been reached.

Returns

true if the top has been reached, false otherwise

4.6.3.9 update()

```
void Motor::update (
    const int newSpeed = (MAX_SPEED + 1) ) [inline]
```

Update the state of the motor.

Parameters

<i>newSpeed</i>	the new speed value to set (default: MAX_SPEED + 1)
-----------------	---

Exceptions

<i>None</i>	
-------------	--

4.6.4 Member Data Documentation

4.6.4.1 homing

```
bool Motor::homing = false
```

The direction of the motor rotation

4.6.4.2 lastPos

```
int Motor::lastPos = 0
```

The last position of the motor based on hall sensor pulses

4.6.4.3 pos

```
int Motor::pos = 0
```

The current sensor The current position of the motor based on hall sensor pulses

4.6.4.4 speed

```
int Motor::speed = DEFAULT_MOTOR_SPEED
```

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

4.6.4.5 totalPulseCount

```
int Motor::totalPulseCount = 0
```

The total number of pulses from full retraction to full extension

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

- [Motor.hpp](#)

4.7 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](#))
This is the class that controls the motors.
- void [initialize](#) ()
Initialize the motors, position storage, current sensors, and PID controllers.
- void [startMotion](#) (const [Direction](#) dir)
- void [extend](#) ()
Extends the motorized system.
- void [retract](#) ()
Retracts the motorized system.
- void [stop](#) ()
Stops the motorized system.
- void [immediateHalt](#) ()
- void [home](#) ()
Function to control the homing.
- void [setSpeed](#) (const int newSpeed, const int softMovementTime=[SOFT_MOVEMENT_TIME_MS](#))
Set the speed of the motor.
- void [zero](#) ()
Zeroes out the position count of all motors.
- void [report](#) ()
Reports the current state of the motor controller to the serial console.
- void [printCurrent](#) ()
Prints the current values of the leader and follower motors.
- void [savePosition](#) (const int slot, const int position_value=-1)

- Saves the position value for a given slot.*

 - void **setPos** (const int newPos)
 - void **updateCurrentReadings** (const int elapsedTime)

Updates the current readings of the leader and follower motors based on the elapsed time.

 - bool **isStopped** () const

Check if the system is stopped.

 - bool **motorsStopped** () const
 - bool **currentAlarmTriggered** ()

Checks if the current alarm is triggered.

 - bool **motorsDesynced** (void) const
 - void **handlePid** ()
 - bool **motorsCloseToEndOfRange** ()

Check if the motors are close to the end of their range.

 - void **handleCurrentAlarm** ()

Disable all motors, reset speed and direction variables, and turn off PID control. Print debug message if debugEnabled is true.

 - void **updateLeadingAndLaggingIndices** ()

Update the leading and lagging indices based on the system direction.

 - void **displayCurrents** ()

Displays the current values of the leader and follower motor currents and velocities.

 - void **sampleCurrents** ()
 - void **setCurrentLimit** ()
 - void **updateSoftMovement** ()
 - void **update** (const float deltaT=0.0f)

Updates the state of the motor system.

 - void **resetSoftMovement** ()
 - void **resetCurrentInformation** ()

Reset the current information for the system.

 - void **disableMotors** ()

Disable the motors.

 - void **updateMotors** ()

Update the motors.

 - void **resetPid** ()

Reset the PID controller.

 - void **resetMotorCurrentAlarms** ()

Reset the current alarms for the motors.

Public Attributes

- int **moveStart** = -1
- The time in microseconds since a motor movement started.*
- int **targetSpeed** = -1
- The target speed of soft movement.*
- int **K_p** = **DEFAULT_KP**
- The proportional gain for the PID controller*
- **PIDController** **pidController**
- The PID controller for the motor synchronization.*
- int **defaultSpeed** = **DEFAULT_MOTOR_SPEED**
- The default speed to operate the motors at on startup.*
- int **speed** = 0

- Current target speed.*
 - int **leaderCurrent** = 0
 - Leader motor current.*
 - int **followerCurrent** = 0
 - Follower motor current.*
 - int **lastLeaderCurrent** = 0
 - Last leader current reading.*
 - int **lastFollowerCurrent** = 0
 - Last follower current reading.*
 - int **leaderCurrentVelocity** = 0
 - Leader current velocity.*
 - int **followerCurrentVelocity** = 0
 - Follower current velocity.*
 - int **minCurrent** = 900
 - Minimum current to enable alarm system for motors*
- int **currentAlarmDelay** = 250000
 - Current delay for overcurrent alarm system.*
- int **alarmCurrentVelocity** = 10000
 - Current velocity limit for alarm system for motors to enable*
- [Motor](#) **motors** [[NUMBER_OF_MOTORS](#)]
 - The motors controlled by this motor controller instance.*
- [Direction](#) **requestedDirection** = [Direction::STOP](#)
 - The requested system level direction.*
- [Direction](#) **systemDirection** = [Direction::STOP](#)
 - The current system level direction indicator.*
- int **laggingIndex** = 0
 - The index in the motors array of the motor that is behind as indicated by the hall sensor.*
- int **leadingIndex** = 0
 - The index in the motors array of the motor that is farther along as indicated by the hall sensor.*
- int **softStart** = -1
 - The timestamp since soft start of movement.*
- int **lastPWMUpdate** = -1
 - The last PWM update interval in microseconds.*
- float **pwmUpdateAmount** = -1.0f
 - The amount to change the PWM duty cycle on soft start.*
- int **lastPrintTime** = -1
 - The last time a debug serial print was sent.*
- int **currentUpdateInterval** = [CURRENT_UPDATE_INTERVAL](#)
 - Interval of time to pass between current updates microseconds*
- int **lastCurrentUpdate** = -1
 - Time in microseconds since the last current update.*
- int **softMovingTime** = -1
- [EMA](#)< 1 > **leaderCurrentFilter**
 - Input filter for leader current readings.*
- [EMA](#)< 1 > **followerCurrentFilter**
 - Input filter for follower current readings.*
- int_fast32_t **samples** = 0
 - Number of current samples.*

- `int currentOffset = 0`
Current offset.
- `int_fast32_t currentDifferenceSum = 0`
Current difference sum.
- `int_fast32_t currentSum = 0`
Current Sum.

4.7.1 Detailed Description

This is the controller of the motors.

Author

Terry Paul Ferguson
terry@terryferguson.us

Version

0.1

4.7.2 Constructor & Destructor Documentation

4.7.2.1 MotorController()

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

This is the class that controls the motors.

Parameters

<code>PWM_FREQUENCY</code>	the frequency of the PWM signal
<code>PWM_RESOLUTION_BITS</code>	the resolution of the PWM signal in bits
<code>DEFAULT_MOTOR_SPEED</code>	the default speed of the motor

Returns

none

Exceptions

<code>none</code>	
-------------------	--

4.7.3 Member Function Documentation

4.7.3.1 currentAlarmTriggered()

```
bool MotorController::currentAlarmTriggered ( ) [inline]
```

Checks if the current alarm is triggered.

Returns

true if the current alarm is triggered, false otherwise

4.7.3.2 disableMotors()

```
void MotorController::disableMotors ( ) [inline]
```

Disable the motors.

Parameters

None	
------	--

Returns

None

Exceptions

None	
------	--

4.7.3.3 displayCurrents()

```
void MotorController::displayCurrents ( ) [inline]
```

Displays the current values of the leader and follower motor currents and velocities.

Returns

void

4.7.3.4 extend()

```
void MotorController::extend ( ) [inline]
```

Extends the motorized system.

This function enables PID control, resets soft movement, sets the speed to the default speed, resets the out of range flag for all motors, sends the extend command to all motors, and sets the system direction and requested direction to extend.

4.7.3.5 home()

```
void MotorController::home ( ) [inline]
```

Function to control the homing.

This function retracts, checks leader and follower out of range status, and updates the motor status until both motors are out of range. It then zeros out the positions of the motors and resets the soft movement status < Set the PID flag to true

< Reset the soft movement

< Set the outOfRange flag to false for all motors

< Set the command to retract for all motors

< Set the system direction to retract

< Set the requested direction to retract

4.7.3.6 immediateHalt()

```
void MotorController::immediateHalt ( ) [inline]
```

Halts the system immediately.

Parameters

None	
------	--

Returns

None

Exceptions

None	
------	--

4.7.3.7 initialize()

```
void MotorController::initialize ( ) [inline]
```

Initialize the motors, position storage, current sensors, and PID controllers.

Parameters

None	
------	--

Returns

None

Exceptions

None	
------	--

4.7.3.8 isStopped()

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

Check if the system is stopped.

Returns

true if the system is stopped, false otherwise

4.7.3.9 motorsCloseToEndOfRange()

```
bool MotorController::motorsCloseToEndOfRange ( ) [inline]
```

Check if the motors are close to the end of their range.

Returns

True if either motor is close to the end of its range, false otherwise.

4.7.3.10 motorsDesynced()

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

Check if the motors are desynchronized.

Returns

true if the motors are desynchronized, false otherwise

4.7.3.11 motorsStopped()

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

Check if both motors are stopped.

Returns

True if both motors are stopped, false otherwise.

4.7.3.12 printCurrent()

```
void MotorController::printCurrent ( ) [inline]
```

Prints the current values of the leader and follower motors.

Returns

void

4.7.3.13 report()

```
void MotorController::report ( ) [inline]
```

Reports the current state of the motor controller to the serial console.

Returns

void

Exceptions

None	
------	--

4.7.3.14 resetCurrentInformation()

```
void MotorController::resetCurrentInformation ( ) [inline]
```

Reset the current information for the system.

Parameters

None	
------	--

Returns

None

Exceptions

None	
------	--

4.7.3.15 resetMotorCurrentAlarms()

```
void MotorController::resetMotorCurrentAlarms ( ) [inline]
```

Reset the current alarms for the motors.

Parameters

<i>None</i>	
-------------	--

Returns

None

Exceptions

<i>None</i>	
-------------	--

4.7.3.16 resetPid()

```
void MotorController::resetPid ( ) [inline]
```

Reset the PID controller.

Parameters

<i>None</i>	
-------------	--

Returns

None

Exceptions

<i>None</i>	
-------------	--

4.7.3.17 resetSoftMovement()

```
void MotorController::resetSoftMovement ( ) [inline]
```

Reset the soft movement.

Parameters

<i>None</i>	
-------------	--

Returns

None

Exceptions

<i>None</i>	
-------------	--

4.7.3.18 retract()

```
void MotorController::retract ( ) [inline]
```

Retracts the motorized system.

This function is used to tell the motorized system to retract. It sets the PID flag to true, resets the soft movement, sets the speed to the default speed, sets the outOfRange flag to false for all motors, sets the command to retract for all motors, sets the system direction and requested direction to retract.

4.7.3.19 sampleCurrents()

```
void MotorController::sampleCurrents ( ) [inline]
```

Calculate the larger current between leader and follower, update the maxCurrent if necessary, and update the currentDifferenceSum, currentSum, and samples.

Parameters

None	
------	--

Returns

None

Exceptions

None	
------	--

4.7.3.20 savePosition()

```
void MotorController::savePosition (
    const int slot,
    const int position_value = -1 ) [inline]
```

Saves the position value for a given slot.

Parameters

<i>slot</i>	The slot index.
<i>position_value</i>	The position value to save.

4.7.3.21 setCurrentLimit()

```
void MotorController::setCurrentLimit ( ) [inline]
```

Sets the current limit for the leader and follower motors based on the average current and current offset. Prints the current alarms set to the serial monitor.

Parameters

<i>None</i>	
-------------	--

Returns

None

Exceptions

<i>None</i>	
-------------	--

4.7.3.22 setPos()

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

Set the desired position for the motors and move them accordingly.

Parameters

<i>newPos</i>	The new desired position for the motors.
---------------	--

4.7.3.23 setSpeed()

```
void MotorController::setSpeed (
    const int newSpeed,
    const int softMovementTime = SOFT\_MOVEMENT\_TIME\_MS ) [inline]
```

Set the speed of the motor.

Parameters

<i>newSpeed</i>	The new speed value.
-----------------	----------------------

4.7.3.24 startMotion()

```
void MotorController::startMotion (
    const Direction dir ) [inline]
```

Starts the motion in the specified direction.

This function is used to tell the motorized system to move in specified direction. It sets the PID flag to true, resets the soft movement, sets the speed to the default speed, sets the outOfRange flag to false for all motors, sets the system direction and requested direction to specified direction.

Parameters

<i>dir</i>	The direction in which to start the motion
------------	--

Exceptions

<i>None</i>	
-------------	--

4.7.3.25 stop()

```
void MotorController::stop ( ) [inline]
```

Stops the motorized system.

This function stops the motorized system by resetting the soft movement and setting the speed to 0.

4.7.3.26 update()

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

Updates the state of the motor system.

Parameters

<i>deltaT</i>	the time interval since the last update (default: 0.0f)
---------------	---

Exceptions

<i>None</i>	
-------------	--

4.7.3.27 updateCurrentReadings()

```
void MotorController::updateCurrentReadings (
    const int elapsedTime ) [inline]
```

Updates the current readings of the leader and follower motors based on the elapsed time.

Parameters

<i>elapsedTime</i>	the elapsed time in microseconds
--------------------	----------------------------------

Returns

void

Exceptions

None	
------	--

4.7.3.28 updateLeadingAndLaggingIndices()

```
void MotorController::updateLeadingAndLaggingIndices ( ) [inline]
```

Update the leading and lagging indices based on the system direction.

Returns

void

4.7.3.29 updateMotors()

```
void MotorController::updateMotors ( ) [inline]
```

Update the motors.

Parameters

None	
------	--

Returns

None

Exceptions

None	
------	--

4.7.3.30 updateSoftMovement()

```
void MotorController::updateSoftMovement ( ) [inline]
```

Updates the soft movement of the system.

Parameters

None	
------	--

Returns

None

Exceptions

None	
------	--

4.7.3.31 zero()

```
void MotorController::zero ( ) [inline]
```

Zeroes out the position count of all motors.

Returns

void

Exceptions

None	
------	--

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

- [MotorController.hpp](#)

4.8 MotorPins Class Reference

Pin number definitions for the motor.

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

4.8.1 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:

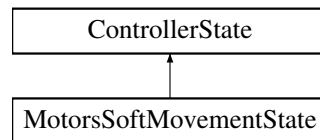
- [MotorPins.hpp](#)

4.9 MotorsSoftMovementState Class Reference

Represents the motors making a soft movement.

```
#include <MotorsSoftMovementState.hpp>
```

Inheritance diagram for MotorsSoftMovementState:



Public Member Functions

- **MotorsSoftMovementState** ([MotorController](#) *pMotorController)
- void [enter](#) ()
Handle the system entering the moving state.
- void [update](#) ()
Update the motor control system in starting state.
- void [leave](#) ()
Handle leaving the starting state.

Public Member Functions inherited from [ControllerState](#)

- **ControllerState** ([MotorController](#) *pMotorController=nullptr)
- void [setController](#) ([MotorController](#) *pMotorController)

Additional Inherited Members

Protected Attributes inherited from [ControllerState](#)

- [MotorController](#) * **controller**

4.9.1 Detailed Description

Represents the motors making a soft movement.

Author

Terry Paul Ferguson
terry@terryferguson.us

Version

0.1

4.9.2 Member Function Documentation

4.9.2.1 enter()

```
void MotorsSoftMovementState::enter ( ) [inline], [virtual]
```

Handle the system entering the moving state.

Returns

void

Implements [ControllerState](#).

4.9.2.2 leave()

```
void MotorsSoftMovementState::leave ( ) [inline], [virtual]
```

Handle leaving the starting state.

Returns

void

Implements [ControllerState](#).

