Motor Control Firmware

Generated by Doxygen 1.9.8

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

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

Chapter 2

Class Index

2.1 Class List

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

Chapter 3

File Index

3.1 File List

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

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:

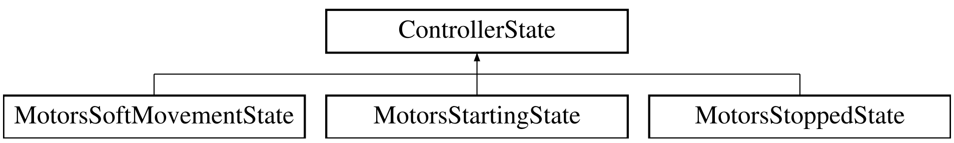
- Commands.hpp

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

pMotorController	The MotorController to set.

Exceptions

None	

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 pLogic↩ Voltage=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

None	

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

pCurrentSensePin	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

K	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 total↩ Pulses, 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 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 )  [inline]
```

The constructor for the motor controlled by the microcontroller.

Parameters

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

Copy name of linear actuator into ID field

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

motorDirection	the direction in which the motor should be driven
specifiedSpeed	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

newSpeed	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

newSpeed	the new speed value to set (default: MAX_SPEED + 1)

Exceptions

None	

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 speed/(2^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 updateLeadingAndLaggingIndicies ()

  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 proprotional gain for the PID controller
- PIDController **pidController**

  The PID controller for the motor synchonization.
- 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

PWM_FREQUENCY	the frequency of the PWM signal
PWM_RESOLUTION_BITS	the resolution of the PWM signal in bits
DEFAULT_MOTOR_SPEED	the default speed of the motor

Returns

none

Exceptions

none	

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

None	

Returns

None

Exceptions

None	

4.7.3.16 resetPid()

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

Reset the PID controller.

Parameters

None	

Returns

None

Exceptions

None	

4.7.3.17 resetSoftMovement()

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

Reset the soft movement.

Parameters

None	

Returns

None

Exceptions

None	

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

slot	The slot index.
position_value	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

None	

Returns

None

Exceptions

None	

4.7.3.22 setPos()

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

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

Parameters

newPos	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

newSpeed	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

dir	The direction in which to start the motion

Exceptions

None	

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

deltaT	the time interval since the last update (default: 0.0f)

Exceptions

None	

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

elapsedTime	the elapsed time in microseconds

Returns

void

Exceptions

None	

4.7.3.28 updateLeadingAndLaggingIndicies()

```
void MotorController::updateLeadingAndLaggingIndicies ( )  [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:

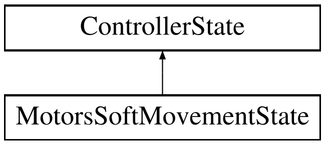
- MotorPins.hpp

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:

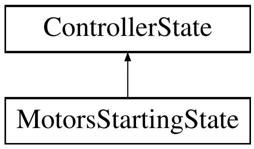
- MotorsSoftMovementState.hpp

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:

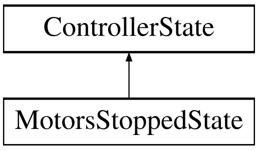
- MotorsStartingState.hpp

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 u↩ MaxIn=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

kp	The proportional gain
ki	The integral gain
kd	The derivative gain
tau	The controller's low-pass filter time constant
uMax	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

leader	The leader motor
follower	The follower motor
speed	The reference to the speed variable
deltaT	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

None	

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

kpIn	the proportional gain parameter
kiIn	the integral gain parameter
kdIn	the derivative gain parameter
u↩ MaxIn	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

pMotorController	the MotorController object to set

Exceptions

None	

4.13.1.2 setState()

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

Sets the state of the controller to a new state.

Parameters

newState	the new state to set the controller to

Exceptions

None	

4.13.1.3 update()

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

Updates the current state of the motor controller system.

Parameters

None	

Returns

None

Exceptions

None	

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.

```
#ifndef _COMMANDS_HPP_
#define _COMMANDS_HPP_
#include <cstdint>
enum class Command : std::uint32_t {
  RETRACT = 17,
  EXTEND,
  REPORT,
  STOP,
  SAVE_TILT_1,
  SAVE_TILT_2,
  SAVE_TILT_3,
  SAVE_TILT_4,
  SAVE_TILT_5,
  GET_TILT_1,
  GET_TILT_2,
  GET_TILT_3,
  GET_TILT_4,
  GET_TILT_5,
  ZERO,
  SYSTEM_RESET,
  TOGGLE_PID,
  HOME,
  TOGGLE_LIMIT_RANGE,
  READ_LIMIT,
  SET_POSITION,
  SET_CURRENT_ALARM,
};
#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.

```

#ifndef _CONTROLLER_STATE_HPP_
#define _CONTROLLER_STATE_HPP_
#include "MotorController.hpp"

class ControllerState {
protected:
  MotorController *controller;

public:
  ControllerState(MotorController *pMotorController = nullptr)
      : controller(pMotorController) {}

  void setController(MotorController *pMotorController) {
    controller = pMotorController;
  }

  virtual void enter() = 0;

  virtual void update() = 0;

  virtual void leave() = 0;

  virtual ~ControllerState() = default;
}; // end class State

#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.

```

#ifndef _CONTROL_PINS_HPP_
#define _CONTROL_PINS_HPP_

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

#define LEADER_CURRENT_SENSE_PIN ADC1_CHANNEL_0

#define FOLLOWER_CURRENT_SENSE_PIN ADC1_CHANNEL_3

#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.

```

#ifndef _CURRENT_SENSE_HPP_
#define _CURRENT_SENSE_HPP_

#include <cmath>
#include <driver/adc.h>
#include <stdint.h>

#include "ControlPins.hpp"
#include "defs.hpp"

class CurrentSense {
private:
  int_fast32_t CALIBRATE_ITERATIONS_SHIFT = 14;
  int_fast32_t SAMPLE_CURRENT_ITERATIONS_SHIFT = 6;

  int32_t MV_PER_AMP = static_cast<int32_t>(185 * 1.132);

  // negitive current flow.
  int32_t ACS_OFFSET = 1885;

  adc1_channel_t currentSensePin;
  double logicVoltage = ADC_LOGIC_VOLTAGE;
  int32_t maxAdcValue = MAX_ADC_VALUE;

public:
  CurrentSense(const adc1_channel_t pCurrentSensePin = ADC1_CHANNEL_0,
               const double pLogicVoltage = ADC_LOGIC_VOLTAGE,
               const int32_t pMaxAdcValue = MAX_ADC_VALUE)
      : currentSensePin(pCurrentSensePin), logicVoltage(pLogicVoltage),
        maxAdcValue(pMaxAdcValue) {}

  void initialize(const adc1_channel_t pCurrentSensePin = ADC1_CHANNEL_0) {
```

```
    // Set the current sensing pin
    currentSensePin = pCurrentSensePin;

    // Configure ADC settings
    adc1_config_width(ADC_WIDTH_12Bit);
    adc1_config_channel_atten(currentSensePin, ADC_ATTEN_DB_11);

    Serial.print("Pin: ");
    Serial.println(static_cast<uint8_t>(currentSensePin));
    Serial.print("Logic Voltage: ");
    Serial.println(logicVoltage);
    Serial.print("Max ADC Value: ");
    Serial.println(maxAdcValue);
    Serial.print("mV per A: ");
    Serial.println(MV_PER_AMP);

    // Calibrate ACS offset
    const int iterations = 1 « CALIBRATE_ITERATIONS_SHIFT;
    int32_t adcSum = 0;

    for (int32_t i = 0; i < iterations; i++) {
      adcSum += adc1_get_raw(currentSensePin);
    }

    ACS_OFFSET = adcSum » CALIBRATE_ITERATIONS_SHIFT;

    Serial.printf("ACS Offset: %d\n", ACS_OFFSET);
  }

  int getCurrent() const {
    // Number of iterations for current sampling
    const int32_t iterations = 1 « SAMPLE_CURRENT_ITERATIONS_SHIFT;

    int32_t currentSum = 0;

