. Documentation

Release 2021.0.2

Author

Jan 29, 2021

ROBOT PROGRAMMING

This project wraps the REV 3rd party libraries and makes them usable for Python teams.

Note: The RobotPy project is not associated with or endorsed by REV Robotics

CHAPTER

ONE

REV LIBRARIES

These are not installed on the Robot by default. For installation instructions, see robotpy-rev install docs.

Warning: REV does not support all platforms, only the following are available:

- Windows x64
- Linux
- RoboRIO

This documentation documents the various classes and methods that are available to Python code, but don't discuss in detail how to actually set up, configure, and tune your REV hardware. For that kind of information, refer to the REV software site.

1.1 rev Package

rev.CANAnalog(self, device, mode)	
rev.CANDigitalInput(self, device, ...)	
rev.CANEncoder(*args, **kwargs)	Overloaded function.
rev.CANError(self, value)	Members:
rev.CANPIDController	
rev.CANSensor	
rev.CANSparkMax(self, deviceID, type)	Create a new SPARK MAX Controller
rev.CANSparkMaxLowLevel(self, deviceID, type)	Create a new SPARK MAX Controller
rev.ControlType(self, value)	Members:
rev.IdleMode(self, value)	Members:
rev.LimitSwitch(self, value)	Members:
rev.LimitSwitchPolarity(self, value)	Members:
rev.MotorType(self, value)	Members:
rev.SparkMax(self, channel)	REV Robotics CAN speed controller controlled via PWM.

1.1.1 CANAnalog

class rev.**CANAnalog**(*self:* rev.CANAnalog, *device:* rev.CANSparkMax, *mode:* *rev._rev.CANAnalog.AnalogMode*) → None

Bases: *rev.CANSensor*

class AnalogMode(*self: rev._rev.CANAnalog.AnalogMode*, *value: int*) → None

Bases: pybind11_builtins.pybind11_object

Analog sensors have the ability to either be absolute or relative. By default, GetAnalog() will return an absolute analog sensor, but it can also be configured to be a relative sensor instead.

Members:

kAbsolute

kRelative

kAbsolute = <AnalogMode.kAbsolute: 0>

kRelative = <AnalogMode.kRelative: 1>

property name

getAverageDepth(*self:* rev.CANAnalog) → int

Get the averafe sampling depth for a quadrature encoder.

Returns The average sampling depth

getInverted(*self:* rev.CANAnalog) → bool

Get the phase of the CANSensor. This will just return false if the user tries to get inverted while the SparkMax is Brushless and using the hall effect sensor.

Returns The phase of the encoder

getMeasurementPeriod(*self:* rev.CANAnalog) → int

Get the number of samples for reading from a quadrature encoder. This value sets the number of samples in the average for velocity readings.

Returns Measurement period in microseconds

getPosition(*self:* rev.CANAnalog) → float

Get the position of the motor. Returns value in the native unit of 'volt' by default, and can be changed by a scale factor using setPositionConversionFactor().

Returns Position of the sensor in volts

getPositionConversionFactor(*self:* rev.CANAnalog) → float

Get the current conversion factor for the position of the analog sensor.

Returns Analog position conversion factor

getVelocity(*self:* rev.CANAnalog) → float

Get the velocity of the motor. Returns value in the native units of 'volts per second' by default, and can be changed by a scale factor using setVelocityConversionFactor().

Returns Velocity of the sensor in volts per second

getVelocityConversionFactor(*self:* rev.CANAnalog) → float

Get the current conversion factor for the velocity of the analog sensor.

Returns Analog velocity conversion factor

getVoltage(*self:* rev.CANAnalog) → float

Get the voltage of the analog sensor.

Returns Voltage of the sensor

setAverageDepth(*self:* rev.CANAnalog, *depth: int*) → *rev.CANError*
Set the average sampling depth for a quadrature encoder. This value sets the number of samples in the average for velocity readings. This can be any value from 1 to 64.