4.9.2.3 update()

```
void MotorsSoftMovementState::update ( ) [inline], [virtual]
```

Update the motor control system in starting state.

Returns

void

Implements [ControllerState](#).

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

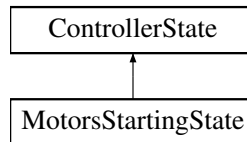
- [MotorsSoftMovementState.hpp](#)

4.10 MotorsStartingState Class Reference

Handles the starting state for motor control system.

```
#include <MotorsStartingState.hpp>
```

Inheritance diagram for MotorsStartingState:



Public Member Functions

- **MotorsStartingState** ([MotorController](#) *pMotorController)
- void [enter](#) ()
Handle the system entering the moving state.
- void [update](#) ()
Update the motor control system in starting state.
- void [leave](#) ()
Handle leaving the starting state.

Public Member Functions inherited from [ControllerState](#)

- **ControllerState** ([MotorController](#) *pMotorController=nullptr)
- void [setController](#) ([MotorController](#) *pMotorController)

Additional Inherited Members

Protected Attributes inherited from [ControllerState](#)

- [MotorController](#) * **controller**

4.10.1 Detailed Description

Handles the starting state for motor control system.

Author

Terry Paul Ferguson
terry@terryferguson.us

Version

0.1

4.10.2 Member Function Documentation

4.10.2.1 enter()

```
void MotorsStartingState::enter ( ) [inline], [virtual]
```

Handle the system entering the moving state.

Returns

void

Implements [ControllerState](#).

4.10.2.2 leave()

```
void MotorsStartingState::leave ( ) [inline], [virtual]
```

Handle leaving the starting state.

Returns

void

Implements [ControllerState](#).

4.10.2.3 update()

```
void MotorsStartingState::update ( ) [inline], [virtual]
```

Update the motor control system in starting state.

Returns

void

Implements [ControllerState](#).

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

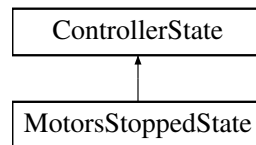
- [MotorsStartingState.hpp](#)

4.11 MotorsStoppedState Class Reference

Handles the stopped state for motor control system.

```
#include <MotorsStoppedState.hpp>
```

Inheritance diagram for MotorsStoppedState:



Public Member Functions

- **MotorsStoppedState** ([MotorController](#) *pMotorController)
- void [enter](#) ()
Handle the system entering the soft motor state.
- void [update](#) ()
Update the motor control system in stopped state.
- void [leave](#) ()
Handle leaving the stopped state.

Public Member Functions inherited from [ControllerState](#)

- **ControllerState** ([MotorController](#) *pMotorController=nullptr)
- void [setController](#) ([MotorController](#) *pMotorController)

Additional Inherited Members

Protected Attributes inherited from [ControllerState](#)

- [MotorController](#) * **controller**

4.11.1 Detailed Description

Handles the stopped state for motor control system.

Author

Terry Paul Ferguson
terry@terryferguson.us

Version

0.1

4.11.2 Member Function Documentation

4.11.2.1 enter()

```
void MotorsStoppedState::enter ( ) [inline], [virtual]
```

Handle the system entering the soft motor state.

Returns

void

Implements [ControllerState](#).

4.11.2.2 leave()

```
void MotorsStoppedState::leave ( ) [inline], [virtual]
```

Handle leaving the stopped state.

Returns

void

Implements [ControllerState](#).

4.11.2.3 update()

```
void MotorsStoppedState::update ( ) [inline], [virtual]
```

Update the motor control system in stopped state.

Returns

void

Implements [ControllerState](#).

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

- [MotorsStoppedState.hpp](#)

4.12 PIDController Class Reference

This is the PID controller for motor synchronization.

```
#include <PIDController.hpp>
```

Public Member Functions

- [PIDController](#) (const int kp=[DEFAULT_KP](#), const float ki=[DEFAULT_KI](#), const float kd=[DEFAULT_KD](#), const float tau=1.0f, const int uMax=[MAX_SPEED](#))

This is the PID controller for motor synchronization.

- void [setParams](#) (const int kpIn, const float kiIn=[DEFAULT_KI](#), const float kdIn=[DEFAULT_KD](#), const int uMaxIn=[MAX_SPEED](#))
- int [adjustSpeed](#) ([Motor](#) &leader, [Motor](#) &follower, const int speed, const float deltaT=0.0f)

Compute the adjusted speed based on the current value, target value, and speed.

- void [setFollowerMaxAccel](#) (float newMaxAccel)
- void [setFollowerMaxSpeed](#) (int newMaxSpeed)
- void [report](#) () const

Reports the PID parameters.

- void [setKd](#) (float newKd)
- void [setKi](#) (float newKi)
- void [reset](#) ()

Reset the PID Controller.

4.12.1 Detailed Description

This is the PID controller for motor synchronization.

Author

Terry Paul Ferguson

terry@terryferguson.us

Version

0.1

4.12.2 Constructor & Destructor Documentation

4.12.2.1 PIDController()

```
PIDController::PIDController (
    const int kp = DEFAULT\_KP,
    const float ki = DEFAULT\_KI,
    const float kd = DEFAULT\_KD,
    const float tau = 1.0f,
    const int uMax = MAX\_SPEED ) [inline]
```

This is the PID controller for motor synchronization.

Parameters

<i>kp</i>	The proportional gain
<i>ki</i>	The integral gain
<i>kd</i>	The derivative gain
<i>tau</i>	The controller's low-pass filter time constant
<i>uMax</i>	Max control signal (speed) for the motors

4.12.3 Member Function Documentation

4.12.3.1 adjustSpeed()

```
int PIDController::adjustSpeed (
    Motor & leader,
    Motor & follower,
    const int speed,
    const float deltaT = 0.0f ) [inline]
```

Compute the adjusted speed based on the current value, target value, and speed.

Parameters

<i>leader</i>	The leader motor
<i>follower</i>	The follower motor
<i>speed</i>	The reference to the speed variable
<i>deltaT</i>	The time difference between the current and previous iteration

Returns

The adjusted speed

4.12.3.2 report()

```
void PIDController::report ( ) const [inline]
```

Reports the PID parameters.

Returns

void

4.12.3.3 reset()

```
void PIDController::reset ( ) [inline]
```

Reset the PID Controller.

Parameters

<i>None</i>	
-------------	--

Returns

None

Exceptions

None	
------	--

4.12.3.4 setParams()

```
void PIDController::setParams (
    const int kpIn,
    const float kiIn = DEFAULT\_KI,
    const float kdIn = DEFAULT\_KD,
    const int uMaxIn = MAX\_SPEED ) [inline]
```

Sets the parameters for the controller.

Parameters

<i>kpIn</i>	the proportional gain parameter
<i>kiIn</i>	the integral gain parameter
<i>kdIn</i>	the derivative gain parameter
$u \leftrightarrow$ <i>MaxIn</i>	the maximum control signal value (default: MAX_SPEED)

Exceptions

None	
------	--

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

- PIDController.hpp

4.13 StateController Class Reference

Public Member Functions

- **StateController** ([MotorController](#) *pMotorController)
- void [setController](#) ([MotorController](#) *pMotorController)
- void [setState](#) (MotorControllerState newState)
- void [update](#) ()

4.13.1 Member Function Documentation

4.13.1.1 setController()

```
void StateController::setController (
    MotorController * pMotorController ) [inline]
```

Sets the [MotorController](#) for the state objects

Parameters

<i>pMotorController</i>	the MotorController object to set
-------------------------	---

Exceptions

<i>None</i>	
-------------	--

4.13.1.2 setState()

```
void StateController::setState (
    MotorControllerState newState ) [inline]
```

Sets the state of the controller to a new state.

Parameters

<i>newState</i>	the new state to set the controller to
-----------------	--

Exceptions

<i>None</i>	
-------------	--

4.13.1.3 update()

```
void StateController::update ( ) [inline]
```

Updates the current state of the motor controller system.

Parameters

<i>None</i>	
-------------	--

Returns

None

Exceptions

<i>None</i>	
-------------	--

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

- StateController.hpp

Chapter 5

File Documentation

5.1 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](#), [HOME](#), [TOGGLE_LIMIT_RANGE](#), [READ_LIMIT](#),
 [SET_POSITION](#), [SET_CURRENT_ALARM](#)}

5.1.1 Enumeration Type Documentation

5.1.1.1 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

Enumerator

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
TOGGLE_PID	Command to tell tell the microcontroller to turn off PID control - 33
HOME	Home the linear actuators - 34
TOGGLE_LIMIT_RANGE	Command to toggle the limit switch - 35
READ_LIMIT	Command to read the limit switches - 36
SET_POSITION	Set a position in hall pulses - 37
SET_CURRENT_ALARM	Set current alarm value - 38

5.2 Commands.hpp

[Go to the documentation of this file.](#)

```

00001
00003 #ifndef _COMMANDS_HPP_
00004 #define _COMMANDS_HPP_
00005
00006 #include <stdint>
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
00065     ZERO,
00066
00068     SYSTEM_RESET,
00069
00071     TOGGLE_PID,
00072
00074     HOME,
00075
00077     TOGGLE_LIMIT_RANGE,
00078
00080     READ_LIMIT,
00081
00083     SET_POSITION,
00084
00086     SET_CURRENT_ALARM,
00087 };
00088
00089 #endif // _COMMANDS_HPP_

```

5.3 ControllerState.hpp File Reference

```
#include "MotorController.hpp"
```

Classes

- class [ControllerState](#)
Base class for controller state.

5.4 ControllerState.hpp

[Go to the documentation of this file.](#)

```
00001
00004 #ifndef _CONTROLLER_STATE_HPP_
00005 #define _CONTROLLER_STATE_HPP_
00006 #include "MotorController.hpp"
00007
00017 class ControllerState {
00018 protected:
00019     MotorController *controller;
00020
00021 public:
00022     ControllerState(MotorController *pMotorController = nullptr)
00023         : controller(pMotorController) {}
00024
00032     void setController(MotorController *pMotorController) {
00033         controller = pMotorController;
00034     }
00035
00036     virtual void enter() = 0;
00037
00038     virtual void update() = 0;
00039
00040     virtual void leave() = 0;
00041
00042     virtual ~ControllerState() = default;
00043 }; // end class State
00044
00045 #endif // _STATE_HPP_
```

5.5 ControlPins.hpp File Reference

```
#include <cstdint>
#include <driver/adc.h>
```

Macros

- #define **LEADER_CURRENT_SENSE_PIN** ADC1_CHANNEL_0
Leader current sense ADC channel pin.
- #define **FOLLOWER_CURRENT_SENSE_PIN** ADC1_CHANNEL_3
Follower current sense ADC channel pin.

5.6 ControlPins.hpp

[Go to the documentation of this file.](#)

```
00001
00003 #ifndef _CONTROL_PINS_HPP_
00004 #define _CONTROL_PINS_HPP_
00005
00006 #include <stdint>
00007 #include <driver/adc.h>
00008
00010 #define LEADER_CURRENT_SENSE_PIN ADC1_CHANNEL_0
00011
00013 #define FOLLOWER_CURRENT_SENSE_PIN ADC1_CHANNEL_3
00014
00015 #endif // _CONTROL_PINS_HPP_
```

5.7 CurrentSense.hpp File Reference

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

Classes

- class [CurrentSense](#)

This implements the current sense functionality for the motors.

5.8 CurrentSense.hpp

[Go to the documentation of this file.](#)

```
00001
00003 #ifndef _CURRENT_SENSE_HPP_
00004 #define _CURRENT_SENSE_HPP_
00005
00006 #include <cmath>
00007 #include <driver/adc.h>
00008 #include <stdint.h>
00009
00010 #include "ControlPins.hpp"
00011 #include "defs.hpp"
00012
00023 class CurrentSense {
00024 private:
00025     int_fast32_t CALIBRATE_ITERATIONS_SHIFT = 14;
00026     int_fast32_t SAMPLE_CURRENT_ITERATIONS_SHIFT = 6;
00027
00029     int32_t MV_PER_AMP = static_cast<int32_t>(185 * 1.132);
00030
00033     // negative current flow.
00034     int32_t ACS_OFFSET = 1885;
00035
00036     adc1_channel_t currentSensePin;
00037     double logicVoltage = ADC_LOGIC_VOLTAGE;
00038     int32_t maxAdcValue = MAX_ADC_VALUE;
00039
00040 public:
00041     CurrentSense(const adc1_channel_t pCurrentSensePin = ADC1_CHANNEL_0,
00042                 const double pLogicVoltage = ADC_LOGIC_VOLTAGE,
00043                 const int32_t pMaxAdcValue = MAX_ADC_VALUE)
00044         : currentSensePin(pCurrentSensePin), logicVoltage(pLogicVoltage),
00045           maxAdcValue(pMaxAdcValue) {}
00046
00053     void initialize(const adc1_channel_t pCurrentSensePin = ADC1_CHANNEL_0) {
```

```

00054 // Set the current sensing pin
00055 currentSensePin = pCurrentSensePin;
00056
00057 // Configure ADC settings
00058 adc1_config_width(ADC_WIDTH_12Bit);
00059 adc1_config_channel_atten(currentSensePin, ADC_ATTEN_DB_11);
00060
00061 Serial.print("Pin: ");
00062 Serial.println(static_cast<uint8_t>(currentSensePin));
00063 Serial.print("Logic Voltage: ");
00064 Serial.println(logicVoltage);
00065 Serial.print("Max ADC Value: ");
00066 Serial.println(maxAdcValue);
00067 Serial.print("mV per A: ");
00068 Serial.println(MV_PER_AMP);
00069
00070 // Calibrate ACS offset
00071 const int iterations = 1 « CALIBRATE_ITERATIONS_SHIFT;
00072 int32_t adcSum = 0;
00073
00074 for (int32_t i = 0; i < iterations; i++) {
00075     adcSum += adc1_get_raw(currentSensePin);
00076 }
00077
00078 ACS_OFFSET = adcSum » CALIBRATE_ITERATIONS_SHIFT;
00079
00080 Serial.printf("ACS Offset: %d\n", ACS_OFFSET);
00081 }
00082
00090 int getCurrent() const {
00091     // Number of iterations for current sampling
00092     const int32_t iterations = 1 « SAMPLE_CURRENT_ITERATIONS_SHIFT;
00093
00094     int32_t currentSum = 0;
00095
00096     // Perform current sampling iterations
00097     for (int i = 0; i < iterations; i++) {
00098         // Calculate ADC offset
00099         const int adcOffset = adc1_get_raw(currentSensePin) - ACS_OFFSET;
00100         // Calculate voltage delta
00101         const double voltageDelta = (adcOffset * (logicVoltage / maxAdcValue));
00102         // Accumulate current sum
00103         currentSum += static_cast<int>(voltageDelta * 1000000.0 / MV_PER_AMP);
00104     }
00105
00106     // Calculate average current
00107     const double averageCurrent = std::abs(
00108         static_cast<double>(currentSum » SAMPLE_CURRENT_ITERATIONS_SHIFT));
00109
00110     // Return average current as an integer
00111     return static_cast<int>(averageCurrent);
00112 } // end method getCurrent
00113 }; // end class CurrentSense
00114
00115 #endif // _CURRENT_SENSE_HPP_

```

5.9 defs.hpp File Reference

```

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

```

Macros

- #define **countof**(a) (sizeof(a) / sizeof(*(a)))

Variables

- const char * **motor_roles** [2] = {"LEADER", "FOLLOWER"}

- String representations of the motor roles at instantiation.*

 - constexpr int **NUM_POSITION_SLOTS** = 5

Number of position slots supported by this firmware.
- const char * [save_position_slot_names](#) [NUM_POSITION_SLOTS]