    // Perform current sampling iterations
    for (int i = 0; i < iterations; i++) {
      // Calculate ADC offset
      const int adcOffset = adc1_get_raw(currentSensePin) - ACS_OFFSET;
      // Calculate voltage delta
      const double voltageDelta = (adcOffset * (logicVoltage / maxAdcValue));
      // Accumulate current sum
      currentSum += static_cast<int>(voltageDelta * 1000000.0 / MV_PER_AMP);
    }

    // Calculate average current
    const double averageCurrent = std::abs(
        static_cast<double>(currentSum » SAMPLE_CURRENT_ITERATIONS_SHIFT));

    // Return average current as an integer
    return static_cast<int>(averageCurrent);
  } // end method getCurrent
};  // end class CurrentSense

#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 pusles 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 $2^\wedge PWM\_RESOLUTION\_BITS$ 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.

```

#ifndef _DEFS_HPP_
#define _DEFS_HPP_

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

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

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

constexpr int NUM_POSITION_SLOTS = 5;

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

int savedPositions[NUM_POSITION_SLOTS] = {0, 0, 0, 0, 0};

bool pid_on = true;

bool limit_range = true;

constexpr int FORMAT_SPIFFS_IF_FAILED = true;

constexpr int NUMBER_OF_MOTORS = 2;

constexpr int LEADER_MAX_PULSES = 2845;

constexpr int FOLLOWER_MAX_PULSES = 2845;

bool debugEnabled = false;

constexpr int PWM_FREQUENCY = 20000;

constexpr int PWM_RESOLUTION_BITS = 8;

constexpr int ADC_RESOLUTION_BITS = 12;

constexpr int DEFAULT_MOTOR_SPEED = (1 « PWM_RESOLUTION_BITS) - 1;

constexpr int MOTOR_END_OF_RANGE_SPEED = 155;

constexpr int MIN_MOTOR_TRAVEL_SPEED = 105;

constexpr int MOTOR_END_OF_RANGE_SPEED_DELTA =
    (DEFAULT_MOTOR_SPEED - MOTOR_END_OF_RANGE_SPEED);

constexpr int SET_POSITION_BUFFER = 3;

constexpr int MILLIS_IN_SEC = 1000;

constexpr int MICROS_IN_MS = 1000;

constexpr int MICROS_IN_SEC = (MILLIS_IN_SEC * MICROS_IN_MS);

constexpr int SOFT_MOVEMENT_TIME_MS = 18000;

constexpr int SOFT_MOVEMENT_MICROS = (SOFT_MOVEMENT_TIME_MS * MICROS_IN_MS);

constexpr int SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS =
    SOFT_MOVEMENT_TIME_MS / 20;
```

```
constexpr int SOFT_MOVEMENT_UPDATE_STEPS =
    (SOFT_MOVEMENT_MICROS / SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS);
constexpr int MAX_SPEED = ((1 « (PWM_RESOLUTION_BITS)) - 1);
constexpr int MIN_SPEED = 0;
constexpr float ADC_LOGIC_VOLTAGE = 3.3f;
constexpr int MAX_ADC_VALUE = (1 « (ADC_RESOLUTION_BITS)) - 1;
constexpr int CURRENT_ALARM_AMOUNT = 1400;
constexpr int CURRENT_LIMIT = 2000;
constexpr int CURRENT_OFFSET = 17;
constexpr int CURRENT_UPDATE_INTERVAL = 10000;
constexpr int CURRENT_ALARM_DELAY = 2500000;
constexpr int ALARM_REVERSE_AMOUNT = 15;
constexpr int FOLLOWER_ALARM_REVERSE_AMOUNT = (ALARM_REVERSE_AMOUNT + 30);
constexpr int LEADER_BUFFER = 3;
constexpr int FOLLOWER_BUFFER = 3;
constexpr int DESYNC_TOLERANCE = 20;
constexpr float PID_ALPHA = 33.333333f;
constexpr int DEFAULT_KP = 79999;
constexpr int RETRACT_KP = 79999;
constexpr int STOP_KP = 79999;
constexpr int EXTEND_RAMP_KP = 79999;
constexpr int RETRACT_RAMP_KP = 79999;
constexpr float DEFAULT_KI = ((DEFAULT_KP / PID_ALPHA) * 6);
constexpr float DEFAULT_KD = (DEFAULT_KP / (PID_ALPHA * 1.5));
constexpr int CURRENT_INCREASE_TOLERANCE_PERCENTAGE = 30;
constexpr float CURRENT_INCREASE_MULTIPLIER =
    1 + (CURRENT_INCREASE_TOLERANCE_PERCENTAGE / 100.0f);
constexpr int SOFT_STOP_TIME_MS = 80;
#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.

```
#ifndef _DIRECTION_HPP_
#define _DIRECTION_HPP_

enum class Direction {
  EXTEND = 0,

  STOP,

  RETRACT
};

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

#endif // _DIRECTION_HPP_
```

5.13 EMA.hpp

```
#pragma once
#include <cstdint>     // uint_fast16_t
#include <limits>      // std::numeric_limits
#include <type_traits> // std::make_unsigned_t, make_signed_t, is_unsigned

template <uint8_t K, class uint_t = uint_fast32_t> class EMA {
public:
  uint_t operator()(uint_t input) {
    state += input;
    uint_t output = (state + half) » K;
    state -= output;
    return output;
  }

  constexpr static uint_t half = 1 « (K - 1);

  void reset() { state = 0; }

private:
  uint_t state = 0;
};
```

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.

```

#ifndef _MOTOR_HPP_
#define _MOTOR_HPP_

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

#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)                                  \
  if (abs(pos - setpoint) > min_delta) {                                          \
    if (pos < setpoint) {                                                         \
```

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

int currentPWMChannel = 0;

class Motor {
private:
  char id[16];
  int pwmRChannel = -1;
  int pwmLChannel = -1;
  MotorPin rPWM_Pin = MotorPin::UNASSIGNED;
  MotorPin lPWM_Pin = MotorPin::UNASSIGNED;
  MotorPin r_EN_Pin = MotorPin::UNASSIGNED;
  MotorPin l_EN_Pin = MotorPin::UNASSIGNED;
  MotorPin hall_1_Pin = MotorPin::UNASSIGNED;
  MotorPin hall_2_Pin = MotorPin::UNASSIGNED;
  MotorPin l_is_pin =
      MotorPin::UNASSIGNED;
  MotorPin r_is_pin =
      MotorPin::UNASSIGNED;
  adc1_channel_t currentSensePin = ADC1_CHANNEL_0;
  int frequency = PWM_FREQUENCY;
  int pwmResolution = 8;
  int desiredPos =
      -1;
  ESP32Encoder distanceSensor;
  CurrentSense currentSense;
public:
  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;

  Motor() {} // end default constructor

  Motor(const char *name, const MotorPin rpwm, const MotorPin lpwm,
        const MotorPin r_en, const MotorPin l_en, const MotorPin hall_1,
        const MotorPin hall_2, const 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)
      : rPWM_Pin(rpwm), lPWM_Pin(lpwm), r_EN_Pin(r_en), l_EN_Pin(l_en),
        hall_1_Pin(hall_1), hall_2_Pin(hall_2),
        currentSensePin(currentSensePin), totalPulseCount(totalPulses),
        frequency(freq), speed(defSpeed), pwmResolution(pwmRes),
        bottomLimitPin(bottomLimitPin), bottomCurrentLimit(currentLimit),
        currentAlarmLimit(alarmCurrentLimit), stopBuffer(stopBuffer),
        outOfRange(false) {
    strncpy(id, name, sizeof(id) - 1);
    id[sizeof(id) - 1] = '\0';
  } // end constructor

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

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

    motorAttachPin(rPWM_Pin, pwmRChannel);
    Serial.printf("Attaching pin %d to RPWM Channel %d\n", rPWM_Pin,
                  pwmRChannel);
    motorAttachPin(lPWM_Pin, pwmLChannel);
    Serial.printf("Attaching pin %d to LPWM Channel %d\n\n", lPWM_Pin,
                  pwmLChannel);

    pinMode(bottomLimitPin, INPUT_PULLUP);

```

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

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

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

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

  currentSense.initialize(currentSensePin);

  Serial.printf("Motor: %s\n"
                "--------------------\n"
                "Frequency:    %5d\n"
                "Resolution:   %5d\n"
                "Speed:        %5d\n"
                "Position:     %5d\n"
                "R_EN Pin:     %5d\n"
                "L_EN Pin:     %5d\n"
                "RPWM Pin:     %5d\n"
                "LPWM Pin:     %5d\n"
                "Hall 1 Pin:   %5d\n"
                "Hall 2 Pin:   %5d\n"
                "Max Position: %5d\n\n"
                "Stop Buffer: %5d pulses\n\n"
                "Alarm Current: %5d mA\n\n",
                id, frequency, pwmResolution, speed, pos, r_EN_Pin, l_EN_Pin,
                rPWM_Pin, lPWM_Pin, hall_1_Pin, hall_2_Pin, totalPulseCount,
                stopBuffer, currentAlarmLimit);
  Serial.printf("RPWM Channel %d - LPWM Channel: %d\n\n", pwmRChannel,
                pwmLChannel);
}

void drive(const Direction motorDirection, const int specifiedSpeed = 0) {
  // Set the drive speed based on the specified speed or the default speed
  const int driveSpeed = specifiedSpeed > 0 ? specifiedSpeed : speed;

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

  switch (motorDirection) {
  case Direction::EXTEND:
    // Drive the motor in the extend direction
    ledcWrite(pwmRChannel, driveSpeed);
    ledcWrite(pwmLChannel, 0);
    break;
  case Direction::RETRACT:
    // Drive the motor in the retract direction
    ledcWrite(pwmRChannel, 0);
    ledcWrite(pwmLChannel, driveSpeed);
    break;

  default:
    break;
  }

  // Update the last position variable
  lastPos = pos;
}

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

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

void disable() {
  // Works as a toggle
  if (dir != Direction::STOP) {
    dir = Direction::STOP;
    motorPinWrite(r_EN_Pin, HIGH);
    motorPinWrite(l_EN_Pin, HIGH);
    speed = 0;
    Serial.printf("Disabled motor: %s\n", id);
  }
}

void zero() {
```

```
    distanceSensor.clearCount();
    lastPos = pos = 0;
  }

  bool hitBottom() const {
    const int current = getCurrent();
    if (debugEnabled) {
      Serial.printf("Current %d <=> Bottom current limit: %d\n", current,
                    bottomCurrentLimit);
    }
    return (current >= bottomCurrentLimit) && (dir == Direction::RETRACT);
  }

  bool topReached() const { return pos >= totalPulseCount; }

  void update(const int newSpeed = (MAX_SPEED + 1)) {
    READ_POSITION_ENCODER()

    bool goingPastBottom = false;
    // Check if the motor is going past the bottom limit
    if (!homing) {
      goingPastBottom = (pos < stopBuffer) && dir == Direction::RETRACT;
    } else {
      speed = speed <= MIN_MOTOR_TRAVEL_SPEED ? speed : MIN_MOTOR_TRAVEL_SPEED;
      goingPastBottom = hitBottom();
      Serial.printf("Motor current: %d\n", getCurrent());
    }

    // Check if the motor is going past the top limit
    const bool goingPastTop = topReached() && dir == Direction::EXTEND;

    // Check whether the motor is out of range
    outOfRange = goingPastTop || goingPastBottom;

    // If the motor is out of range or in the STOP direction, stop it
    if (outOfRange || dir == Direction::STOP) {
      if (!homing && (dir == Direction::STOP) && pos < 0) {
        zero();
      }
      dir = Direction::STOP;
      ledcWrite(pwmRChannel, 0);
      ledcWrite(pwmLChannel, 0);
      motorPinWrite(r_EN_Pin, HIGH);
      motorPinWrite(l_EN_Pin, HIGH);
      return;
    }

    // Set the motor speed based on the input
    if (newSpeed > MAX_SPEED || newSpeed < 0) {
      drive(dir, this->speed);
    } else {
      drive(dir, newSpeed);
    }
  }

  void readPos() { READ_POSITION_ENCODER() }

  float getNormalizedPos() {
    READ_POSITION_ENCODER()

    if (totalPulseCount == 0)
      return 0.0f;
    return static_cast<float>(pos) / static_cast<float>(totalPulseCount);
  }

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

  int getCurrent() const { return currentSense.getCurrent(); }

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

  void setSpeed(int newSpeed) {
    // Constrain the new speed value between 0 and the maximum value allowed by
    // the PWM resolution.
    speed = constrain(newSpeed, 0, MAX_SPEED);
  }
}; // end class Motor