String representations of the names of position slots.
- int **savedPositions** [NUM_POSITION_SLOTS] = {0, 0, 0, 0, 0}

Storage for position in hall sensor pulses relative to initial position when powered on.
- bool **pid_on** = true

Whether PID is on or off.
- bool **limit_range** = true

Whether limit range is on or off.
- constexpr int **FORMAT_SPIFFS_IF_FAILED** = true
- constexpr int **NUMBER_OF_MOTORS** = 2

The number of motors controlled by this system.
- constexpr int **LEADER_MAX_PULSES** = 2845

Number of pulses for leader motor's maximum extension.
- constexpr int **FOLLOWER_MAX_PULSES** = 2845

Number of pulses for follower motor's maximum extension.
- bool **debugEnabled** = false

Indicates whether debug messages should be sent to serial.
- constexpr int **PWM_FREQUENCY** = 20000

The frequency of the PWM signal sent to the motor controllers.
- constexpr int **PWM_RESOLUTION_BITS** = 8

The resolution of the PWM signal in bits (provides $2^{\text{PWM_RESOLUTION_BITS}}$ of possible levels)
- constexpr int **ADC_RESOLUTION_BITS** = 12

The resolution of the ADC used in bits.
- constexpr int **DEFAULT_MOTOR_SPEED** = (1 << [PWM_RESOLUTION_BITS](#)) - 1

The default speed of the motors given in PWM value.
- constexpr int **MOTOR_END_OF_RANGE_SPEED** = 155

The motor speed at the extremes of the range. Should be < DEFAULT_MOTOR_SPEED
- constexpr int **MIN_MOTOR_TRAVEL_SPEED** = 105

Minimum travel speed.
- constexpr int [MOTOR_END_OF_RANGE_SPEED_DELTA](#)

The difference in speed values between DEFAULT_MOTOR_SPEED and MOTOR_END_OF_RANGE_SPEED
- constexpr int **SET_POSITION_BUFFER** = 3

Set position buffer in hall pulses.
- constexpr int **MILLIS_IN_SEC** = 1000

The number of milliseconds in a second.
- constexpr int **MICROS_IN_MS** = 1000

The number of microseconds in a millisecond.
- constexpr int **MICROS_IN_SEC** = (MILLIS_IN_SEC * [MICROS_IN_MS](#))

The number of microseconds in a second.
- constexpr int **SOFT_MOVEMENT_TIME_MS** = 18000

The number of milliseconds over which the soft movement occurs.
- constexpr int **SOFT_MOVEMENT_MICROS** = (SOFT_MOVEMENT_TIME_MS * [MICROS_IN_MS](#))

The number of microseconds over which the soft movement occurs.
- constexpr int [SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS](#)

The minimum interval between PWM updates in microseconds.
- constexpr int [SOFT_MOVEMENT_UPDATE_STEPS](#)

The maximum number of PWM updates over which the soft movement occurs.
- constexpr int **MAX_SPEED** = ((1 << ([PWM_RESOLUTION_BITS](#))) - 1)

- The maximum speed value that can be represented by PWM.*

 - constexpr int **MIN_SPEED** = 0
- The minimum speed value that can be represented by PWM.*

 - constexpr float **ADC_LOGIC_VOLTAGE** = 3.3f
 - constexpr int **MAX_ADC_VALUE** = (1 << (**ADC_RESOLUTION_BITS**)) - 1
- The maximum ADC value that can be represented by the ADC resolution.*

 - constexpr int **CURRENT_ALARM_AMOUNT** = 1400
- The default minimum current in milliamps needed to trip alarm.*

 - constexpr int **CURRENT_LIMIT** = 2000
- Maximum current in milliamps allowed before the system halts.*

 - constexpr int **CURRENT_OFFSET** = 17
- The current offset between the two motors.*

 - constexpr int **CURRENT_UPDATE_INTERVAL** = 10000
- The minimum interval in microseconds between current reading updates.*

 - constexpr int **CURRENT_ALARM_DELAY** = 2500000
- The minimum time the motors must be moving before enabling the current alarm.*

 - constexpr int **ALARM_REVERSE_AMOUNT** = 15
- The number of hall pulses to back up leader motor after hitting bottom on homing routine.*

 - constexpr int **FOLLOWER_ALARM_REVERSE_AMOUNT** = (**ALARM_REVERSE_AMOUNT** + 30)
- The number of hall pulses to back up follower motor after hitting bottom on homing routine.*

 - constexpr int **LEADER_BUFFER** = 3
- A buffer accounting for the lag in hall pulses between a stop request and the leader motor physically stopping.*

 - constexpr int **FOLLOWER_BUFFER** = 3
- A buffer accounting for the lag in hall pulses between a stop request and the follower motor physically stopping.*

 - constexpr int **DESYNC_TOLERANCE** = 20
- The number of hall pulse difference between the leader and follower motors before triggering an immediate halt.*

 - constexpr float **PID_ALPHA** = 33.333333f
- The alpha ratio value used to PID calculation parameters.*

 - constexpr int **DEFAULT_KP** = 79999
- The default propotional gain used in PID calculation.*

 - constexpr int **RETRACT_KP** = 79999
- The propotional gain used in PID calculation for extension.*

 - constexpr int **STOP_KP** = 79999
- The propotional gain used in PID calculation for stopping.*

 - constexpr int **EXTEND_RAMP_KP** = 79999
- The propotional gain used in PID calculation for extension ramping.*

 - constexpr int **RETRACT_RAMP_KP** = 79999
- The propotional gain used in PID calculation for retraction ramping.*

 - constexpr float **DEFAULT_KI** = ((**DEFAULT_KP** / **PID_ALPHA**) * 6)
- The integral gain used in PID calculation.*

 - constexpr float **DEFAULT_KD** = (**DEFAULT_KP** / (**PID_ALPHA** * 1.5))
- The derivative gain used in PID calculation.*

 - constexpr int **CURRENT_INCREASE_TOLERANCE_PERCENTAGE** = 30
- The tolerance percentage for the current increase before alarm.*

 - constexpr float **CURRENT_INCREASE_MULTIPLIER**
- The current increase multiplier based on the current increase tolerance percentage.*

 - constexpr int **SOFT_STOP_TIME_MS** = 80
- Soft stop time in milliseconds.*

5.9.1 Variable Documentation

5.9.1.1 ADC_LOGIC_VOLTAGE

```
constexpr float ADC_LOGIC_VOLTAGE = 3.3f [constexpr]
```

The logic level voltage of the ADC

5.9.1.2 CURRENT_INCREASE_MULTIPLIER

```
constexpr float CURRENT_INCREASE_MULTIPLIER [constexpr]
```

Initial value:

```
=  
1 + (CURRENT_INCREASE_TOLERANCE_PERCENTAGE / 100.0f)
```

The current increase multiplier based on the current increase tolerance percentage.

5.9.1.3 MOTOR_END_OF_RANGE_SPEED_DELTA

```
constexpr int MOTOR_END_OF_RANGE_SPEED_DELTA [constexpr]
```

Initial value:

```
=  
(DEFAULT_MOTOR_SPEED - MOTOR_END_OF_RANGE_SPEED)
```

The difference in speed values between DEFAULT_MOTOR_SPEED and MOTOR_END_OF_RANGE_SPEED

5.9.1.4 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",  
}
```

String representations of the names of position slots.

5.9.1.5 SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS

```
constexpr int SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS [constexpr]
```

Initial value:

```
=  
SOFT_MOVEMENT_TIME_MS / 20
```

The minimum interval between PWM updates in microseconds.

5.9.1.6 SOFT_MOVEMENT_UPDATE_STEPS

```
constexpr int SOFT_MOVEMENT_UPDATE_STEPS [constexpr]
```

Initial value:

```
=  
(SOFT_MOVEMENT_MICROS / SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS)
```

The maximum number of PWM updates over which the soft movement occurs.

5.10 defs.hpp

[Go to the documentation of this file.](#)

```
00001  
00003 #ifndef _DEFS_HPP_  
00004 #define _DEFS_HPP_  
00005  
00006 #define countof(a) (sizeof(a) / sizeof(*(a)))  
00007  
00008 #include "Commands.hpp"  
00009 #include "ControlPins.hpp"  
00010 #include "Direction.hpp"  
00011 #include "MotorPins.hpp"  
00012  
00014 const char *motor_roles[2] = {"LEADER", "FOLLOWER"};  
00015  
00017 constexpr int NUM_POSITION_SLOTS = 5;  
00018  
00020 const char *save_position_slot_names[NUM_POSITION_SLOTS] = {  
00021     "tilt-1", "tilt-2", "tilt-3", "tilt-4", "tilt-5",  
00022 };  
00023  
00028 int savedPositions[NUM_POSITION_SLOTS] = {0, 0, 0, 0, 0};  
00029  
00031 bool pid_on = true;  
00032  
00034 bool limit_range = true;  
00035  
00036 constexpr int FORMAT_SPIFFS_IF_FAILED = true;  
00037  
00039 constexpr int NUMBER_OF_MOTORS = 2;  
00040  
00042 constexpr int LEADER_MAX_PULSES = 2845;  
00043  
00045 constexpr int FOLLOWER_MAX_PULSES = 2845;  
00046  
00048 bool debugEnabled = false;  
00049  
00051 constexpr int PWM_FREQUENCY = 20000;  
00052  
00055 constexpr int PWM_RESOLUTION_BITS = 8;  
00056  
00058 constexpr int ADC_RESOLUTION_BITS = 12;  
00059  
00061 constexpr int DEFAULT_MOTOR_SPEED = (1 « PWM_RESOLUTION_BITS) - 1;  
00062  
00065 constexpr int MOTOR_END_OF_RANGE_SPEED = 155;  
00066  
00068 constexpr int MIN_MOTOR_TRAVEL_SPEED = 105;  
00069  
00072 constexpr int MOTOR_END_OF_RANGE_SPEED_DELTA =  
00073     (DEFAULT_MOTOR_SPEED - MOTOR_END_OF_RANGE_SPEED);  
00074  
00076 constexpr int SET_POSITION_BUFFER = 3;  
00077  
00079 constexpr int MILLIS_IN_SEC = 1000;  
00080  
00082 constexpr int MICROS_IN_MS = 1000;  
00083  
00085 constexpr int MICROS_IN_SEC = (MILLIS_IN_SEC * MICROS_IN_MS);  
00086  
00088 constexpr int SOFT_MOVEMENT_TIME_MS = 18000;  
00089  
00091 constexpr int SOFT_MOVEMENT_MICROS = (SOFT_MOVEMENT_TIME_MS * MICROS_IN_MS);  
00092  
00094 constexpr int SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS =  
00095     SOFT_MOVEMENT_TIME_MS / 20;
```

```

00096
00099 constexpr int SOFT_MOVEMENT_UPDATE_STEPS =
00100     (SOFT_MOVEMENT_MICROS / SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS);
00101
00103 constexpr int MAX_SPEED = ((1 << (PWM_RESOLUTION_BITS)) - 1);
00104
00106 constexpr int MIN_SPEED = 0;
00107
00109 constexpr float ADC_LOGIC_VOLTAGE = 3.3f;
00110
00113 constexpr int MAX_ADC_VALUE = (1 << (ADC_RESOLUTION_BITS)) - 1;
00114
00116 constexpr int CURRENT_ALARM_AMOUNT = 1400;
00117
00119 constexpr int CURRENT_LIMIT = 2000;
00120
00122 constexpr int CURRENT_OFFSET = 17;
00123
00126 constexpr int CURRENT_UPDATE_INTERVAL = 10000;
00127
00130 constexpr int CURRENT_ALARM_DELAY = 2500000;
00131
00134 constexpr int ALARM_REVERSE_AMOUNT = 15;
00135
00138 constexpr int FOLLOWER_ALARM_REVERSE_AMOUNT = (ALARM_REVERSE_AMOUNT + 30);
00139
00142 constexpr int LEADER_BUFFER = 3;
00143
00146 constexpr int FOLLOWER_BUFFER = 3;
00147
00150 constexpr int DESYNC_TOLERANCE = 20;
00151
00153 constexpr float PID_ALPHA = 33.333333f;
00154
00156 constexpr int DEFAULT_KP = 79999;
00157
00159 constexpr int RETRACT_KP = 79999;
00160
00162 constexpr int STOP_KP = 79999;
00163
00165 constexpr int EXTEND_RAMP_KP = 79999;
00166
00169 constexpr int RETRACT_RAMP_KP = 79999;
00170
00172 constexpr float DEFAULT_KI = ((DEFAULT_KP / PID_ALPHA) * 6);
00173
00175 constexpr float DEFAULT_KD = (DEFAULT_KP / (PID_ALPHA * 1.5));
00176
00178 constexpr int CURRENT_INCREASE_TOLERANCE_PERCENTAGE = 30;
00179
00182 constexpr float CURRENT_INCREASE_MULTIPLIER =
00183     1 + (CURRENT_INCREASE_TOLERANCE_PERCENTAGE / 100.0f);
00184
00186 constexpr int SOFT_STOP_TIME_MS = 80;
00187
00188 #endif // _DEFS_HPP_

```

5.11 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.

5.11.1 Enumeration Type Documentation

5.11.1.1 Direction

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

Enumerator

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

5.12 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_

```

5.13 EMA.hpp

```

00001 #pragma once
00002 #include <cstdint> // uint_fast16_t
00003 #include <limits> // std::numeric_limits
00004 #include <type_traits> // std::make_unsigned_t, make_signed_t, is_unsigned
00005
00022 template <uint8_t K, class uint_t = uint_fast32_t> class EMA {
00023 public:
00025     uint_t operator()(uint_t input) {
00026         state += input;
00027         uint_t output = (state + half) » K;
00028         state -= output;
00029         return output;
00030     }
00031
00033     constexpr static uint_t half = 1 « (K - 1);
00034
00035     void reset() { state = 0; }
00036
00037 private:
00038     uint_t state = 0;
00039 };

```

5.14 Motor.hpp File Reference

```

#include "ControlPins.hpp"
#include "CurrentSense.hpp"
#include "PinMacros.hpp"
#include "defs.hpp"
#include <ESP32Encoder.h>
#include <cstring>
#include <driver/adc.h>

```

Classes

- class [Motor](#)

This class represents the motor controlled by the microcontroller.