#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.

```

#ifndef _MOTOR_CONTROLLER_HPP_
#define _MOTOR_CONTROLLER_HPP_

#include <Preferences.h>
#include <math.h>

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

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

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

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

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

class MotorController {
private:
  enum MotorRoles {
    LEADER,
    FOLLOWER
  };

  const int motorPulseTotals[NUMBER_OF_MOTORS] = {LEADER_MAX_PULSES,
                                                  FOLLOWER_MAX_PULSES};

  bool homing = false;

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

  int pwmFrequency = PWM_FREQUENCY;

  int pwmResolution = PWM_RESOLUTION_BITS;

  int desiredPos = -1;

  Preferences positionStorage;

  bool stopping = false;

  bool softStartQueued = true;

  int leaderWorkingCurrent = -1;

  int followerWorkingCurrent = -1;

  int maxCurrent = -1;

  bool currentAlarmSet = false;

  float deltaT = 0.0f;

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

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

  double Kp, Ki, Kd;
  double prevError;
  double integral;
  double maxIntegral = 1000.0; // limit for integral wind-up
  int intermediateSpeed = -1;
```

```
  double control(double setpoint, double actualPosition) {
    double error = setpoint - actualPosition;

    integral += error;
    // Prevent integral wind-up
    if (integral > maxIntegral)
      integral = maxIntegral;
    else if (integral < -maxIntegral)
      integral = -maxIntegral;

    double derivative = error - prevError;

    double output = Kp * error + Ki * integral + Kd * derivative;
    prevError = error;
    return output;
  }

public:
  int moveStart = -1;

  int targetSpeed = -1;

  int K_p = DEFAULT_KP;

  PIDController pidController;

  int defaultSpeed = DEFAULT_MOTOR_SPEED;

  int speed = 0;

  int leaderCurrent = 0;

  int followerCurrent = 0;

  int lastLeaderCurrent = 0;

  int lastFollowerCurrent = 0;

  int leaderCurrentVelocity = 0;

  int followerCurrentVelocity = 0;

  int minCurrent = 900;

  int currentAlarmDelay = 250000;

  int alarmCurrentVelocity = 10000;

  Motor motors[NUMBER_OF_MOTORS];

  Direction requestedDirection = Direction::STOP;

  Direction systemDirection = Direction::STOP;

  int laggingIndex = 0;

  int leadingIndex = 0;

  int softStart = -1;

  int lastPWMUpdate = -1;

  float pwmUpdateAmount = -1.0f;

  int lastPrintTime = -1;

  int currentUpdateInterval = CURRENT_UPDATE_INTERVAL;

  int lastCurrentUpdate = -1;

  int softMovingTime = -1;

  EMA<1> leaderCurrentFilter;

  EMA<1> followerCurrentFilter;

  int_fast32_t samples = 0;

  int currentOffset = 0;

  int_fast32_t currentDifferenceSum = 0;

  int_fast32_t currentSum = 0;

  MotorController(const int pwmFrequency = PWM_FREQUENCY,
                  const int pwmResolution = PWM_RESOLUTION_BITS,
```

```
                   const int defaultSpeed = DEFAULT_MOTOR_SPEED)
      : pwmFrequency(pwmFrequency), pwmResolution(pwmResolution),
        defaultSpeed(defaultSpeed), currentAlarmSet(false) {
    char buf[256];
    snprintf(
        buf, 256,
        "Controller Params: Frequency: %d - Resolution: %d - Duty Cycle: %d\n",
        pwmFrequency, pwmResolution, defaultSpeed);
    Serial.println(buf);
    speed = targetSpeed = 0;
    systemDirection = Direction::STOP;
    ALL_MOTORS(motors[motor].speed = 0;)
    ALL_MOTORS(motors[motor].currentAlarmLimit = 1000;)
  }

  void initialize() {
    // Initialize the leader motor
    motors[0] = Motor("Leader",                    // Motor name
                      MotorPin::MOTOR1_RPWM_PIN,   // Right PWM pin
                      MotorPin::MOTOR1_LPWM_PIN,   // Left PWM pin
                      MotorPin::MOTOR1_R_EN_PIN,   // Right enable pin
                      MotorPin::MOTOR1_L_EN_PIN,   // Left enable pin
                      MotorPin::MOTOR1_HALL1_PIN,  // Hall sensor 1 pin
                      MotorPin::MOTOR1_HALL2_PIN,  // Hall sensor 2 pin
                      LEADER_CURRENT_SENSE_PIN,    // Current sense pin
                      motorPulseTotals[0],         // Motor pulse total
                      PWM_FREQUENCY,               // PWM frequency
                      defaultSpeed,                // Default speed
                      pwmResolution,               // PWM resolution
                      MOTOR1_LIMIT,                // Motor bottom limit
                      minCurrent,     // Motor current limit for bottom finding
                      CURRENT_LIMIT,  // Alarm current in mA
                      LEADER_BUFFER // Buffer for retraction stop in hall pulses
    );

    // Initialize the follower motor
    motors[1] =
        Motor("Follower",                    // Motor name
              MotorPin::MOTOR2_RPWM_PIN,   // Right PWM pin
              MotorPin::MOTOR2_LPWM_PIN,   // Left PWM pin
              MotorPin::MOTOR2_R_EN_PIN,   // Right enable pin
              MotorPin::MOTOR2_L_EN_PIN,   // Left enable pin
              MotorPin::MOTOR2_HALL1_PIN,  // Hall sensor 1 pin
              MotorPin::MOTOR2_HALL2_PIN,  // Hall sensor 2 pin
              FOLLOWER_CURRENT_SENSE_PIN,  // Current sense pin
              motorPulseTotals[1],         // Motor pulse total
              PWM_FREQUENCY,               // PWM frequency
              defaultSpeed,                // Default speed
              pwmResolution,               // PWM resolution
              MOTOR2_LIMIT,                // Motor bottom limit
              minCurrent, // Motor current limit for bottom finding
              CURRENT_LIMIT - CURRENT_OFFSET, // Alarm current in mA
              FOLLOWER_BUFFER // Buffer for retraction stop in hall pulse
        );

    // Begin position storage
    positionStorage.begin("evox-tilt", false);

    // Load the stored positions
    loadPositions();

    // Initialize the motors
    initializeMotors();

    // Get the current of the leader motor
    leaderCurrent = motors[LEADER].getCurrent();

    // Get the current of the follower motor
    followerCurrent = motors[FOLLOWER].getCurrent();

    // Set parameters for PID controller to defaults
    pidController.setParams(DEFAULT_KP, MAX_SPEED);

    // Print system initialization message
    Serial.println("System initialized.");
  }

  void startMotion(const Direction dir) {
    pid_on = true; // Enable PID control
    stopping = false;
    pidController.reset(); // Reset the time parameters for PID
    resetSoftMovement();
    ; // Reset soft movement
    /*
    if (speed != DEFAULT_MOTOR_SPEED) {
      setSpeed(DEFAULT_MOTOR_SPEED);
    }
```

```
  */
  speed = MIN_MOTOR_TRAVEL_SPEED;
  ALL_MOTORS(motors[motor].outOfRange =
                 false;)     // Reset out of range flag for all motors
  systemDirection = dir;     // Set system direction to extend
  requestedDirection = dir; // Set requested direction to extend
  softStartQueued = true;
  resetCurrentInformation();
  moveStart = micros();
}

void extend() {
  startMotion(Direction::EXTEND);
  ALL_MOTORS_COMMAND(extend); // Send extend command to all motors
}

void retract() {
  startMotion(Direction::RETRACT);
  ALL_MOTORS_COMMAND(retract); // Send extend command to all motors
}

void stop() {
  // Reset the soft movement
  stopping = true;
  pidController.reset(); // Reset the time parameters for PID
  resetSoftMovement();
  ;
  softStartQueued = true; //
  pidController.setParams(STOP_KP);

  // Update the requested direction to STOP
  requestedDirection = Direction::STOP;
  moveStart = micros();
  resetCurrentInformation();
  motors[LEADER].currentAlarmLimit = CURRENT_LIMIT;
  motors[FOLLOWER].currentAlarmLimit = CURRENT_LIMIT;
}

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

  disableMotors();
  currentUpdateInterval = CURRENT_UPDATE_INTERVAL;
  resetCurrentInformation();
}

void home() {
  // Print a message indicating that the home placeholder has been called
  Serial.println("Home placeholder called.");

  // Retract the motors
  pid_on = true;
  resetSoftMovement();
  ;
  resetCurrentInformation();
  speed = MIN_MOTOR_TRAVEL_SPEED;
  targetSpeed = -1;
  ALL_MOTORS(motors[motor].outOfRange =
                 false;)
  ALL_MOTORS_COMMAND(retract)
  systemDirection =
      Direction::RETRACT;
  requestedDirection =
      Direction::RETRACT;

  long lastTimestamp = micros();
  long lastPrint = lastTimestamp + 1000;
  int currentAlarmStart = micros();
  const int currentAlarmDelay = CURRENT_ALARM_DELAY;
  motors[LEADER].bottomCurrentLimit = minCurrent = CURRENT_LIMIT;
  motors[FOLLOWER].bottomCurrentLimit = minCurrent = CURRENT_LIMIT;
  bool followerBottomHit = false;
  bool leaderBottomHit = false;
  pid_on = false;
  currentAlarmSet = false;
  ALL_MOTORS(motors[motor].speed = MIN_MOTOR_TRAVEL_SPEED;)
  homing = true;
  ALL_MOTORS(motors[motor].homing = true;)

  // Loop until both leader and follower motors are out of range
  for (;;) {
    currentAlarmSet = false;
    const long timestamp = micros();
    const int currentDeltaTime = timestamp - currentAlarmStart;
```

```
      // Check if it's time to update the motor current limits
      if (currentDeltaTime > currentAlarmDelay) {
        motors[LEADER].bottomCurrentLimit = 300;
        motors[FOLLOWER].bottomCurrentLimit = 300;
      }

      if (!leaderBottomHit) {
        leaderBottomHit = motors[LEADER].isStopped();
      }

      if (!followerBottomHit) {
        followerBottomHit = motors[FOLLOWER].isStopped();
      }

      Serial.printf("Leader bottom hit: %d\n", leaderBottomHit);
      Serial.printf("Follower bottom hit: %d\n", followerBottomHit);
      //  Check if both leader and follower motors are out of range
      if (leaderBottomHit && followerBottomHit) {
        Serial.println("Bottom hit.");
        pid_on = true;

        // Zero the leader and follower motors
        ALL_MOTORS_COMMAND(zero)

        // Disable all motors, reset soft movement, and set the direction to
        // STOP
        ALL_MOTORS_COMMAND(disable)
        resetSoftMovement();
        resetCurrentInformation();
        systemDirection = requestedDirection = Direction::STOP;

        // Calculate the time delta and update the motor status
        const float deltaT = ((float)(timestamp - lastTimestamp) / 1.0e6);
        update(deltaT);

        // Exit the loop
        break;
      }

      // Calculate the time delta and update the motor status
      const float deltaT = ((float)(timestamp - lastTimestamp) / 1.0e6);
      lastTimestamp = timestamp;
      update(deltaT);

      if (timestamp - lastPrint > 250000) {
        report();
      }
    }

    Serial.printf("Backing up %d pulses\n", ALARM_REVERSE_AMOUNT);
    lastTimestamp = micros();
    // Extend slightly
    pid_on = true; // Enable PID control
    resetSoftMovement();
    ; // Reset soft movement
    resetCurrentInformation();
    speed = MIN_MOTOR_TRAVEL_SPEED;
    targetSpeed = -1;
    ALL_MOTORS(motors[motor].outOfRange =
                   false;) // Reset out of range flag for all motors
    ALL_MOTORS(
        motors[motor].speed =
            MIN_MOTOR_TRAVEL_SPEED;) // Reset out of range flag for all motors
    ALL_MOTORS_COMMAND(extend);       // Send extend command to all motors
    systemDirection = Direction::EXTEND;     // Set system direction to extend
    requestedDirection = Direction::EXTEND; // Set requested direction to extend

    for (;;) {
      const long timestamp = micros();

      if (motors[LEADER].pos >= ALARM_REVERSE_AMOUNT ||
          motors[FOLLOWER].pos >= FOLLOWER_ALARM_REVERSE_AMOUNT) {
        // Zero the leader and follower motors
        ALL_MOTORS_COMMAND(zero)
        ALL_MOTORS(motors[motor].homing = false;)
        homing = false;

        // Disable all motors, reset soft movement, and set the direction to
        // STOP
        Serial.println("Stopping");
        resetSoftMovement();
        ;

        // Set the speed to 0
        speed = 0;
        targetSpeed = -1;
        ALL_MOTORS_COMMAND(disable)
```

```
        systemDirection = requestedDirection = Direction::STOP;

        // Calculate the time delta and update the motor status
        const float deltaT = ((float)(timestamp - lastTimestamp) / 1.0e6);
        pidController.reset(); // Reset the time parameters for PID
        update(deltaT);
        resetCurrentInformation();
        break;
      }
      // Calculate the time delta and update the motor status
      const float deltaT = ((float)(timestamp - lastTimestamp) / 1.0e6);
      update(deltaT);
    }
  }

  void setSpeed(const int newSpeed,
                const int softMovementTime = SOFT_MOVEMENT_TIME_MS) {

    targetSpeed = newSpeed;

    // Update the target speed
    Serial.printf("SetSpeed(%d)\n", newSpeed);
    Serial.printf("Speed: %d\n", speed);
    Serial.printf("Target speed: %d\n", targetSpeed);

    // Reset the soft start and last PWM update times
    softStart = lastPWMUpdate = micros();
    const int softMovementTimeMicros = softMovementTime * MICROS_IN_MS;
    softMovingTime = softMovementTimeMicros;
    const int softMovementUpdateSteps =
        (softMovementTimeMicros / SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS);
    // Calculate the amount to update the PWM duty cycle per step
    pwmUpdateAmount =
        ceil((float)abs(targetSpeed - speed) / softMovementUpdateSteps);

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

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

    if (newSpeed < speed) {
      // Reducing speed - use intermediate setpoint
      intermediateSpeed = speed - (speed - newSpeed) / 3;
    }

    if (requestedDirection != Direction::RETRACT) {
      pidController.setParams(RETRACT_RAMP_KP);
    } else if (requestedDirection == Direction::EXTEND) {
      pidController.setParams(EXTEND_RAMP_KP);
    }
  }

  void zero() { ALL_MOTORS_COMMAND(zero) }

  void report() {
    ALL_MOTORS_COMMAND(readPos)
    Serial.printf("MotorController\n--------------------\nSpeed: %d\nTarget "
                  "Speed: %d\n\n",
                  speed, targetSpeed);
    pidController.report();
    Serial.printf("\n\n\n");
    displayCurrents();
    Serial.printf("Current Alarm Status: %s\n\n",
                  currentAlarmSet ? "true" : "false");
    ALL_MOTORS_COMMAND(displayInfo)
  }

  void printCurrent() {
    // Check if the motors are stopped
    if (!motorsStopped()) {
      // Print the current values
      Serial.printf("Leader Current: %d\n", leaderCurrent);
      Serial.printf("Follower Current: %d\n", followerCurrent);
    }
  }
```

```
  void savePosition(const int slot, const int position_value = -1) {
    // Make sure the slot index is within valid range and if no position was
    // manually specified, then use the current position of the leader motor
    const int positionToSave =
        position_value > -1 ? position_value : motors[LEADER].pos;

    if (slot > 0 && slot < NUM_POSITION_SLOTS) {
      // Store the position value in the savedPositions array.
      savedPositions[slot - 1] = positionToSave;

      // Store the position value in the positionStorage.
      positionStorage.putInt(save_position_slot_names[slot], positionToSave);
    }
  }

  void setPos(const int newPos) {
    desiredPos = constrain(newPos, 0, motors[LEADER].totalPulseCount);

    // Check if the current position is less than the desired position
    if (motors[LEADER].pos < desiredPos) {
      extend();
    }
    // Check if the current position is greater than the desired position
    else if (motors[LEADER].pos > desiredPos) {
      retract();
    }
  }

  void updateCurrentReadings(const int elapsedTime) {
    // Store the last readings of the leader and follower currents
    const double lastLeaderCurrent = leaderCurrent;
    const double lastFollowerCurrent = followerCurrent;

    const int lCurrent = motors[LEADER].getCurrent();
    const int fCurrent = motors[FOLLOWER].getCurrent();

    // Get the filtered current readings of the leader and follower motors
    leaderCurrent = leaderCurrentFilter(lCurrent);
    followerCurrent = followerCurrentFilter(fCurrent);

    // Calculate the time factor
    const double timeFactor = 1000000.0 / elapsedTime;

    // Calculate the velocities of the leader and follower currents
    leaderCurrentVelocity = static_cast<int>(
        (leaderCurrent - lastLeaderCurrent) * timeFactor + 0.5);
    followerCurrentVelocity = static_cast<int>(
        (followerCurrent - lastFollowerCurrent) * timeFactor + 0.5);
  }

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

  bool motorsStopped() const {
    // Check if the leader motor is stopped
    bool isLeaderStopped = motors[LEADER].isStopped();

    // Check if the follower motor is stopped
    bool isFollowerStopped = motors[FOLLOWER].isStopped();

    // Return true if both motors are stopped
    return isLeaderStopped && isFollowerStopped;
  }

  bool currentAlarmTriggered() {
    return currentAlarmSet &&
               (leaderCurrent > motors[LEADER].currentAlarmLimit) ||
           (followerCurrent > motors[FOLLOWER].currentAlarmLimit);
  }

  bool motorsDesynced(void) const {
    return abs(motors[LEADER].pos - motors[FOLLOWER].pos) > DESYNC_TOLERANCE;
  }

  void handlePid() {
    updateLeadingAndLaggingIndicies();
    // Speed to set the faster motor to as calculated by the PID algorithm
    const int adjustedSpeed = pidController.adjustSpeed(
        motors[leadingIndex], motors[laggingIndex], speed, deltaT);

    motors[leadingIndex].speed = adjustedSpeed;
    motors[laggingIndex].speed = speed;
  }

  bool motorsCloseToEndOfRange() {
    // Read the current position of all motors
    ALL_MOTORS_COMMAND(readPos)

```

```
  // Get the normalized position of the leader motor
  double leaderPos = motors[LEADER].getNormalizedPos();

  // Get the normalized position of the follower motor
  double followerPos = motors[FOLLOWER].getNormalizedPos();

  // Check if the leader motor is close to the end of its range
  // Check if the leader motor is close to the end of its range
  bool leaderCloseToEnd =
      (leaderPos < 0.1 && motors[LEADER].dir == Direction::RETRACT) ||
      (leaderPos > 0.9 && motors[LEADER].dir == Direction::EXTEND);

  // Check if the follower motor is close to the end of its range
  bool followerCloseToEnd =
      (followerPos < 0.15 && motors[FOLLOWER].dir == Direction::RETRACT) ||
      (followerPos > 0.85 && motors[FOLLOWER].dir == Direction::EXTEND);

  // Return true if either motor is close to the end of its range
  return leaderCloseToEnd && followerCloseToEnd;
}

void handleCurrentAlarm() {
  // Print debug message if debugEnabled is true
  displayCurrents();
  Serial.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
  Serial.println("!!!!!!!!!!!!!!!!!!!!!!!!ALARM!!!!!!!!!!!!!!!!!!");
  Serial.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");

  /*
  immediateHalt();
  systemDirection = requestedDirection = Direction::STOP;
  resetSoftMovement();
  resetCurrentInformation();

  */
  report();
}

void updateLeadingAndLaggingIndicies() {
  // Check if the system direction is to extend
  if (Direction::EXTEND == systemDirection) {
    // Update the lagging index based on the normalized positions of the
    // motors
    laggingIndex = (motors[LEADER].getNormalizedPos() <
                    motors[FOLLOWER].getNormalizedPos())
                       ? MotorRoles::LEADER
                       : MotorRoles::FOLLOWER;
    // Update the leading index based on the normalized positions of the
    // motors
    leadingIndex = (motors[LEADER].getNormalizedPos() >=
                    motors[FOLLOWER].getNormalizedPos())
                       ? MotorRoles::LEADER
                       : MotorRoles::FOLLOWER;
  }
  // Check if the system direction is to retract
  else if (Direction::RETRACT == systemDirection) {
    // Update the lagging index based on the normalized positions of the
    // motors
    laggingIndex = (motors[LEADER].getNormalizedPos() >
                    motors[FOLLOWER].getNormalizedPos())
                       ? MotorRoles::LEADER
                       : MotorRoles::FOLLOWER;
    // Update the leading index based on the normalized positions of the
    // motors
    leadingIndex = (motors[LEADER].getNormalizedPos() <=
                    motors[FOLLOWER].getNormalizedPos())
                       ? MotorRoles::LEADER
                       : MotorRoles::FOLLOWER;
  }
}

void displayCurrents() {
  Serial.printf("Leader Motor Current: %d\n", leaderCurrent);
  // Serial.printf("Leader current velocity: %d\n", leaderCurrentVelocity);
  Serial.printf("Follower Motor Current: %d\n", followerCurrent);
  // Serial.printf("Follower current velocity: %d\n",
  // followerCurrentVelocity);
}

void sampleCurrents() {
  const int largerCurrent = std::max(leaderCurrent, followerCurrent);
  if (largerCurrent > maxCurrent) {
    maxCurrent = largerCurrent;
  }

  const int currentDifference = abs(leaderCurrent - followerCurrent);
  currentDifferenceSum += currentDifference;
```

```
    currentSum += leaderCurrent;
    samples++;
  }

  void setCurrentLimit() {
    leaderWorkingCurrent = leaderCurrent;

    const int averageCurrent = currentSum / samples;

    const int currentSetVal = static_cast<int>(
        (((averageCurrent + ((leaderCurrent + followerCurrent) / 2)) / 2)) *
        CURRENT_INCREASE_MULTIPLIER);

    currentOffset = currentDifferenceSum / samples;

    Serial.printf("Current alarms set to: Leader => %d, Follower => %d\n",
                  currentSetVal, currentSetVal - currentOffset);

    motors[LEADER].currentAlarmLimit = currentSetVal;
    motors[FOLLOWER].currentAlarmLimit = currentSetVal - currentOffset;

    currentAlarmSet = true;
  }

  void updateSoftMovement() {
    const long currentTime = micros();

    if (targetSpeed >= 0) {
      // Distance from the current speed to the target speed
      const int speedDelta = abs(speed - targetSpeed);

      // The time since soft movement started
      const long moveTimeDelta = currentTime - softStart;

      // Calculate the time since the last PWM update
      const long updateTimeDelta = currentTime - lastPWMUpdate;

      if (updateTimeDelta >= SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS) {
        const bool timeToUpdate = moveTimeDelta < softMovingTime;
        // Get the true PWM update amount based on the actual elapsed time
        const float updateAmount =
            pwmUpdateAmount * (static_cast<float>(updateTimeDelta) /
                               SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS);
        const bool speedDeltaEnough = speedDelta >= abs(updateAmount);

        if (timeToUpdate && speedDeltaEnough) {

          const float newSpeed = static_cast<float>(speed + updateAmount);
          speed = static_cast<int>(floorf(newSpeed));
          // Serial.printf("Speed <== %d\n", speed);
          lastPWMUpdate = micros();
        } else {
          // Set the speed to the target speed and reset the soft movement if
          // time expired or there is less than one full update step until we
          // reach the target speed.
          speed = targetSpeed;
          resetSoftMovement();

          if (requestedDirection == Direction::STOP) {
            systemDirection = Direction::STOP;

            immediateHalt();
            stopping = false;
            report();
          }
        }
      }
      if (intermediateSpeed >= 0) {
        // Transition to intermediate speed
        double output = control(intermediateSpeed,
                                speed); // Assuming speed is the current speed
        if (abs(output - intermediateSpeed) < 1) { // Or a small threshold
          intermediateSpeed = -1;
        }
      } else {
        // Regular target speed tracking
        control(targetSpeed, speed);
      }
    }
  }

  void update(const float deltaT = 0.0f) {
    // Get current time in microseconds
    const long currentTime = micros();
    // Calculate the time since the last current update
    const long currentUpdateDelta = currentTime - lastCurrentUpdate;