Macros

- `#define MOTOR1_LIMIT 32`
- `#define MOTOR2_LIMIT 33`
- `#define MOTOR1_TLIMIT 34`
- `#define MOTOR2_TLIMIT 35`
- `#define READ_POSITION_ENCODER() this->pos = distanceSensor.getCount();`
- `#define MOVE_TO_POS(setpoint, min_delta, buffer)`

Variables

- `int currentPWMChannel = 0`

5.14.1 Macro Definition Documentation

5.14.1.1 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;
    }
}
```

```
\\
\\
\\
\\
\\
```

5.15 Motor.hpp

[Go to the documentation of this file.](#)

```
00001
00003 #ifndef _MOTOR_HPP_
00004 #define _MOTOR_HPP_
00005
00006 #include "ControlPins.hpp"
00007 #include "CurrentSense.hpp"
00008 #include "PinMacros.hpp"
00009 #include "defs.hpp"
00010 #include <ESP32Encoder.h>
00011 #include <cstring>
00012 #include <driver/adc.h>
00013
00014 #define MOTOR1_LIMIT 32
00015 #define MOTOR2_LIMIT 33
00016 #define MOTOR1_TLIMIT 34
00017 #define MOTOR2_TLIMIT 35
00018
00019 #define READ_POSITION_ENCODER() this->pos = distanceSensor.getCount();
00020 #define MOVE_TO_POS(setpoint, min_delta, buffer)
00021     if (abs(pos - setpoint) > min_delta) {
00022         if (pos < setpoint) {
```

```
\\
\\
\\
```

```

00023         desiredPos = setpoint - buffer;
00024     } else if (pos > newPos) {
00025         desiredPos = setpoint + buffer;
00026     }
00027 }
00028
00029 int currentPWMChannel = 0;
00030
00031 class Motor {
00032 private:
00033     char id[16];
00034     int pwmRChannel = -1;
00035     int pwmLChannel = -1;
00036     MotorPin rPWM_Pin = MotorPin::UNASSIGNED;
00037     MotorPin lPWM_Pin = MotorPin::UNASSIGNED;
00038     MotorPin r_EN_Pin = MotorPin::UNASSIGNED;
00039     MotorPin l_EN_Pin = MotorPin::UNASSIGNED;
00040     MotorPin hall_1_Pin = MotorPin::UNASSIGNED;
00041     MotorPin hall_2_Pin = MotorPin::UNASSIGNED;
00042     MotorPin l_is_pin =
00043         MotorPin::UNASSIGNED;
00044     MotorPin r_is_pin =
00045         MotorPin::UNASSIGNED;
00046     adc1_channel_t currentSensePin = ADC1_CHANNEL_0;
00047     int frequency = PWM_FREQUENCY;
00048     int pwmResolution = 8;
00049     int desiredPos =
00050         -1;
00051     ESP32Encoder distanceSensor;
00052     CurrentSense currentSense;
00053 public:
00054     int pos = 0;
00055     int lastPos = 0;
00056     int speed = DEFAULT_MOTOR_SPEED;
00057     int totalPulseCount = 0;
00058
00059     int stopBuffer = 0;
00060
00061     int bottomLimitPin = -1;
00062     int bottomCurrentLimit = -1;
00063     int currentAlarmLimit = -1;
00064
00065     bool outOfRange = false;
00066
00067     Direction dir = Direction::STOP;
00068     bool homing = false;
00069
00070     Motor() {} // end default constructor
00071
00072     Motor(const char *name, const MotorPin rpwm, const MotorPin lpwm,
00073           const MotorPin r_en, const MotorPin l_en, const MotorPin hall_1,
00074           const MotorPin hall_2, const adc1_channel_t currentSensePin,
00075           const int totalPulses, const int freq = PWM_FREQUENCY,
00076           const int defSpeed = MIN_MOTOR_TRAVEL_SPEED,
00077           const int pwmRes = PWM_RESOLUTION_BITS, const int bottomLimitPin = -1,
00078           const int currentLimit = -1, const int alarmCurrentLimit = -1,
00079           const int stopBuffer = 0)
00080         : rPWM_Pin(rpwm), lPWM_Pin(lpwm), r_EN_Pin(r_en), l_EN_Pin(l_en),
00081           hall_1_Pin(hall_1), hall_2_Pin(hall_2),
00082           currentSensePin(currentSensePin), totalPulseCount(totalPulses),
00083           frequency(freq), speed(defSpeed), pwmResolution(pwmRes),
00084           bottomLimitPin(bottomLimitPin), bottomCurrentLimit(currentLimit),
00085           currentAlarmLimit(alarmCurrentLimit), stopBuffer(stopBuffer),
00086           outOfRange(false) {
00087         strncpy(id, name, sizeof(id) - 1);
00088         id[sizeof(id) - 1] = '\0';
00089     } // end constructor
00090
00091     void initialize() {
00092         // At least two channels are needed for the linear actuator motor
00093         if (currentPWMChannel > -1 && currentPWMChannel < 14) {
00094             pwmRChannel = currentPWMChannel++;
00095             pwmLChannel = currentPWMChannel++;
00096         }
00097
00098         ledcSetup(pwmRChannel, frequency, pwmResolution);
00099         ledcSetup(pwmLChannel, frequency, pwmResolution);
00100
00101         motorAttachPin(rPWM_Pin, pwmRChannel);
00102         Serial.printf("Attaching pin %d to RPWM Channel %d\n", rPWM_Pin,
00103                       pwmRChannel);
00104         motorAttachPin(lPWM_Pin, pwmLChannel);
00105         Serial.printf("Attaching pin %d to LPWM Channel %d\n", lPWM_Pin,
00106                       pwmLChannel);
00107
00108         pinMode(bottomLimitPin, INPUT_PULLUP);
00109     }

```

```

00143     motorPinMode(r_EN_Pin, OUTPUT);
00144     motorPinMode(l_EN_Pin, OUTPUT);
00145
00146     motorPinWrite(r_EN_Pin, HIGH);
00147     motorPinWrite(l_EN_Pin, HIGH);
00148
00149     ledcWrite(pwmRChannel, 0);
00150     ledcWrite(pwmLChannel, 0);
00151
00152     distanceSensor.attachSingleEdge(static_cast<int>(hall_1_Pin),
00153                                     static_cast<int>(hall_2_Pin));
00154     distanceSensor.clearCount();
00155     READ_POSITION_ENCODER()
00156
00157     currentSense.initialize(currentSensePin);
00158
00159     Serial.printf("Motor: %s\n"
00160                  "-----\n"
00161                  "Frequency:    %5d\n"
00162                  "Resolution:   %5d\n"
00163                  "Speed:        %5d\n"
00164                  "Position:     %5d\n"
00165                  "R_EN Pin:     %5d\n"
00166                  "L_EN Pin:     %5d\n"
00167                  "RPWM Pin:     %5d\n"
00168                  "LPWM Pin:     %5d\n"
00169                  "Hall 1 Pin:    %5d\n"
00170                  "Hall 2 Pin:    %5d\n"
00171                  "Max Position: %5d\n\n"
00172                  "Stop Buffer: %5d pulses\n\n"
00173                  "Alarm Current: %5d mA\n\n",
00174                  id, frequency, pwmResolution, speed, pos, r_EN_Pin, l_EN_Pin,
00175                  rPWM_Pin, lPWM_Pin, hall_1_Pin, hall_2_Pin, totalPulseCount,
00176                  stopBuffer, currentAlarmLimit);
00177     Serial.printf("RPWM Channel %d - LPWM Channel: %d\n\n", pwmRChannel,
00178                  pwmLChannel);
00179 }
00180
00181 void drive(const Direction motorDirection, const int specifiedSpeed = 0) {
00182     // Set the drive speed based on the specified speed or the default speed
00183     const int driveSpeed = specifiedSpeed > 0 ? specifiedSpeed : speed;
00184
00185     motorPinWrite(r_EN_Pin, HIGH);
00186     motorPinWrite(l_EN_Pin, HIGH);
00187
00188     switch (motorDirection) {
00189     case Direction::EXTEND:
00190         // Drive the motor in the extend direction
00191         ledcWrite(pwmRChannel, driveSpeed);
00192         ledcWrite(pwmLChannel, 0);
00193         break;
00194     case Direction::RETRACT:
00195         // Drive the motor in the retract direction
00196         ledcWrite(pwmRChannel, 0);
00197         ledcWrite(pwmLChannel, driveSpeed);
00198         break;
00199     default:
00200         break;
00201     }
00202
00203     // Update the last position variable
00204     lastPos = pos;
00205 }
00206
00207 void extend() {
00208     // Works as a toggle
00209     dir = (dir != Direction::EXTEND) ? Direction::EXTEND : Direction::STOP;
00210 }
00211
00212 void retract() {
00213     // Works as a toggle
00214     dir = (dir != Direction::RETRACT) ? Direction::RETRACT : Direction::STOP;
00215 }
00216
00217 void disable() {
00218     // Works as a toggle
00219     if (dir != Direction::STOP) {
00220         dir = Direction::STOP;
00221         motorPinWrite(r_EN_Pin, HIGH);
00222         motorPinWrite(l_EN_Pin, HIGH);
00223         speed = 0;
00224         Serial.printf("Disabled motor: %s\n", id);
00225     }
00226 }
00227
00228 void zero() {

```

```

00247     distanceSensor.clearCount();
00248     lastPos = pos = 0;
00249 }
00250
00251 bool hitBottom() const {
00252     const int current = getCurrent();
00253     if (debugEnabled) {
00254         Serial.printf("Current %d <=> Bottom current limit: %d\n", current,
00255             bottomCurrentLimit);
00256     }
00257     return (current >= bottomCurrentLimit) && (dir == Direction::RETRACT);
00258 }
00259
00260 bool topReached() const { return pos >= totalPulseCount; }
00261
00262 void update(const int newSpeed = (MAX_SPEED + 1)) {
00263     READ_POSITION_ENCODER()
00264
00265     bool goingPastBottom = false;
00266     // Check if the motor is going past the bottom limit
00267     if (!homing) {
00268         goingPastBottom = (pos < stopBuffer) && dir == Direction::RETRACT;
00269     } else {
00270         speed = speed <= MIN_MOTOR_TRAVEL_SPEED ? speed : MIN_MOTOR_TRAVEL_SPEED;
00271         goingPastBottom = hitBottom();
00272         Serial.printf("Motor current: %d\n", getCurrent());
00273     }
00274
00275     // Check if the motor is going past the top limit
00276     const bool goingPastTop = topReached() && dir == Direction::EXTEND;
00277
00278     // Check whether the motor is out of range
00279     outOfRange = goingPastTop || goingPastBottom;
00280
00281     // If the motor is out of range or in the STOP direction, stop it
00282     if (outOfRange || dir == Direction::STOP) {
00283         if (!homing && (dir == Direction::STOP) && pos < 0) {
00284             zero();
00285         }
00286         dir = Direction::STOP;
00287         ledcWrite(pwmRChannel, 0);
00288         ledcWrite(pwmLChannel, 0);
00289         motorPinWrite(r_EN_Pin, HIGH);
00290         motorPinWrite(l_EN_Pin, HIGH);
00291         return;
00292     }
00293
00294     // Set the motor speed based on the input
00295     if (newSpeed > MAX_SPEED || newSpeed < 0) {
00296         drive(dir, this->speed);
00297     } else {
00298         drive(dir, newSpeed);
00299     }
00300 }
00301
00302 void readPos() { READ_POSITION_ENCODER() }
00303
00304 float getNormalizedPos() {
00305     READ_POSITION_ENCODER()
00306
00307     if (totalPulseCount == 0)
00308         return 0.0f;
00309     return static_cast<float>(pos) / static_cast<float>(totalPulseCount);
00310 }
00311
00312 void displayInfo() {
00313     Serial.printf("Motor %s - Direction: %s, pos: %d, currentAlarm %d mA\n", id,
00314         directions[static_cast<int>(dir)], pos, currentAlarmLimit);
00315     Serial.printf("Motor %s - Speed: %d, desired pos: %d\n", id, speed,
00316         desiredPos);
00317     Serial.printf("Motor %s - Max hall position: %d \n\n", id, totalPulseCount);
00318 }
00319
00320 int getCurrent() const { return currentSense.getCurrent(); }
00321
00322 bool isStopped() const { return dir == Direction::STOP; }
00323
00324 void setSpeed(int newSpeed) {
00325     // Constrain the new speed value between 0 and the maximum value allowed by
00326     // the PWM resolution.
00327     speed = constrain(newSpeed, 0, MAX_SPEED);
00328 }
00329 }; // end class Motor
00330
00331 #endif // _MOTOR_HPP_

```

5.16 MotorController.hpp File Reference

```
#include <Preferences.h>
#include <math.h>
#include "EMA.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 [ALL_MOTORS](#)(operation)
- #define [ALL_MOTORS_COMMAND](#)(command) [ALL_MOTORS](#)(motors[motor].command());
- #define [RESTORE_POSITION](#)(slot) motor_controller.setPos([savedPositions](#)[slot]);
- #define [SERIAL_SAVE_POSITION](#)(slot)

5.16.1 Macro Definition Documentation

5.16.1.1 ALL_MOTORS

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

Value:

```
for (int motor = 0; motor < NUMBER\_OF\_MOTORS; motor++) {
    operation
}
```

5.16.1.2 SERIAL_SAVE_POSITION

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

Value:

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


5.17 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 #include <math.h>
00008
00009 #include "EMA.hpp"
00010 #include "Motor.hpp"
00011 #include "PIDController.hpp"
00012 #include "PinMacros.hpp"
00013 #include "defs.hpp"
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 RESTORE_POSITION(slot) motor_controller.setPos(savedPositions[slot]);
00023
00024 #define SERIAL_SAVE_POSITION(slot)                                \
00025     if (Serial.available() > 0) {                                \
00026         int new_pos = Serial.parseInt();                        \
00027         motor_controller.savePosition(slot, new_pos);           \
00028     }                                                         \
00029
00030 class MotorController {
00031 private:
00032     enum MotorRoles {
00033         LEADER,
00034         FOLLOWER
00035     };
00036
00037     const int motorPulseTotals[NUMBER_OF_MOTORS] = {LEADER_MAX_PULSES,
00038                                                         FOLLOWER_MAX_PULSES};
00039
00040     bool homing = false;
00041
00042     // const int motorPulseTotals[2] = {2055, 2050};
00043
00044     int pwmFrequency = PWM_FREQUENCY;
00045
00046     int pwmResolution = PWM_RESOLUTION_BITS;
00047
00048     int desiredPos = -1;
00049
00050     Preferences positionStorage;
00051
00052     bool stopping = false;
00053
00054     bool softStartQueued = true;
00055
00056     int leaderWorkingCurrent = -1;
00057
00058     int followerWorkingCurrent = -1;
00059
00060     int maxCurrent = -1;
00061
00062     bool currentAlarmSet = false;
00063
00064     float deltaT = 0.0f;
00065
00066     void loadPositions() {
00067         for (int slot = 0; slot < NUM_POSITION_SLOTS; slot++) {
00068             savedPositions[slot] =
00069                 positionStorage.getInt(save_position_slot_names[slot]);
00070         }
00071     }
00072
00073     void initializeMotors() {
00074         resetSoftMovement();
00075         ALL_MOTORS_COMMAND(initialize)
00076         immediateHalt();
00077     }
00078
00079     double Kp, Ki, Kd;
00080     double prevError;
00081     double integral;
00082     double maxIntegral = 1000.0; // limit for integral wind-up
00083     int intermediateSpeed = -1;

```

```

00115
00116 double control(double setpoint, double actualPosition) {
00117     double error = setpoint - actualPosition;
00118
00119     integral += error;
00120     // Prevent integral wind-up
00121     if (integral > maxIntegral)
00122         integral = maxIntegral;
00123     else if (integral < -maxIntegral)
00124         integral = -maxIntegral;
00125
00126     double derivative = error - prevError;
00127
00128     double output = Kp * error + Ki * integral + Kd * derivative;
00129     prevError = error;
00130     return output;
00131 }
00132
00133 public:
00134     int moveStart = -1;
00135
00136     int targetSpeed = -1;
00137
00138     int K_p = DEFAULT_KP;
00139
00140     PIDController pidController;
00141
00142     int defaultSpeed = DEFAULT_MOTOR_SPEED;
00143
00144     int speed = 0;
00145
00146     int leaderCurrent = 0;
00147
00148     int followerCurrent = 0;
00149
00150     int lastLeaderCurrent = 0;
00151
00152     int lastFollowerCurrent = 0;
00153
00154     int leaderCurrentVelocity = 0;
00155
00156     int followerCurrentVelocity = 0;
00157
00158     int minCurrent = 900;
00159
00160     int currentAlarmDelay = 250000;
00161
00162     int alarmCurrentVelocity = 10000;
00163
00164     Motor motors[NUMBER_OF_MOTORS];
00165
00166     Direction requestedDirection = Direction::STOP;
00167
00168     Direction systemDirection = Direction::STOP;
00169
00170     int laggingIndex = 0;
00171
00172     int leadingIndex = 0;
00173
00174     int softStart = -1;
00175
00176     int lastPWMUpdate = -1;
00177
00178     float pwmUpdateAmount = -1.0f;
00179
00180     int lastPrintTime = -1;
00181
00182     int currentUpdateInterval = CURRENT_UPDATE_INTERVAL;
00183
00184     int lastCurrentUpdate = -1;
00185
00186     int softMovingTime = -1;
00187
00188     EMA<1> leaderCurrentFilter;
00189
00190     EMA<1> followerCurrentFilter;
00191
00192     int_fast32_t samples = 0;
00193
00194     int currentOffset = 0;
00195
00196     int_fast32_t currentDifferenceSum = 0;
00197
00198     int_fast32_t currentSum = 0;
00199
00200     MotorController(const int pwmFrequency = PWM_FREQUENCY,
00201                     const int pwmResolution = PWM_RESOLUTION_BITS,

```

```

00248         const int defaultSpeed = DEFAULT_MOTOR_SPEED)
00249     : pwmFrequency(pwmFrequency), pwmResolution(pwmResolution),
00250     defaultSpeed(defaultSpeed), currentAlarmSet(false) {
00251     char buf[256];
00252     snprintf(
00253         buf, 256,
00254         "Controller Params: Frequency: %d - Resolution: %d - Duty Cycle: %d\n",
00255         pwmFrequency, pwmResolution, defaultSpeed);
00256     Serial.println(buf);
00257     speed = targetSpeed = 0;
00258     systemDirection = Direction::STOP;
00259     ALL_MOTORS(motors[motor].speed = 0;)
00260     ALL_MOTORS(motors[motor].currentAlarmLimit = 1000;)
00261 }
00262
00273 void initialize() {
00274     // Initialize the leader motor
00275     motors[0] = Motor("Leader", // Motor name
00276         MotorPin::MOTOR1_RPWM_PIN, // Right PWM pin
00277         MotorPin::MOTOR1_LPWM_PIN, // Left PWM pin
00278         MotorPin::MOTOR1_R_EN_PIN, // Right enable pin
00279         MotorPin::MOTOR1_L_EN_PIN, // Left enable pin
00280         MotorPin::MOTOR1_HALL1_PIN, // Hall sensor 1 pin
00281         MotorPin::MOTOR1_HALL2_PIN, // Hall sensor 2 pin
00282         LEADER_CURRENT_SENSE_PIN, // Current sense pin
00283         motorPulseTotals[0], // Motor pulse total
00284         PWM_FREQUENCY, // PWM frequency
00285         defaultSpeed, // Default speed
00286         pwmResolution, // PWM resolution
00287         MOTOR1_LIMIT, // Motor bottom limit
00288         minCurrent, // Motor current limit for bottom finding
00289         CURRENT_LIMIT, // Alarm current in mA
00290         LEADER_BUFFER // Buffer for retraction stop in hall pulses
00291     );
00292
00293     // Initialize the follower motor
00294     motors[1] =
00295         Motor("Follower", // Motor name
00296             MotorPin::MOTOR2_RPWM_PIN, // Right PWM pin
00297             MotorPin::MOTOR2_LPWM_PIN, // Left PWM pin
00298             MotorPin::MOTOR2_R_EN_PIN, // Right enable pin
00299             MotorPin::MOTOR2_L_EN_PIN, // Left enable pin
00300             MotorPin::MOTOR2_HALL1_PIN, // Hall sensor 1 pin
00301             MotorPin::MOTOR2_HALL2_PIN, // Hall sensor 2 pin
00302             FOLLOWER_CURRENT_SENSE_PIN, // Current sense pin
00303             motorPulseTotals[1], // Motor pulse total
00304             PWM_FREQUENCY, // PWM frequency
00305             defaultSpeed, // Default speed
00306             pwmResolution, // PWM resolution
00307             MOTOR2_LIMIT, // Motor bottom limit
00308             minCurrent, // Motor current limit for bottom finding
00309             CURRENT_LIMIT - CURRENT_OFFSET, // Alarm current in mA
00310             FOLLOWER_BUFFER // Buffer for retraction stop in hall pulse
00311         );
00312
00313     // Begin position storage
00314     positionStorage.begin("evox-tilt", false);
00315
00316     // Load the stored positions
00317     loadPositions();
00318
00319     // Initialize the motors
00320     initializeMotors();
00321
00322     // Get the current of the leader motor
00323     leaderCurrent = motors[LEADER].getCurrent();
00324
00325     // Get the current of the follower motor
00326     followerCurrent = motors[FOLLOWER].getCurrent();
00327
00328     // Set parameters for PID controller to defaults
00329     pidController.setParams(DEFAULT_KP, MAX_SPEED);
00330
00331     // Print system initialization message
00332     Serial.println("System initialized.");
00333 }
00334
00348 void startMotion(const Direction dir) {
00349     pid_on = true; // Enable PID control
00350     stopping = false;
00351     pidController.reset(); // Reset the time parameters for PID
00352     resetSoftMovement();
00353     ; // Reset soft movement
00354     /*
00355     if (speed != DEFAULT_MOTOR_SPEED) {
00356         setSpeed(DEFAULT_MOTOR_SPEED);
00357     }

```

```

00358     */
00359     speed = MIN_MOTOR_TRAVEL_SPEED;
00360     ALL_MOTORS(motors[motor].outOfRange =
00361         false; // Reset out of range flag for all motors
00362     systemDirection = dir; // Set system direction to extend
00363     requestedDirection = dir; // Set requested direction to extend
00364     softStartQueued = true;
00365     resetCurrentInformation();
00366     moveStart = micros();
00367 }
00368
00377 void extend() {
00378     startMotion(Direction::EXTEND);
00379     ALL_MOTORS_COMMAND(extend); // Send extend command to all motors
00380 }
00381
00391 void retract() {
00392     startMotion(Direction::RETRACT);
00393     ALL_MOTORS_COMMAND(retract); // Send extend command to all motors
00394 }
00395
00402 void stop() {
00403     // Reset the soft movement
00404     stopping = true;
00405     pidController.reset(); // Reset the time parameters for PID
00406     resetSoftMovement();
00407     ;
00408     softStartQueued = true; //
00409     pidController.setParams(STOP_KP);
00410
00411     // Update the requested direction to STOP
00412     requestedDirection = Direction::STOP;
00413     moveStart = micros();
00414     resetCurrentInformation();
00415     motors[LEADER].currentAlarmLimit = CURRENT_LIMIT;
00416     motors[FOLLOWER].currentAlarmLimit = CURRENT_LIMIT;
00417 }
00418
00428 void immediateHalt() {
00429     speed = targetSpeed = 0;
00430     systemDirection = Direction::STOP;
00431     requestedDirection = Direction::STOP;
00432     resetSoftMovement();
00433
00434     disableMotors();
00435     currentUpdateInterval = CURRENT_UPDATE_INTERVAL;
00436     resetCurrentInformation();
00437 }
00438
00447 void home() {
00448     // Print a message indicating that the home placeholder has been called
00449     Serial.println("Home placeholder called.");
00450
00451     // Retract the motors
00452     pid_on = true;
00453     resetSoftMovement();
00454     ;
00455     resetCurrentInformation();
00456     speed = MIN_MOTOR_TRAVEL_SPEED;
00457     targetSpeed = -1;
00458     ALL_MOTORS(motors[motor].outOfRange =
00459         false;
00460     ALL_MOTORS_COMMAND(retract)
00461     systemDirection =
00462         Direction::RETRACT;
00463     requestedDirection =
00464         Direction::RETRACT;
00465
00466     long lastTimestamp = micros();
00467     long lastPrint = lastTimestamp + 1000;
00468     int currentAlarmStart = micros();
00469     const int currentAlarmDelay = CURRENT_ALARM_DELAY;
00470     motors[LEADER].bottomCurrentLimit = minCurrent = CURRENT_LIMIT;
00471     motors[FOLLOWER].bottomCurrentLimit = minCurrent = CURRENT_LIMIT;
00472     bool followerBottomHit = false;
00473     bool leaderBottomHit = false;
00474     pid_on = false;
00475     currentAlarmSet = false;
00476     ALL_MOTORS(motors[motor].speed = MIN_MOTOR_TRAVEL_SPEED;
00477     homing = true;
00478     ALL_MOTORS(motors[motor].homing = true;
00479
00480     // Loop until both leader and follower motors are out of range
00481     for (;;) {
00482         currentAlarmSet = false;
00483         const long timestamp = micros();
00484         const int currentDeltaTime = timestamp - currentAlarmStart;

```

```

00485     // Check if it's time to update the motor current limits
00486     if (currentDeltaTime > currentAlarmDelay) {
00487         motors[LEADER].bottomCurrentLimit = 300;
00488         motors[FOLLOWER].bottomCurrentLimit = 300;
00489     }
00490
00491     if (!leaderBottomHit) {
00492         leaderBottomHit = motors[LEADER].isStopped();
00493     }
00494
00495     if (!followerBottomHit) {
00496         followerBottomHit = motors[FOLLOWER].isStopped();
00497     }
00498
00499     Serial.printf("Leader bottom hit: %d\n", leaderBottomHit);
00500     Serial.printf("Follower bottom hit: %d\n", followerBottomHit);
00501     // Check if both leader and follower motors are out of range
00502     if (leaderBottomHit && followerBottomHit) {
00503         Serial.println("Bottom hit.");
00504         pid_on = true;
00505
00506         // Zero the leader and follower motors
00507         ALL_MOTORS_COMMAND(zero)
00508
00509         // Disable all motors, reset soft movement, and set the direction to
00510         // STOP
00511         ALL_MOTORS_COMMAND(disable)
00512         resetSoftMovement();
00513         resetCurrentInformation();
00514         systemDirection = requestedDirection = Direction::STOP;
00515
00516         // Calculate the time delta and update the motor status
00517         const float deltaT = ((float)(timestamp - lastTimestamp) / 1.0e6);
00518         update(deltaT);
00519
00520         // Exit the loop
00521         break;
00522     }
00523
00524     // Calculate the time delta and update the motor status
00525     const float deltaT = ((float)(timestamp - lastTimestamp) / 1.0e6);
00526     lastTimestamp = timestamp;
00527     update(deltaT);
00528
00529     if (timestamp - lastPrint > 250000) {
00530         report();
00531     }
00532 }
00533
00534 Serial.printf("Backing up %d pulses\n", ALARM_REVERSE_AMOUNT);
00535 lastTimestamp = micros();
00536 // Extend slightly
00537 pid_on = true; // Enable PID control
00538 resetSoftMovement();
00539 ; // Reset soft movement
00540 resetCurrentInformation();
00541 speed = MIN_MOTOR_TRAVEL_SPEED;
00542 targetSpeed = -1;
00543 ALL_MOTORS(motors[motor].outOfRange =
00544     false;) // Reset out of range flag for all motors
00545 ALL_MOTORS(
00546     motors[motor].speed =
00547         MIN_MOTOR_TRAVEL_SPEED;) // Reset out of range flag for all motors
00548 ALL_MOTORS_COMMAND(extend); // Send extend command to all motors
00549 systemDirection = Direction::EXTEND; // Set system direction to extend
00550 requestedDirection = Direction::EXTEND; // Set requested direction to extend
00551
00552 for (;;) {
00553     const long timestamp = micros();
00554
00555     if (motors[LEADER].pos >= ALARM_REVERSE_AMOUNT ||
00556         motors[FOLLOWER].pos >= FOLLOWER_ALARM_REVERSE_AMOUNT) {
00557         // Zero the leader and follower motors
00558         ALL_MOTORS_COMMAND(zero)
00559         ALL_MOTORS(motors[motor].homing = false;)
00560         homing = false;
00561
00562         // Disable all motors, reset soft movement, and set the direction to
00563         // STOP
00564         Serial.println("Stopping");
00565         resetSoftMovement();
00566         ;
00567
00568         // Set the speed to 0
00569         speed = 0;
00570         targetSpeed = -1;
00571         ALL_MOTORS_COMMAND(disable)

```

```

00572         systemDirection = requestedDirection = Direction::STOP;
00573
00574         // Calculate the time delta and update the motor status
00575         const float deltaT = ((float)(timestamp - lastTimestamp) / 1.0e6);
00576         pidController.reset(); // Reset the time parameters for PID
00577         update(deltaT);
00578         resetCurrentInformation();
00579         break;
00580     }
00581     // Calculate the time delta and update the motor status
00582     const float deltaT = ((float)(timestamp - lastTimestamp) / 1.0e6);
00583     update(deltaT);
00584 }
00585 }
00586
00592 void setSpeed(const int newSpeed,
00593             const int softMovementTime = SOFT_MOVEMENT_TIME_MS) {
00594
00595     targetSpeed = newSpeed;
00596
00597     // Update the target speed
00598     Serial.printf("SetSpeed(%d)\n", newSpeed);
00599     Serial.printf("Speed: %d\n", speed);
00600     Serial.printf("Target speed: %d\n", targetSpeed);
00601
00602     // Reset the soft start and last PWM update times
00603     softStart = lastPWMUpdate = micros();
00604     const int softMovementTimeMicros = softMovementTime * MICROS_IN_MS;
00605     softMovingTime = softMovementTimeMicros;
00606     const int softMovementUpdateSteps =
00607         (softMovementTimeMicros / SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS);
00608     // Calculate the amount to update the PWM duty cycle per step
00609     pwmUpdateAmount =
00610         ceil((float)abs(targetSpeed - speed) / softMovementUpdateSteps);
00611
00612     // If the new speed is lower, make the pwmUpdateAmount negative
00613     if (targetSpeed < speed) {
00614         pwmUpdateAmount = -pwmUpdateAmount;
00615     }
00616
00617     // Print debug information if debugEnabled is true
00618     if (debugEnabled) {
00619         Serial.printf("MotorController\n"
00620                     "-----\n"
00621                     "setSpeed(%d)\n"
00622                     "speed: %3d\n"
00623                     "target speed: %3d\n"
00624                     "pwmUpdateAmount: %3.6f\n\n",
00625                     newSpeed, speed, targetSpeed, pwmUpdateAmount);
00626     }
00627
00628     if (newSpeed < speed) {
00629         // Reducing speed - use intermediate setpoint
00630         intermediateSpeed = speed - (speed - newSpeed) / 3;
00631     }
00632
00633     if (requestedDirection != Direction::RETRACT) {
00634         pidController.setParams(RETRACT_RAMP_KP);
00635     } else if (requestedDirection == Direction::EXTEND) {
00636         pidController.setParams(EXTEND_RAMP_KP);
00637     }
00638 }
00639
00647 void zero() { ALL_MOTORS_COMMAND(zero) }
00648
00657 void report() {
00658     ALL_MOTORS_COMMAND(readPos)
00659     Serial.printf("MotorController\n-----\nSpeed: %d\nTarget "
00660                 "Speed: %d\n\n",
00661                 speed, targetSpeed);
00662     pidController.report();
00663     Serial.printf("\n\n\n");
00664     displayCurrents();
00665     Serial.printf("Current Alarm Status: %s\n\n",
00666                 currentAlarmSet ? "true" : "false");
00667     ALL_MOTORS_COMMAND(displayInfo)
00668 }
00669
00675 void printCurrent() {
00676     // Check if the motors are stopped
00677     if (!motorsStopped()) {
00678         // Print the current values
00679         Serial.printf("Leader Current: %d\n", leaderCurrent);
00680         Serial.printf("Follower Current: %d\n", followerCurrent);
00681     }
00682 }
00683

```