```

```
  const int currentMs = currentTime / MICROS_IN_MS;
  const int moveStartMs = currentTime / MICROS_IN_MS;
  this->deltaT = deltaT;

  if (softStartQueued && (currentTime - moveStart) > 100000) {
    if (!stopping) {
      setSpeed(DEFAULT_MOTOR_SPEED);
    } else {
      setSpeed(0, SOFT_STOP_TIME_MS);
    }
    softStartQueued = false;
  }

  if (motorsStopped()) {
    resetCurrentInformation();
    resetSoftMovement();
    systemDirection = requestedDirection = Direction::STOP;
  }

  // Immediate halt on motor desynchronization
  /*
  if (motorsDesynced()) {
    immediateHalt();
  }
  */

  // Check if the system is stopped. If so, ensure current is disabled to the
  // motors.
  if (Direction::STOP == systemDirection || motorsStopped()) {
    ALL_MOTORS_COMMAND(disable)
    ALL_MOTORS_COMMAND(update)
    return;
  }

  // If the motors are close to the end of the range of movement and are not
  // in a soft-stop, and aren't ramping up/down, then set the speed to end of
  // range speed
  if (motorsCloseToEndOfRange() && speed != MOTOR_END_OF_RANGE_SPEED &&
      targetSpeed < 0) {
    if (!stopping) {
      setSpeed(MOTOR_END_OF_RANGE_SPEED);
    } else {
      setSpeed(0, MIN_MOTOR_TRAVEL_SPEED);
    }
  }

  // Check if it's time to update the current readings
  if (currentUpdateDelta >= currentUpdateInterval) {
    // Check if the motors are not stopped
    if (!motorsStopped()) {
      updateCurrentReadings(currentUpdateDelta);
      // Update the last current update time
      lastCurrentUpdate = currentTime;

      // Display the current readings if debug is enabled
      // displayCurrents();

      if (!currentAlarmSet) {
        sampleCurrents();
      }

      if ((currentTime - moveStart) > CURRENT_ALARM_DELAY) {
        if (!currentAlarmSet) {
          setCurrentLimit();
        }

        if (currentAlarmTriggered()) {
          handleCurrentAlarm();
        }
      }
    }
  }

  if (desiredPos != -1) {
    if (abs(desiredPos - motors[LEADER].pos) < SET_POSITION_BUFFER) {
      if (debugEnabled) {
        Serial.printf("Desired Pos: %d - REACHED\n", desiredPos);
      }
      desiredPos = -1;
      stop();
    }
  }

  // Check if the soft movement system has a target speed
  if (targetSpeed >= 0) {
    updateSoftMovement();
  }
```

```
    if (pid_on) {
      handlePid();
    } else {
      motors[leadingIndex].speed = speed;
      motors[laggingIndex].speed = speed;
    }

    ALL_MOTORS_COMMAND(update)
  }

  void resetSoftMovement() {
    pwmUpdateAmount = 0;
    lastPWMUpdate = -1;
    softStart = -1;
    targetSpeed = -1;
    currentUpdateInterval = CURRENT_UPDATE_INTERVAL;
    pidController.setParams(DEFAULT_KP);
    softStartQueued = false;
  }

  void resetCurrentInformation() {
    currentAlarmSet = false;
    maxCurrent = -1;
    samples = 0;
    currentDifferenceSum = 0;
    currentSum = 0;
    currentOffset = 0;
    resetMotorCurrentAlarms();
    leaderCurrentFilter.reset();
    followerCurrentFilter.reset();
  }

  void disableMotors() {
    ALL_MOTORS_COMMAND(disable)
    ALL_MOTORS_COMMAND(update)
  }

  void updateMotors() { ALL_MOTORS_COMMAND(update) }

  void resetPid() {
    // Set parameters for PID controller to defaults
    pidController.setParams(DEFAULT_KP, MAX_SPEED);
  }

  void resetMotorCurrentAlarms() {
    motors[LEADER].currentAlarmLimit = CURRENT_LIMIT;
    motors[FOLLOWER].currentAlarmLimit = CURRENT_LIMIT;
  }
};

#endif // _MOTOR_CONTROLLER_HPP_
```