```

00690 void savePosition(const int slot, const int position_value = -1) {
00691     // Make sure the slot index is within valid range and if no position was
00692     // manually specified, then use the current position of the leader motor
00693     const int positionToSave =
00694         position_value > -1 ? position_value : motors[LEADER].pos;
00695
00696     if (slot > 0 && slot < NUM_POSITION_SLOTS) {
00697         // Store the position value in the savedPositions array.
00698         savedPositions[slot - 1] = positionToSave;
00699
00700         // Store the position value in the positionStorage.
00701         positionStorage.putInt(save_position_slot_names[slot], positionToSave);
00702     }
00703 }
00704
00710 void setPos(const int newPos) {
00711     desiredPos = constrain(newPos, 0, motors[LEADER].totalPulseCount);
00712
00713     // Check if the current position is less than the desired position
00714     if (motors[LEADER].pos < desiredPos) {
00715         extend();
00716     }
00717     // Check if the current position is greater than the desired position
00718     else if (motors[LEADER].pos > desiredPos) {
00719         retract();
00720     }
00721 }
00722
00733 void updateCurrentReadings(const int elapsedTime) {
00734     // Store the last readings of the leader and follower currents
00735     const double lastLeaderCurrent = leaderCurrent;
00736     const double lastFollowerCurrent = followerCurrent;
00737
00738     const int lCurrent = motors[LEADER].getCurrent();
00739     const int fCurrent = motors[FOLLOWER].getCurrent();
00740
00741     // Get the filtered current readings of the leader and follower motors
00742     leaderCurrent = leaderCurrentFilter(lCurrent);
00743     followerCurrent = followerCurrentFilter(fCurrent);
00744
00745     // Calculate the time factor
00746     const double timeFactor = 1000000.0 / elapsedTime;
00747
00748     // Calculate the velocities of the leader and follower currents
00749     leaderCurrentVelocity = static_cast<int>(
00750         (leaderCurrent - lastLeaderCurrent) * timeFactor + 0.5);
00751     followerCurrentVelocity = static_cast<int>(
00752         (followerCurrent - lastFollowerCurrent) * timeFactor + 0.5);
00753 }
00754
00760 bool isStopped() const { return systemDirection == Direction::STOP; }
00761
00767 bool motorsStopped() const {
00768     // Check if the leader motor is stopped
00769     bool isLeaderStopped = motors[LEADER].isStopped();
00770
00771     // Check if the follower motor is stopped
00772     bool isFollowerStopped = motors[FOLLOWER].isStopped();
00773
00774     // Return true if both motors are stopped
00775     return isLeaderStopped && isFollowerStopped;
00776 }
00777
00783 bool currentAlarmTriggered() {
00784     return currentAlarmSet &&
00785         (leaderCurrent > motors[LEADER].currentAlarmLimit) ||
00786         (followerCurrent > motors[FOLLOWER].currentAlarmLimit);
00787 }
00788
00794 bool motorsDesynced(void) const {
00795     return abs(motors[LEADER].pos - motors[FOLLOWER].pos) > DESYNC_TOLERANCE;
00796 }
00797
00798 void handlePid() {
00799     updateLeadingAndLaggingIndices();
00800     // Speed to set the faster motor to as calculated by the PID algorithm
00801     const int adjustedSpeed = pidController.adjustSpeed(
00802         motors[leadingIndex], motors[laggingIndex], speed, deltaT);
00803
00804     motors[leadingIndex].speed = adjustedSpeed;
00805     motors[laggingIndex].speed = speed;
00806 }
00807
00814 bool motorsCloseToEndOfRange() {
00815     // Read the current position of all motors
00816     ALL_MOTORS_COMMAND(readPos)
00817

```

```

00818 // Get the normalized position of the leader motor
00819 double leaderPos = motors[LEADER].getNormalizedPos();
00820
00821 // Get the normalized position of the follower motor
00822 double followerPos = motors[FOLLOWER].getNormalizedPos();
00823
00824 // Check if the leader motor is close to the end of its range
00825 // Check if the leader motor is close to the end of its range
00826 bool leaderCloseToEnd =
00827     (leaderPos < 0.1 && motors[LEADER].dir == Direction::RETRACT) ||
00828     (leaderPos > 0.9 && motors[LEADER].dir == Direction::EXTEND);
00829
00830 // Check if the follower motor is close to the end of its range
00831 bool followerCloseToEnd =
00832     (followerPos < 0.15 && motors[FOLLOWER].dir == Direction::RETRACT) ||
00833     (followerPos > 0.85 && motors[FOLLOWER].dir == Direction::EXTEND);
00834
00835 // Return true if either motor is close to the end of its range
00836 return leaderCloseToEnd && followerCloseToEnd;
00837 }
00838
00843 void handleCurrentAlarm() {
00844     // Print debug message if debugEnabled is true
00845     displayCurrents();
00846     Serial.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
00847     Serial.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!ALARM!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
00848     Serial.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
00849
00850     /*
00851     immediateHalt();
00852     systemDirection = requestedDirection = Direction::STOP;
00853     resetSoftMovement();
00854     resetCurrentInformation();
00855
00856     */
00857     report();
00858 }
00859
00866 void updateLeadingAndLaggingIndices() {
00867     // Check if the system direction is to extend
00868     if (Direction::EXTEND == systemDirection) {
00869         // Update the lagging index based on the normalized positions of the
00870         // motors
00871         laggingIndex = (motors[LEADER].getNormalizedPos() <
00872             motors[FOLLOWER].getNormalizedPos())
00873             ? MotorRoles::LEADER
00874             : MotorRoles::FOLLOWER;
00875         // Update the leading index based on the normalized positions of the
00876         // motors
00877         leadingIndex = (motors[LEADER].getNormalizedPos() >=
00878             motors[FOLLOWER].getNormalizedPos())
00879             ? MotorRoles::LEADER
00880             : MotorRoles::FOLLOWER;
00881     }
00882     // Check if the system direction is to retract
00883     else if (Direction::RETRACT == systemDirection) {
00884         // Update the lagging index based on the normalized positions of the
00885         // motors
00886         laggingIndex = (motors[LEADER].getNormalizedPos() >
00887             motors[FOLLOWER].getNormalizedPos())
00888             ? MotorRoles::LEADER
00889             : MotorRoles::FOLLOWER;
00890         // Update the leading index based on the normalized positions of the
00891         // motors
00892         leadingIndex = (motors[LEADER].getNormalizedPos() <=
00893             motors[FOLLOWER].getNormalizedPos())
00894             ? MotorRoles::LEADER
00895             : MotorRoles::FOLLOWER;
00896     }
00897 }
00898
00905 void displayCurrents() {
00906     Serial.printf("Leader Motor Current: %d\n", leaderCurrent);
00907     // Serial.printf("Leader current velocity: %d\n", leaderCurrentVelocity);
00908     Serial.printf("Follower Motor Current: %d\n", followerCurrent);
00909     // Serial.printf("Follower current velocity: %d\n",
00910     // followerCurrentVelocity);
00911 }
00912
00924 void sampleCurrents() {
00925     const int largerCurrent = std::max(leaderCurrent, followerCurrent);
00926     if (largerCurrent > maxCurrent) {
00927         maxCurrent = largerCurrent;
00928     }
00929
00930     const int currentDifference = abs(leaderCurrent - followerCurrent);
00931     currentDifferenceSum += currentDifference;

```



```

00932     currentSum += leaderCurrent;
00933     samples++;
00934 }
00935
00947 void setCurrentLimit() {
00948     leaderWorkingCurrent = leaderCurrent;
00949
00950     const int averageCurrent = currentSum / samples;
00951
00952     const int currentSetVal = static_cast<int>{
00953         ((averageCurrent + ((leaderCurrent + followerCurrent) / 2)) / 2) *
00954         CURRENT_INCREASE_MULTIPLIER);
00955
00956     currentOffset = currentDifferenceSum / samples;
00957
00958     Serial.printf("Current alarms set to: Leader => %d, Follower => %d\n",
00959         currentSetVal, currentSetVal - currentOffset);
00960
00961     motors[LEADER].currentAlarmLimit = currentSetVal;
00962     motors[FOLLOWER].currentAlarmLimit = currentSetVal - currentOffset;
00963
00964     currentAlarmSet = true;
00965 }
00966
00976 void updateSoftMovement() {
00977     const long currentTime = micros();
00978
00979     if (targetSpeed >= 0) {
00980         // Distance from the current speed to the target speed
00981         const int speedDelta = abs(speed - targetSpeed);
00982
00983         // The time since soft movement started
00984         const long moveTimeDelta = currentTime - softStart;
00985
00986         // Calculate the time since the last PWM update
00987         const long updateTimeDelta = currentTime - lastPWMUpdate;
00988
00989         if (updateTimeDelta >= SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS) {
00990             const bool timeToUpdate = moveTimeDelta < softMovingTime;
00991             // Get the true PWM update amount based on the actual elapsed time
00992             const float updateAmount =
00993                 pwmUpdateAmount * (static_cast<float>(updateTimeDelta) /
00994                     SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS);
00995             const bool speedDeltaEnough = speedDelta >= abs(updateAmount);
00996
00997             if (timeToUpdate && speedDeltaEnough) {
00998
00999                 const float newSpeed = static_cast<float>(speed + updateAmount);
01000                 speed = static_cast<int>(floorf(newSpeed));
01001                 // Serial.printf("Speed <== %d\n", speed);
01002                 lastPWMUpdate = micros();
01003             } else {
01004                 // Set the speed to the target speed and reset the soft movement if
01005                 // time expired or there is less than one full update step until we
01006                 // reach the target speed.
01007                 speed = targetSpeed;
01008                 resetSoftMovement();
01009
01010                 if (requestedDirection == Direction::STOP) {
01011                     systemDirection = Direction::STOP;
01012
01013                     immediateHalt();
01014                     stopping = false;
01015                     report();
01016                 }
01017             }
01018         }
01019         if (intermediateSpeed >= 0) {
01020             // Transition to intermediate speed
01021             double output = control(intermediateSpeed,
01022                 speed); // Assuming speed is the current speed
01023             if (abs(output - intermediateSpeed) < 1) { // Or a small threshold
01024                 intermediateSpeed = -1;
01025             }
01026         } else {
01027             // Regular target speed tracking
01028             control(targetSpeed, speed);
01029         }
01030     }
01031 }
01032
01040 void update(const float deltaT = 0.0f) {
01041     // Get current time in microseconds
01042     const long currentTime = micros();
01043     // Calculate the time since the last current update
01044     const long currentUpdateDelta = currentTime - lastCurrentUpdate;
01045

```

```

01046     const int currentMs = currentTime / MICROS_IN_MS;
01047     const int moveStartMs = currentTime / MICROS_IN_MS;
01048     this->deltaT = deltaT;
01049
01050     if (softStartQueued && (currentTime - moveStart) > 100000) {
01051         if (!stopping) {
01052             setSpeed(DEFAULT_MOTOR_SPEED);
01053         } else {
01054             setSpeed(0, SOFT_STOP_TIME_MS);
01055         }
01056         softStartQueued = false;
01057     }
01058
01059     if (motorsStopped()) {
01060         resetCurrentInformation();
01061         resetSoftMovement();
01062         systemDirection = requestedDirection = Direction::STOP;
01063     }
01064
01065     // Immediate halt on motor desynchronization
01066     /*
01067     if (motorsDesynced()) {
01068         immediateHalt();
01069     }
01070     */
01071
01072     // Check if the system is stopped. If so, ensure current is disabled to the
01073     // motors.
01074     if (Direction::STOP == systemDirection || motorsStopped()) {
01075         ALL_MOTORS_COMMAND(disable)
01076         ALL_MOTORS_COMMAND(update)
01077         return;
01078     }
01079
01080     // If the motors are close to the end of the range of movement and are not
01081     // in a soft-stop, and aren't ramping up/down, then set the speed to end of
01082     // range speed
01083     if (motorsCloseToEndOfRange() && speed != MOTOR_END_OF_RANGE_SPEED &&
01084         targetSpeed < 0) {
01085         if (!stopping) {
01086             setSpeed(MOTOR_END_OF_RANGE_SPEED);
01087         } else {
01088             setSpeed(0, MIN_MOTOR_TRAVEL_SPEED);
01089         }
01090     }
01091
01092     // Check if it's time to update the current readings
01093     if (currentUpdateDelta >= currentUpdateInterval) {
01094         // Check if the motors are not stopped
01095         if (!motorsStopped()) {
01096             updateCurrentReadings(currentUpdateDelta);
01097             // Update the last current update time
01098             lastCurrentUpdate = currentTime;
01099
01100             // Display the current readings if debug is enabled
01101             // displayCurrents();
01102
01103             if (!currentAlarmSet) {
01104                 sampleCurrents();
01105             }
01106
01107             if ((currentTime - moveStart) > CURRENT_ALARM_DELAY) {
01108                 if (!currentAlarmSet) {
01109                     setCurrentLimit();
01110                 }
01111
01112                 if (currentAlarmTriggered()) {
01113                     handleCurrentAlarm();
01114                 }
01115             }
01116         }
01117     }
01118
01119     if (desiredPos != -1) {
01120         if (abs(desiredPos - motors[LEADER].pos) < SET_POSITION_BUFFER) {
01121             if (debugEnabled) {
01122                 Serial.printf("Desired Pos: %d - REACHED\n", desiredPos);
01123             }
01124             desiredPos = -1;
01125             stop();
01126         }
01127     }
01128
01129     // Check if the soft movement system has a target speed
01130     if (targetSpeed >= 0) {
01131         updateSoftMovement();
01132     }

```

```

01133
01134     if (pid_on) {
01135         handlePid();
01136     } else {
01137         motors[leadingIndex].speed = speed;
01138         motors[laggingIndex].speed = speed;
01139     }
01140
01141     ALL_MOTORS_COMMAND(update)
01142 }
01143
01153 void resetSoftMovement() {
01154     pwmUpdateAmount = 0;
01155     lastPWMUpdate = -1;
01156     softStart = -1;
01157     targetSpeed = -1;
01158     currentUpdateInterval = CURRENT_UPDATE_INTERVAL;
01159     pidController.setParams(DEFAULT_KP);
01160     softStartQueued = false;
01161 }
01162
01172 void resetCurrentInformation() {
01173     currentAlarmSet = false;
01174     maxCurrent = -1;
01175     samples = 0;
01176     currentDifferenceSum = 0;
01177     currentSum = 0;
01178     currentOffset = 0;
01179     resetMotorCurrentAlarms();
01180     leaderCurrentFilter.reset();
01181     followerCurrentFilter.reset();
01182 }
01183
01193 void disableMotors() {
01194     ALL_MOTORS_COMMAND(disable)
01195     ALL_MOTORS_COMMAND(update)
01196 }
01197
01207 void updateMotors() { ALL_MOTORS_COMMAND(update) }
01208
01218 void resetPid() {
01219     // Set parameters for PID controller to defaults
01220     pidController.setParams(DEFAULT_KP, MAX_SPEED);
01221 }
01222
01232 void resetMotorCurrentAlarms() {
01233     motors[LEADER].currentAlarmLimit = CURRENT_LIMIT;
01234     motors[FOLLOWER].currentAlarmLimit = CURRENT_LIMIT;
01235 }
01236 };
01237
01238 #endif // _MOTOR_CONTROLLER_HPP_

```

5.18 MotorPins.hpp File Reference

```
#include <stdint>
```

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 , [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 }

5.18.1 Enumeration Type Documentation

5.18.1.1 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
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

5.19 MotorPins.hpp

[Go to the documentation of this file.](#)

```

00001
00003 #ifndef _MOTOR_PINS_HPP_
00004 #define _MOTOR_PINS_HPP_
00005
00006 #include <stdint>
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     MOTOR2_RPWM_PIN = 5,
00044
00046     MOTOR2_LPWM_PIN = 17,
00047
00049     MOTOR2_R_EN_PIN = 16,
00050
00052     MOTOR2_L_EN_PIN = 15,
00053
00055     MOTOR2_HALL1_PIN = 14,
00056
00058     MOTOR2_HALL2_PIN = 13
00059 };
00060
00061 #endif // _MOTOR_PINS_HPP_

```

5.20 MotorsSoftMovementState.hpp File Reference