5.18 MotorPins.hpp File Reference

```
#include <cstdint>
```

Enumerations

- enum class MotorPin : std::uint8_t {
  UNASSIGNED = 0 , MOTOR1_RPWM_PIN = 25 , MOTOR1_LPWM_PIN = 19 , MOTOR1_R_EN_PIN = 26 , MOTOR1_L_EN_PIN = 18 , MOTOR1_HALL1_PIN = 22 , MOTOR1_HALL2_PIN = 23 , 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.

```

#ifndef _MOTOR_PINS_HPP_
#define _MOTOR_PINS_HPP_

#include <cstdint>

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

  MOTOR1_RPWM_PIN = 25,

  MOTOR1_LPWM_PIN = 19,

  MOTOR1_R_EN_PIN = 26,

  MOTOR1_L_EN_PIN = 18,

  MOTOR1_HALL1_PIN = 22,

  MOTOR1_HALL2_PIN = 23,

  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
};

#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.

```

#ifndef _MOTORS_SOFT_MOVEMENT_STATE_HPP_
#define _MOTORS_SOFT_MOVEMENT_STATE_HPP_

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

class MotorsSoftMovementState : public ControllerState {
public:
  MotorsSoftMovementState();
  MotorsSoftMovementState(MotorController *pMotorController);

  void enter() {
    Serial.print("Motors soft movement started");

    if (nullptr != controller) {
      // Reset current information
      controller->resetCurrentInformation();

      // Reset soft movement
      controller->resetSoftMovement();

      // Reset PID controller
      controller->resetPid();

      // Set travel speed to the minimum
      controller->targetSpeed = MIN_MOTOR_TRAVEL_SPEED;

      controller->requestedDirection = Direction::STOP;

      // Save the start time of the motors
      controller->moveStart = micros();
    } else {
      Serial.println("MotorsSoftMovementState - No controller");
    }
  }

  void update() {
    if (nullptr != controller) {
      controller->updateSoftMovement();

      if (pid_on) {
        controller->handlePid();
      }

      controller->updateMotors();
    } else {
      Serial.println("MotorsSoftMovementState - No controller");
    }
  }

  void leave() { Serial.print("Motors soft movement stopped"); }
}; // end class MotorsSoftMovementState

#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.

```cpp

#ifndef _MOTORS_STARTING_STATE_HPP_
#define _MOTORS_STARTING_STATE_HPP_

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

class MotorsStartingState : public ControllerState {
public:
  MotorsStartingState();
  MotorsStartingState(MotorController *pMotorController);

  void enter() {
    Serial.print("Motors starting");

    if (nullptr != controller) {
      // Reset current information
      controller->resetCurrentInformation();

      // Reset soft movement
      controller->resetSoftMovement();

      // Reset PID controller
      controller->resetPid();

      // Set travel speed to the minimum
      controller->speed = MIN_MOTOR_TRAVEL_SPEED;

      // Save the start time of the motors
      controller->moveStart = micros();
    } else {
      Serial.println("MotorsStartingState - No controller");
    }
  }

  void update() {
    if (pid_on) {
      controller->handlePid();
    }

    controller->sampleCurrents();
  }

  void leave() { Serial.print("Motors moving to soft movement state"); }

}; // end class MotorsStartingState

#endif // _MOTORS_STARTING_STATE_HPP_
```

5.24 MotorsStoppedState.hpp File Reference

```cpp
#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.

```

#ifndef _MOTORS_STOPPED_STATE_HPP_
#define _MOTORS_STOPPED_STATE_HPP_

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

class MotorsStoppedState : public ControllerState {
public:
  MotorsStoppedState();

  MotorsStoppedState(MotorController *pMotorController);

  void enter() {
    Serial.print("Motors Stopped");

    if (nullptr != controller) {
      // Reset current information
      controller->resetCurrentInformation();

      // Reset soft movement
      controller->resetSoftMovement();

      // Set directions to stop
      controller->requestedDirection = Direction::STOP;
      controller->systemDirection = Direction::STOP;

      // Disable motors
      controller->disableMotors();
    } else {
      Serial.println("Motors Stopped State - No controller");
    }
  }

  void update() { controller->updateMotors(); }

  void leave() { Serial.print("Motors starting"); }
}; // end class State

#endif // _MOTORS_STOPPED_STATE_HPP_
```

5.26 PIDController.hpp

```

#ifndef _PID_CONTROLLER_HPP__
#define _PID_CONTROLLER_HPP__

#include "defs.hpp"
#include <cmath>
#include <cstring>
#include <limits>
#include <stdio.h>

class PIDController {
private:
  float errorIntegral = 0.0f;
  float previousMeasurement = 0.0f;
  float positionDifference = 0.0f;
  float maxSpeedAdjustmentRate = 900.0f;
  float filteredDerivative = 0.0f;
  const float alpha = 0.5; // Filter constant
  int K_p;
  float K_i = DEFAULT_KI;
  float K_d = DEFAULT_KD;
  float tau = 1.0f;
  int uMax;
  float derivative = 0.0f;
  float limMinInteg, limMaxInteg;
  float lastError = 0.0f;
  float followerMaxAccel =
      5.0f;
  int followerMaxSpeed = MAX_SPEED;
  float filteredSpeed = 0.0f;

  const float MAX_ACCELERATION_INCREASE = 0.01f;
  const float MAX_ACCELERATION_LIMIT = 5.0f;
  const float SPEED_ALPHA = 0.5f;      // The filtering constant for speed.
```