```

#include "ControllerState.hpp"
#include "MotorController.hpp"
#include "defs.hpp"
#include <Arduino.h>

```

Classes

- class [MotorsSoftMovementState](#)

Represents the motors making a soft movement.

5.21 MotorsSoftMovementState.hpp

[Go to the documentation of this file.](#)

```

00001
00003 #ifndef _MOTORS_SOFT_MOVEMENT_STATE_HPP_
00004 #define _MOTORS_SOFT_MOVEMENT_STATE_HPP_
00005
00006 #include "ControllerState.hpp"
00007 #include "MotorController.hpp"
00008 #include "defs.hpp"
00009 #include <Arduino.h>
00010
00020 class MotorsSoftMovementState : public ControllerState {
00021 public:
00022     MotorsSoftMovementState();
00023     MotorsSoftMovementState(MotorController *pMotorController);
00024
00030 void enter() {
00031     Serial.print("Motors soft movement started");
00032
00033     if (nullptr != controller) {
00034         // Reset current information
00035         controller->resetCurrentInformation();
00036
00037         // Reset soft movement
00038         controller->resetSoftMovement();
00039
00040         // Reset PID controller
00041         controller->resetPid();
00042
00043         // Set travel speed to the minimum
00044         controller->targetSpeed = MIN_MOTOR_TRAVEL_SPEED;
00045
00046         controller->requestedDirection = Direction::STOP;
00047
00048         // Save the start time of the motors
00049         controller->moveStart = micros();
00050     } else {
00051         Serial.println("MotorsSoftMovementState - No controller");
00052     }
00053 }
00054
00060 void update() {
00061     if (nullptr != controller) {
00062         controller->updateSoftMovement();
00063
00064         if (pid_on) {
00065             controller->handlePid();
00066         }
00067
00068         controller->updateMotors();
00069     } else {
00070         Serial.println("MotorsSoftMovementState - No controller");
00071     }
00072 }
00073
00079 void leave() { Serial.print("Motors soft movement stopped"); }
00080 }; // end class MotorsSoftMovementState
00081
00082 #endif // _MOTORS_STOPPING_STATE_HPP_

```

5.22 MotorsStartingState.hpp File Reference

```

#include "ControllerState.hpp"
#include "MotorController.hpp"
#include "defs.hpp"
#include <Arduino.h>

```

Classes

- class [MotorsStartingState](#)
Handles the starting state for motor control system.

5.23 MotorsStartingState.hpp

[Go to the documentation of this file.](#)

```

00001
00003 #ifndef _MOTORS_STARTING_STATE_HPP_
00004 #define _MOTORS_STARTING_STATE_HPP_
00005
00006 #include "ControllerState.hpp"
00007 #include "MotorController.hpp"
00008 #include "defs.hpp"
00009 #include <Arduino.h>
00010
00020 class MotorsStartingState : public ControllerState {
00021 public:
00022     MotorsStartingState();
00023     MotorsStartingState(MotorController *pMotorController);
00024
00030 void enter() {
00031     Serial.print("Motors starting");
00032
00033     if (nullptr != controller) {
00034         // Reset current information
00035         controller->resetCurrentInformation();
00036
00037         // Reset soft movement
00038         controller->resetSoftMovement();
00039
00040         // Reset PID controller
00041         controller->resetPid();
00042
00043         // Set travel speed to the minimum
00044         controller->speed = MIN_MOTOR_TRAVEL_SPEED;
00045
00046         // Save the start time of the motors
00047         controller->moveStart = micros();
00048     } else {
00049         Serial.println("MotorsStartingState - No controller");
00050     }
00051 }
00052
00058 void update() {
00059     if (pid_on) {
00060         controller->handlePid();
00061     }
00062
00063     controller->sampleCurrents();
00064 }
00065
00071 void leave() { Serial.print("Motors moving to soft movement state"); }
00072
00073 }; // end class MotorsStartingState
00074
00075 #endif // _MOTORS_STARTING_STATE_HPP_

```

5.24 MotorsStoppedState.hpp File Reference

```

#include "ControllerState.hpp"
#include "MotorController.hpp"
#include <Arduino.h>

```

Classes

- class [MotorsStoppedState](#)
Handles the stopped state for motor control system.

5.25 MotorsStoppedState.hpp

[Go to the documentation of this file.](#)

```

00001
00003 #ifndef _MOTORS_STOPPED_STATE_HPP_
00004 #define _MOTORS_STOPPED_STATE_HPP_
00005
00006 #include "ControllerState.hpp"
00007 #include "MotorController.hpp"
00008 #include <Arduino.h>
00009
00019 class MotorsStoppedState : public ControllerState {
00020 public:
00021     MotorsStoppedState();
00022
00023     MotorsStoppedState(MotorController *pMotorController);
00024
00030 void enter() {
00031     Serial.print("Motors Stopped");
00032
00033     if (nullptr != controller) {
00034         // Reset current information
00035         controller->resetCurrentInformation();
00036
00037         // Reset soft movement
00038         controller->resetSoftMovement();
00039
00040         // Set directions to stop
00041         controller->requestedDirection = Direction::STOP;
00042         controller->systemDirection = Direction::STOP;
00043
00044         // Disable motors
00045         controller->disableMotors();
00046     } else {
00047         Serial.println("Motors Stopped State - No controller");
00048     }
00049 }
00050
00056 void update() { controller->updateMotors(); }
00057
00063 void leave() { Serial.print("Motors starting"); }
00064 }; // end class State
00065
00066 #endif // _MOTORS_STOPPED_STATE_HPP_

```

5.26 PIDController.hpp

```

00001
00003 #ifndef _PID_CONTROLLER_HPP_
00004 #define _PID_CONTROLLER_HPP_
00005
00006 #include "defs.hpp"
00007 #include <cmath>
00008 #include <cstring>
00009 #include <limits>
00010 #include <stdio.h>
00011
00022 class PIDController {
00023 private:
00024     float errorIntegral = 0.0f;
00025     float previousMeasurement = 0.0f;
00026     float positionDifference = 0.0f;
00027     float maxSpeedAdjustmentRate = 900.0f;
00028     float filteredDerivative = 0.0f;
00029     const float alpha = 0.5; // Filter constant
00030     int K_p;
00031     float K_i = DEFAULT_KI;
00032     float K_d = DEFAULT_KD;
00033     float tau = 1.0f;
00034     int uMax;
00035     float derivative = 0.0f;
00036     float limMinInteg, limMaxInteg;
00037     float lastError = 0.0f;
00038     float followerMaxAccel =
00039         5.0f;
00040     int followerMaxSpeed = MAX_SPEED;
00041     float filteredSpeed = 0.0f;
00042
00043     const float MAX_ACCELERATION_INCREASE = 0.01f;
00044     const float MAX_ACCELERATION_LIMIT = 5.0f;
00045     const float SPEED_ALPHA = 0.5f; // The filtering constant for speed.

```

```

00046  const float SETPOINT_WEIGHT = 1.15f; // or 0.8f as per your requirement
00047  const int POSITION_DELTA_SPEED_SCALER_EXTEND = 70000;
00048  const int POSITION_DELTA_SPEED_SCALER_RETRACT = 70000;
00049
00050  const int MAX_POSITION_SCALE_VALUE = 50;
00051
00052 public:
00062  PIDController(const int kp = DEFAULT_KP, const float ki = DEFAULT_KI,
00063               const float kd = DEFAULT_KD, const float tau = 1.0f,
00064               const int uMax = MAX_SPEED)
00065      : K_p(kp), K_i(ki), K_d(kd), tau(tau), uMax(uMax),
00066        followerMaxSpeed(0.9 * uMax), // Initialize followerMaxSpeed
00067        limMinInteg(
00068            -std::numeric_limits<float>::infinity()), // Or another appropriate
00069                                                    // value
00070        limMaxInteg(
00071            std::numeric_limits<float>::infinity()), // Or another appropriate
00072                                                    // value
00073        errorIntegral(limMinInteg), // Now it is safe to use limMinInteg
00074        filteredSpeed(0.0f) {
00075
00076  if (debugEnabled) {
00077      Serial.println("PID Controller Initialized");
00078      Serial.printf("PID Parameters:\n");
00079      Serial.printf("K_p: %d\n", K_p);
00080      Serial.printf("K_i: %d\n", K_i);
00081      Serial.printf("K_d: %d\n", K_d);
00082      Serial.printf("Last Error: %f\n", lastError);
00083      Serial.printf("Error integral: %f\n", errorIntegral);
00084      Serial.printf("Max Speed: %d\n", uMax);
00085      Serial.println("-----");
00086  }
00087  }
00088
00099  void setParams(const int kpIn, const float kiIn = DEFAULT_KI,
00100               const float kdIn = DEFAULT_KD, const int uMaxIn = MAX_SPEED) {
00101      K_p = kpIn;
00102      K_i = kiIn;
00103      K_d = kdIn;
00104      uMax = uMaxIn;
00105
00106      if (fabs(kiIn) < 1e-6) { // Checks if Ki is almost zero
00107          K_i = 0.0f;
00108          limMinInteg =
00109              -std::numeric_limits<float>::infinity(); // Changed to negative
00110                                                         // infinity
00111          limMaxInteg = std::numeric_limits<float>::infinity(); // Changed to
00112                                                         // positive infinity
00113      } else {
00114          K_i = kiIn;
00115          limMinInteg = -uMax / K_i; // Recalculated the minimum limit
00116          limMaxInteg = uMax / K_i;
00117      }
00118      errorIntegral = constrain(errorIntegral, limMinInteg, limMaxInteg);
00119  }
00120
00133  int adjustSpeed(Motor &leader, Motor &follower, const int speed,
00134                const float deltaT = 0.0f) {
00135
00136      const float leadingMotorPosition = leader.getNormalizedPos();
00137      const float laggingMotorPosition = follower.getNormalizedPos();
00138      positionDifference = (laggingMotorPosition - leadingMotorPosition);
00139      const float error = SETPOINT_WEIGHT * positionDifference;
00140      const float derivativeTerm = K_d * filteredDerivative;
00141      const float integralTerm = K_i * errorIntegral;
00142      maxSpeedAdjustmentRate +=
00143          MAX_ACCELERATION_INCREASE * 1000; // scaling up the increment
00144      maxSpeedAdjustmentRate = std::min(
00145          maxSpeedAdjustmentRate, followerMaxAccel * 1000); // scale up the limit
00146
00147      // Update filtered speed before it's used in PID calculations
00148      filteredSpeed = SPEED_ALPHA * filteredSpeed + (1 - SPEED_ALPHA) * speed;
00149
00150      // Calculate the derivative and then use it to calculate the filtered
00151      // derivative
00152      if (deltaT > 0.0f) { // Avoid division by zero
00153          derivative = (error - lastError) / deltaT;
00154          filteredDerivative =
00155              alpha * filteredDerivative + (1 - alpha) * derivative;
00156      } else {
00157          derivative = 0.0f;
00158      }
00159
00160      const int proportionalTerm = static_cast<int>(round(error * K_p * 3));
00161
00162      errorIntegral += error * deltaT;
00163      errorIntegral =

```



```

00164         constrain(errorIntegral, limMinInteg, limMaxInteg); // Anti-Windup
00165
00166     const int scalarValue = leader.dir == Direction::EXTEND
00167         ? POSITION_DELTA_SPEED_SCALER_EXTEND
00168         : POSITION_DELTA_SPEED_SCALER_RETRACT;
00169
00170     const int directionSign = leader.dir == Direction::EXTEND ? 1 : -1;
00171     const int positionDeltaBoost =
00172         constrain(directionSign * positionDifference * scalarValue,
00173             -MAX_POSITION_SCALE_VALUE, MAX_POSITION_SCALE_VALUE);
00174
00175     // Serial.printf("Position Delta Boost: %d\n", positionDeltaBoost);
00176
00177     const int u = speed - (proportionalTerm + integralTerm + derivativeTerm);
00178
00179     const float maxDeltaSpeed = maxSpeedAdjustmentRate * deltaT *
00180         0.001; // scaling down the adjustment rate
00181     int adjustedDeltaSpeed = constrain(u, -maxDeltaSpeed, maxDeltaSpeed);
00182
00183     int adjustedSpeed = speed + adjustedDeltaSpeed + positionDeltaBoost;
00184
00185     // Serial.printf("Adjusted Speed: %d\n", adjustedSpeed);
00186
00187     if (debugEnabled) {
00188         // Debug prints
00189     }
00190
00191     lastError = error;
00192     previousMeasurement = laggingMotorPosition;
00193     return constrain(adjustedSpeed, MIN_SPEED, MAX_SPEED);
00194 }
00195
00196 void setFollowerMaxAccel(float newMaxAccel) {
00197     followerMaxAccel = newMaxAccel;
00198 }
00199
00200 void setFollowerMaxSpeed(int newMaxSpeed) { followerMaxSpeed = newMaxSpeed; }
00201
00202 void report() const {
00203     Serial.printf("\nPID Parameters:\n");
00204     Serial.println("-----");
00205     Serial.printf("K_p: %d\n", K_p);
00206     Serial.printf("K_i: %f\n", K_i);
00207     Serial.printf("K_d: %f\n", K_d);
00208     Serial.printf("Position Difference: %f\n", positionDifference);
00209     Serial.println("-----");
00210 }
00211
00212 void setKd(float newKd) { K_d = newKd; }
00213 void setKi(float newKi) {
00214     if (fabs(newKi) < 1e-6) { // Checks if newKi is almost zero
00215         K_i = 0.0f;
00216         limMinInteg =
00217             -std::numeric_limits<float>::infinity(); // Changed to negative
00218             // infinity
00219         limMaxInteg = std::numeric_limits<float>::infinity(); // Changed to
00220             // positive infinity
00221     } else {
00222         K_i = newKi;
00223         limMinInteg = -uMax / K_i; // Recalculated the minimum limit
00224         limMaxInteg = uMax / K_i;
00225     }
00226     errorIntegral = constrain(errorIntegral, limMinInteg, limMaxInteg);
00227 }
00228
00229 void reset() {
00230     lastError = 0.0f;
00231     errorIntegral = 0.0f;
00232 }
00233 };
00234
00235 #endif // _PID_CONTROLLER_HPP_

```

5.27 PinMacros.hpp File Reference

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

Macros

- `#define FSET_TO_ANALOG_PIN(pin, var_to_set, range_min, range_max) var_to_set = fmap(analogRead(pin), 0, MAX_ADC_VALUE, range_min, range_max)`
- `#define SET_TO_ANALOG_PIN(pin, range_min, range_max) map(analogRead(pin), 0, MAX_ADC_VALUE, range_min, range_max)`
- `#define SET_TO_ANALOG_PIN_FUNC(pin, func, range_min, range_max) func(map(analogRead(pin), 0, MAX_ADC_VALUE, 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.

5.27.1 Function Documentation

5.27.1.1 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

<i>x</i>	12-bit ADC value
<i>in_min</i>	Minimum input value
<i>in_max</i>	Maximum input value
<i>out_min</i>	Minimum output value
<i>out_max</i>	Maximum output value

Returns

Mapped output value for input value

5.28 PinMacros.hpp

[Go to the documentation of this file.](#)