```
  const float SETPOINT_WEIGHT = 1.15f; // or 0.8f as per your requirement
  const int POSITION_DELTA_SPEED_SCALER_EXTEND = 70000;
  const int POSITION_DELTA_SPEED_SCALER_RETRACT = 70000;

  const int MAX_POSITION_SCALE_VALUE = 50;

public:
  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)
      : K_p(kp), K_i(ki), K_d(kd), tau(tau), uMax(uMax),
        followerMaxSpeed(0.9 * uMax), // Initialize followerMaxSpeed
        limMinInteg(
            -std::numeric_limits<float>::infinity()), // Or another appropriate
                                                      // value
        limMaxInteg(
            std::numeric_limits<float>::infinity()), // Or another appropriate
                                                     // value
        errorIntegral(limMinInteg), // Now it is safe to use limMinInteg
        filteredSpeed(0.0f) {

    if (debugEnabled) {
      Serial.println("PID Controller Initialized");
      Serial.printf("PID Parameters:\n");
      Serial.printf("K_p: %d\n", K_p);
      Serial.printf("K_i: %d\n", K_i);
      Serial.printf("K_d: %d\n", K_d);
      Serial.printf("Last Error: %f\n", lastError);
      Serial.printf("Error integral: %f\n", errorIntegral);
      Serial.printf("Max Speed: %d\n", uMax);
      Serial.println("---------------------------");
    }
  }

  void setParams(const int kpIn, const float kiIn = DEFAULT_KI,
                 const float kdIn = DEFAULT_KD, const int uMaxIn = MAX_SPEED) {
    K_p = kpIn;
    K_i = kiIn;
    K_d = kdIn;
    uMax = uMaxIn;

    if (fabs(kiIn) < 1e-6) { // Checks if Ki is almost zero
      K_i = 0.0f;
      limMinInteg =
          -std::numeric_limits<float>::infinity(); // Changed to negative
                                                   // infinity
      limMaxInteg = std::numeric_limits<float>::infinity(); // Changed to
                                                            // positive infinity
    } else {
      K_i = kiIn;
      limMinInteg = -uMax / K_i; // Recalculated the minimum limit
      limMaxInteg = uMax / K_i;
    }
    errorIntegral = constrain(errorIntegral, limMinInteg, limMaxInteg);
  }

  int adjustSpeed(Motor &leader, Motor &follower, const int speed,
                  const float deltaT = 0.0f) {

    const float leadingMotorPosition = leader.getNormalizedPos();
    const float laggingMotorPosition = follower.getNormalizedPos();
    positionDifference = (laggingMotorPosition - leadingMotorPosition);
    const float error = SETPOINT_WEIGHT * positionDifference;
    const float derivativeTerm = K_d * filteredDerivative;
    const float integralTerm = K_i * errorIntegral;
    maxSpeedAdjustmentRate +=
        MAX_ACCELERATION_INCREASE * 1000; // scaling up the increment
    maxSpeedAdjustmentRate = std::min(
        maxSpeedAdjustmentRate, followerMaxAccel * 1000); // scale up the limit

    // Update filtered speed before it's used in PID calculations
    filteredSpeed = SPEED_ALPHA * filteredSpeed + (1 - SPEED_ALPHA) * speed;

    // Calculate the derivative and then use it to calculate the filtered
    // derivative
    if (deltaT > 0.0f) { // Avoid division by zero
      derivative = (error - lastError) / deltaT;
      filteredDerivative =
          alpha * filteredDerivative + (1 - alpha) * derivative;
    } else {
      derivative = 0.0f;
    }

    const int proportionalTerm = static_cast<int>(round(error * K_p * 3));

    errorIntegral += error * deltaT;
    errorIntegral =
```

```
        constrain(errorIntegral, limMinInteg, limMaxInteg); // Anti-Windup

    const int scalarValue = leader.dir == Direction::EXTEND
                                ? POSITION_DELTA_SPEED_SCALER_EXTEND
                                : POSITION_DELTA_SPEED_SCALER_RETRACT;

    const int directionSign = leader.dir == Direction::EXTEND ? 1 : -1;
    const int positionDeltaBoost =
        constrain(directionSign * positionDifference * scalarValue,
                  -MAX_POSITION_SCALE_VALUE, MAX_POSITION_SCALE_VALUE);

    // Serial.printf("Position Delta Boost: %d\n", positionDeltaBoost);

    const int u = speed - (proportionalTerm + integralTerm + derivativeTerm);

    const float maxDeltaSpeed = maxSpeedAdjustmentRate * deltaT *
                                0.001; // scaling down the adjustment rate
    int adjustedDeltaSpeed = constrain(u, -maxDeltaSpeed, maxDeltaSpeed);

    int adjustedSpeed = speed + adjustedDeltaSpeed + positionDeltaBoost;

    // Serial.printf("Adjusted Speed: %d\n", adjustedSpeed);

    if (debugEnabled) {
      // Debug prints
    }

    lastError = error;
    previousMeasurement = laggingMotorPosition;
    return constrain(adjustedSpeed, MIN_SPEED, MAX_SPEED);
  }

  void setFollowerMaxAccel(float newMaxAccel) {
    followerMaxAccel = newMaxAccel;
  }

  void setFollowerMaxSpeed(int newMaxSpeed) { followerMaxSpeed = newMaxSpeed; }

  void report() const {
    Serial.printf("\nPID Parameters:\n");
    Serial.println("----------------------------");
    Serial.printf("K_p: %d\n", K_p);
    Serial.printf("K_i: %f\n", K_i);
    Serial.printf("K_d: %f\n", K_d);
    Serial.printf("Position Difference: %f\n", positionDifference);
    Serial.println("----------------------------");
  }

  void setKd(float newKd) { K_d = newKd; }
  void setKi(float newKi) {
    if (fabs(newKi) < 1e-6) { // Checks if newKi is almost zero
      K_i = 0.0f;
      limMinInteg =
          -std::numeric_limits<float>::infinity(); // Changed to negative
                                                   // infinity
      limMaxInteg = std::numeric_limits<float>::infinity(); // Changed to
                                                            // positive infinity
    } else {
      K_i = newKi;
      limMinInteg = -uMax / K_i; // Recalculated the minimum limit
      limMaxInteg = uMax / K_i;
    }
    errorIntegral = constrain(errorIntegral, limMinInteg, limMaxInteg);
  }

  void reset() {
    lastError = 0.0f;
    errorIntegral = 0.0f;
  }
};

#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(analog←Read(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

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

Returns

Mapped output value for input value

5.28 PinMacros.hpp

Go to the documentation of this file.

```
00001
00002 #ifndef _PIN_MACROS_HPP_
```

```
#define _PIN_MACROS_HPP_

#include "defs.hpp"

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

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

#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.

```

#ifndef _POTENTIOMTER_PINS_HPP_
#define _POTENTIOMTER_PINS_HPP_
```

```

#include <cstdint>

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

  KP_POT_PIN = 32,
};

#endif // _POTENTIOMTER_PINS_HPP_
```

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.

```

#ifndef _ROUTE_MACROS_HPP__
#define _ROUTE_MACROS_HPP__

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

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

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

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

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

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

#endif // _ROUTE_MACROS_HPP__
```

5.33 StateController.hpp

```
#ifndef _STATE_CONTROLLER_
#define _STATE_CONTROLLER_

#include "MotorsSoftMovementState.hpp"
#include "MotorsStartingState.hpp"
#include "MotorsStoppedState.hpp"
#include "defs.hpp"

enum MotorControllerState {
  MOTORS_STARTING_STATE,
  MOTORS_SOFT_MOVEMENT_STATE,
  MOTORS_STOPPING_STATE,
  MOTORS_STOPPED_STATE
};
constexpr int NUMBER_OF_MOTOR_STATES = 4;

class StateController {
  MotorsStartingState motorsStartingState;
  MotorsSoftMovementState motorsSoftMovementState;
  MotorsStoppedState motorsStoppedState;

  ControllerState *motorsStateMap[NUMBER_OF_MOTOR_STATES];
```

```

  ControllerState *currentState = nullptr;

  void initializeStateMap() {
    motorsStateMap[MOTORS_STARTING_STATE] = &motorsStartingState;
    motorsStateMap[MOTORS_SOFT_MOVEMENT_STATE] = &motorsSoftMovementState;
    motorsStateMap[MOTORS_STOPPED_STATE] = &motorsStoppedState;
  }

public:
  StateController();
  StateController(MotorController *pMotorController) {
    setController(pMotorController);
  }

  void setController(MotorController *pMotorController) {
    motorsStartingState.setController(pMotorController);
    motorsSoftMovementState.setController(pMotorController);
    motorsStoppedState.setController(pMotorController);
  }

  void setState(MotorControllerState newState) {
    if (currentState != nullptr) {
      currentState->leave();
    }

    currentState = motorsStateMap[newState];
    currentState->enter();
  }

  void update() {
    if (currentState != nullptr) {
      currentState->update();
    }
  }
}; // end class StateController

#endif // _STATE_CONTROLLER_
```

Index