```
00001
00002 #ifndef _PIN_MACROS_HPP_
```

```

00003 #define _PIN_MACROS_HPP_
00004
00005 #include "defs.hpp"
00006
00014 float fmap(float x, float in_min, float in_max, float out_min, float out_max) {
00015     return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
00016 }
00017
00018 #define FSET_TO_ANALOG_PIN(pin, var_to_set, range_min, range_max) \
00019     var_to_set = fmap(analogRead(pin), 0, MAX_ADC_VALUE, range_min, range_max)
00020
00021 #define SET_TO_ANALOG_PIN(pin, range_min, range_max) \
00022     map(analogRead(pin), 0, MAX_ADC_VALUE, range_min, range_max)
00023
00024 #define SET_TO_ANALOG_PIN_FUNC(pin, func, range_min, range_max) \
00025     func(map(analogRead(pin), 0, MAX_ADC_VALUE, range_min, range_max))
00026
00027 #define motorPinWrite(pin, value) \
00028     digitalWrite(static_cast<std::uint8_t>(pin), value)
00029
00030 #define motorPinWrite(pin, value) \
00031     digitalWrite(static_cast<std::uint8_t>(pin), value)
00032
00033 #define motorPinMode(pin, value) pinMode(static_cast<std::uint8_t>(pin), value)
00034
00035 #define motorAttachPin(pin, channel) \
00036     ledcAttachPin(static_cast<std::uint8_t>(pin), channel)
00037
00038 #define motorAnalogRead(pin) analogRead(static_cast<std::uint8_t>(pin))
00039
00040 #endif // _PIN_MACROS_HPP_

```

5.29 PotentiometerPins.hpp File Reference

```
#include <cstdint>
```

Enumerations

- enum class [PotentiometerPins](#) : std::uint8_t { [SPEED_POT_PIN](#) = 35 , [KP_POT_PIN](#) = 32 }

5.29.1 Enumeration Type Documentation

5.29.1.1 PotentiometerPins

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

Enumerator

SPEED_POT_PIN	Speed potentiometer pin
KP_POT_PIN	PID gain potentiometer pin

5.30 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 // _POTENTIOMETER_PINS_HPP_

```

5.31 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);

5.31.1 Macro Definition Documentation

5.31.1.1 LOAD_SAVED_POSITION

```

#define LOAD_SAVED_POSITION(
    position,
    response_text )

```

Value:

```

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

```

5.31.1.2 MOTOR_COMMAND

```

#define MOTOR_COMMAND(
    command,
    response_text )

```

Value:

```

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

```

5.31.1.3 SET_POS_HANDLER

```

#define SET_POS_HANDLER(
    slot )

```

Value:

```

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

```

5.31.1.4 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.";
}
```

5.32 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__
```

5.33 StateController.hpp

```
00001 #ifndef _STATE_CONTROLLER_
00002 #define _STATE_CONTROLLER_
00003
00004 #include "MotorsSoftMovementState.hpp"
00005 #include "MotorsStartingState.hpp"
00006 #include "MotorsStoppedState.hpp"
00007 #include "defs.hpp"
00008
00009 enum MotorControllerState {
00010     MOTORS_STARTING_STATE,
00011     MOTORS_SOFT_MOVEMENT_STATE,
00012     MOTORS_STOPPING_STATE,
00013     MOTORS_STOPPED_STATE
00014 };
00015 constexpr int NUMBER_OF_MOTOR_STATES = 4;
00016
00017 class StateController {
00018     MotorsStartingState motorsStartingState;
00019     MotorsSoftMovementState motorsSoftMovementState;
00020     MotorsStoppedState motorsStoppedState;
00021
00022     ControllerState *motorsStateMap[NUMBER_OF_MOTOR_STATES];
```

```
00023
00024     ControllerState *currentState = nullptr;
00025
00038     void initializeStateMap() {
00039         motorsStateMap[MOTORS_STARTING_STATE] = &motorsStartingState;
00040         motorsStateMap[MOTORS_SOFT_MOVEMENT_STATE] = &motorsSoftMovementState;
00041         motorsStateMap[MOTORS_STOPPED_STATE] = &motorsStoppedState;
00042     }
00043
00044 public:
00045     StateController();
00046     StateController(MotorController *pMotorController) {
00047         setController(pMotorController);
00048     }
00049
00057     void setController(MotorController *pMotorController) {
00058         motorsStartingState.setController(pMotorController);
00059         motorsSoftMovementState.setController(pMotorController);
00060         motorsStoppedState.setController(pMotorController);
00061     }
00062
00070     void setState(MotorControllerState newState) {
00071         if (currentState != nullptr) {
00072             currentState->leave();
00073         }
00074
00075         currentState = motorsStateMap[newState];
00076         currentState->enter();
00077     }
00078
00088     void update() {
00089         if (currentState != nullptr) {
00090             currentState->update();
00091         }
00092     }
00093 }; // end class StateController
00094
00095 #endif // _STATE_CONTROLLER_
```

Index

ADC_LOGIC_VOLTAGE
 defs.hpp, 48
adjustSpeed
 PIDController, 38
ALL_MOTORS
 MotorController.hpp, 56

Command
 Commands.hpp, 41
Commands.hpp, 41
 Command, 41
 EXTEND, 41
 GET_TILT_1, 41
 GET_TILT_2, 41
 GET_TILT_3, 42
 GET_TILT_4, 42
 GET_TILT_5, 42
 HOME, 42
 READ_LIMIT, 42
 REPORT, 41
 RETRACT, 41
 SAVE_TILT_1, 41
 SAVE_TILT_2, 41
 SAVE_TILT_3, 41
 SAVE_TILT_4, 41
 SAVE_TILT_5, 41
 SET_CURRENT_ALARM, 42
 SET_POSITION, 42
 STOP, 41
 SYSTEM_RESET, 42
 TOGGLE_LIMIT_RANGE, 42
 TOGGLE_PID, 42
 ZERO, 42
CommandType, 7
ControllerState, 7
 enter, 8
 leave, 8
 setController, 8
 update, 9
ControllerState.hpp, 43
ControlPins.hpp, 43
CURRENT_INCREASE_MULTIPLIER
 defs.hpp, 48
currentAlarmTriggered
 MotorController, 21
CurrentSense, 9
 getCurrent, 10
 initialize, 10
CurrentSense.hpp, 44

defs.hpp, 45
 ADC_LOGIC_VOLTAGE, 48
 CURRENT_INCREASE_MULTIPLIER, 48
 MOTOR_END_OF_RANGE_SPEED_DELTA, 48
 save_position_slot_names, 48
 SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS, 48
 SOFT_MOVEMENT_UPDATE_STEPS, 48
Direction, 10
 Direction.hpp, 50
Direction.hpp, 50
 Direction, 50
 EXTEND, 51
 RETRACT, 51
 STOP, 51
disableMotors
 MotorController, 21
displayCurrents
 MotorController, 21
displayInfo
 Motor, 14
drive
 Motor, 14

EMA< K, uint_t >, 11
enter
 ControllerState, 8
 MotorsSoftMovementState, 32
 MotorsStartingState, 34
 MotorsStoppedState, 36
EXTEND
 Commands.hpp, 41
 Direction.hpp, 51
extend
 MotorController, 21

fmap
 PinMacros.hpp, 74

GET_TILT_1
 Commands.hpp, 41
GET_TILT_2
 Commands.hpp, 41
GET_TILT_3
 Commands.hpp, 42
GET_TILT_4
 Commands.hpp, 42
GET_TILT_5
 Commands.hpp, 42
getCurrent

- CurrentSense, 10
- Motor, 14
- getNormalizedPos
 - Motor, 14
- hitBottom
 - Motor, 15
- HOME
 - Commands.hpp, 42
- home
 - MotorController, 21
- homing
 - Motor, 16
- immediateHalt
 - MotorController, 22
- initialize
 - CurrentSense, 10
 - MotorController, 22
- isStopped
 - Motor, 15
 - MotorController, 23
- KP_POT_PIN
 - PotentiometerPins.hpp, 75
- lastPos
 - Motor, 16
- leave
 - ControllerState, 8
 - MotorsSoftMovementState, 32
 - MotorsStartingState, 34
 - MotorsStoppedState, 36
- LOAD_SAVED_POSITION
 - RouteMacros.hpp, 76
- Motor, 11
 - displayInfo, 14
 - drive, 14
 - getCurrent, 14
 - getNormalizedPos, 14
 - hitBottom, 15
 - homing, 16
 - isStopped, 15
 - lastPos, 16
 - Motor, 13
 - pos, 16
 - setSpeed, 15
 - speed, 16
 - topReached, 15
 - totalPulseCount, 17
 - update, 16
- Motor.hpp, 51
 - MOVE_TO_POS, 52
- MOTOR1_HALL1_PIN
 - MotorPins.hpp, 68
- MOTOR1_HALL2_PIN
 - MotorPins.hpp, 68
- MOTOR1_L_EN_PIN
 - MotorPins.hpp, 68
- MOTOR1_LPWM_PIN
 - MotorPins.hpp, 68
- MOTOR1_R_EN_PIN
 - MotorPins.hpp, 68
- MOTOR1_RPWM_PIN
 - MotorPins.hpp, 68
- MOTOR2_HALL1_PIN
 - MotorPins.hpp, 68
- MOTOR2_HALL2_PIN
 - MotorPins.hpp, 68
- MOTOR2_L_EN_PIN
 - MotorPins.hpp, 68
- MOTOR2_LPWM_PIN
 - MotorPins.hpp, 68
- MOTOR2_R_EN_PIN
 - MotorPins.hpp, 68
- MOTOR2_RPWM_PIN
 - MotorPins.hpp, 68
- MOTOR_COMMAND
 - RouteMacros.hpp, 76
- MOTOR_END_OF_RANGE_SPEED_DELTA
 - defs.hpp, 48
- MotorController, 17
 - currentAlarmTriggered, 21
 - disableMotors, 21
 - displayCurrents, 21
 - extend, 21
 - home, 21
 - immediateHalt, 22
 - initialize, 22
 - isStopped, 23
 - MotorController, 20
 - motorsCloseToEndOfRange, 23
 - motorsDesynced, 23
 - motorsStopped, 23
 - printCurrent, 23
 - report, 24
 - resetCurrentInformation, 24
 - resetMotorCurrentAlarms, 24
 - resetPid, 25
 - resetSoftMovement, 25
 - retract, 26
 - sampleCurrents, 26
 - savePosition, 26
 - setCurrentLimit, 26
 - setPos, 27
 - setSpeed, 27
 - startMotion, 27
 - stop, 28
 - update, 28
 - updateCurrentReadings, 28
 - updateLeadingAndLaggingIndicies, 29
 - updateMotors, 29
 - updateSoftMovement, 29
 - zero, 30
- MotorController.hpp, 56
 - ALL_MOTORS, 56

- SERIAL_SAVE_POSITION, 56
- MotorPin
 - MotorPins.hpp, 67
- MotorPins, 30
- MotorPins.hpp, 67
 - MOTOR1_HALL1_PIN, 68
 - MOTOR1_HALL2_PIN, 68
 - MOTOR1_L_EN_PIN, 68
 - MOTOR1_LPWM_PIN, 68
 - MOTOR1_R_EN_PIN, 68
 - MOTOR1_RPWM_PIN, 68
 - MOTOR2_HALL1_PIN, 68
 - MOTOR2_HALL2_PIN, 68
 - MOTOR2_L_EN_PIN, 68
 - MOTOR2_LPWM_PIN, 68
 - MOTOR2_R_EN_PIN, 68
 - MOTOR2_RPWM_PIN, 68
 - MotorPin, 67
 - UNASSIGNED, 68
- motorsCloseToEndOfRange
 - MotorController, 23
- motorsDesynced
 - MotorController, 23
- MotorsSoftMovementState, 31
 - enter, 32
 - leave, 32
 - update, 32
- MotorsSoftMovementState.hpp, 68
- MotorsStartingState, 33
 - enter, 34
 - leave, 34
 - update, 34
- MotorsStartingState.hpp, 69
- motorsStopped
 - MotorController, 23
- MotorsStoppedState, 35
 - enter, 36
 - leave, 36
 - update, 36
- MotorsStoppedState.hpp, 70
- MOVE_TO_POS
 - Motor.hpp, 52
- PIDController, 36
 - adjustSpeed, 38
 - PIDController, 37
 - report, 38
 - reset, 38
 - setParams, 39
- PinMacros.hpp, 73
 - fmap, 74
- pos
 - Motor, 16
- PotentiometerPins
 - PotentiometerPins.hpp, 75
- PotentiometerPins.hpp, 75
 - KP_POT_PIN, 75
 - PotentiometerPins, 75
 - SPEED_POT_PIN, 75
- printCurrent
 - MotorController, 23
- READ_LIMIT
 - Commands.hpp, 42
- REPORT
 - Commands.hpp, 41
- report
 - MotorController, 24
 - PIDController, 38
- reset
 - PIDController, 38
- resetCurrentInformation
 - MotorController, 24
- resetMotorCurrentAlarms
 - MotorController, 24
- resetPid
 - MotorController, 25
- resetSoftMovement
 - MotorController, 25
- RETRACT
 - Commands.hpp, 41
 - Direction.hpp, 51
- retract
 - MotorController, 26
- RouteMacros.hpp, 76
 - LOAD_SAVED_POSITION, 76
 - MOTOR_COMMAND, 76
 - SET_POS_HANDLER, 76
 - SET_TILT, 76
- sampleCurrents
 - MotorController, 26
- save_position_slot_names
 - defs.hpp, 48
- SAVE_TILT_1
 - Commands.hpp, 41
- SAVE_TILT_2
 - Commands.hpp, 41
- SAVE_TILT_3
 - Commands.hpp, 41
- SAVE_TILT_4
 - Commands.hpp, 41
- SAVE_TILT_5
 - Commands.hpp, 41
- savePosition
 - MotorController, 26
- SERIAL_SAVE_POSITION
 - MotorController.hpp, 56
- SET_CURRENT_ALARM
 - Commands.hpp, 42
- SET_POS_HANDLER
 - RouteMacros.hpp, 76
- SET_POSITION
 - Commands.hpp, 42
- SET_TILT
 - RouteMacros.hpp, 76
- setController
 - ControllerState, 8

- StateController, [39](#)
- setCurrentLimit
 - MotorController, [26](#)
- setParams
 - PIDController, [39](#)
- setPos
 - MotorController, [27](#)
- setSpeed
 - Motor, [15](#)
 - MotorController, [27](#)
- setState
 - StateController, [40](#)
- SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS
 - defs.hpp, [48](#)
- SOFT_MOVEMENT_UPDATE_STEPS
 - defs.hpp, [48](#)
- speed
 - Motor, [16](#)
- SPEED_POT_PIN
 - PotentiometerPins.hpp, [75](#)
- startMotion
 - MotorController, [27](#)
- StateController, [39](#)
 - setController, [39](#)
 - setState, [40](#)
 - update, [40](#)
- STOP
 - Commands.hpp, [41](#)
 - Direction.hpp, [51](#)
- stop
 - MotorController, [28](#)
- SYSTEM_RESET
 - Commands.hpp, [42](#)
- TOGGLE_LIMIT_RANGE
 - Commands.hpp, [42](#)
- TOGGLE_PID
 - Commands.hpp, [42](#)
- topReached
 - Motor, [15](#)
- totalPulseCount
 - Motor, [17](#)
- UNASSIGNED
 - MotorPins.hpp, [68](#)
- update
 - ControllerState, [9](#)
 - Motor, [16](#)
 - MotorController, [28](#)
 - MotorsSoftMovementState, [32](#)
 - MotorsStartingState, [34](#)
 - MotorsStoppedState, [36](#)
 - StateController, [40](#)
- updateCurrentReadings
 - MotorController, [28](#)
- updateLeadingAndLaggingIndices
 - MotorController, [29](#)
- updateMotors
 - MotorController, [29](#)
- updateSoftMovement
 - MotorController, [29](#)
- ZERO
 - Commands.hpp, [42](#)
- zero
 - MotorController, [30](#)