When the SparkMax controller is in Brushless mode, this will not change any behavior.

Parameters **depth** – The average sampling depth between 1 and 64 (default)

Returns CANError.kOK if successful

setInverted(*self:* rev.CANAnalog, *inverted: bool*) → *rev.CANError*
Set the phase of the CANSensor so that it is set to be in phase with the motor itself. This only works for quadrature encoders. This will throw an error if the user tries to set inverted while the SparkMax is Brushless and using the hall effect sensor.

Parameters **inverted** – The phase of the encoder

Returns CANError.kOK if successful

setMeasurementPeriod(*self:* rev.CANAnalog, *period_ms: int*) → *rev.CANError*
Set the measurement period for velocity measurements of a quadrature encoder. When the SparkMax controller is in Brushless mode, this will not change any behavior.

The basic formula to calculate velocity is change in positon / change in time. This parameter sets the change in time for measurement.

Parameters **period_us** – Measurement period in milliseconds. This number may be between 1 and 100 (default).

Returns CANError.kOK if successful

setPositionConversionFactor(*self:* rev.CANAnalog, *factor: float*) → *rev.CANError*
Set the conversion factor for the position of the analog sensor. By default, revolutions per volt is 1. Changing the position conversion factor will also change the position units.

Parameters **factor** – The conversion factor which will be multiplied by volts

Returns CANError Set to CANError.kOK if successful

setVelocityConversionFactor(*self:* rev.CANAnalog, *factor: float*) → *rev.CANError*
Set the conversion factor for the veolocity of the analog sensor. By default, revolutions per volt second is 1. Changing the velocity conversion factor will also change the velocity units.

Parameters **factor** – The conversion factor which will be multipled by volts per second

Returns CANError Set to CANError.kOK is successful

1.1.2 CANDigitalInput

class rev.**CANDigitalInput**(*self:* rev.CANDigitalInput, *device:* rev.CANSparkMax, *limitSwitch:* *rev._rev.CANDigitalInput.LimitSwitch*, *polarity:* *rev._rev.CANDigitalInput.LimitSwitchPolarity*) → None
Bases: pybind11_builtins.pybind11_object

class LimitSwitch(*self: rev._rev.CANDigitalInput.LimitSwitch*, *value: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kForward

kReverse

kForward = <LimitSwitch.kForward: 0>

kReverse = <LimitSwitch.kReverse: 1>

property name

class LimitSwitchPolarity(*self: rev._rev.CANDigitalInput.LimitSwitchPolarity*, *value: int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kNormallyOpen

kNormallyClosed

kNormallyClosed = <LimitSwitchPolarity.kNormallyClosed: 1>

kNormallyOpen = <LimitSwitchPolarity.kNormallyOpen: 0>

property name

enableLimitSwitch(*self:* rev.CANDigitalInput, *enable: bool*) → *rev.CANError*

Enables or disables controller shutdown based on limit switch.

get(*self:* rev.CANDigitalInput) → bool

Get the value from a digital input channel.

Retrieve the value of a single digital input channel from a motor controller. This method will return the state of the limit input based on the selected polarity, whether or not it is enabled.

isLimitSwitchEnabled(*self:* rev.CANDigitalInput) → bool

Returns true if limit switch is enabled.

1.1.3 CANEncoder

class rev.**CANEncoder**(**args*, ***kwargs*)

Bases: *rev.CANSensor*

Overloaded function.

1. __init__(self: rev._rev.CANEncoder, device: rev._rev.CANSparkMax, sensorType: rev._rev.CANEncoder.EncoderType = <EncoderType.kHallSensor: 1>, counts_per_rev: int = 42) -> None

2. __init__(self: rev._rev.CANEncoder, device: rev._rev.CANSparkMax, sensorType: rev._rev.CANEncoder.AlternateEncoderType, counts_per_rev: int) -> None

class AlternateEncoderType(*self: rev._rev.CANEncoder.AlternateEncoderType*, *value: int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kQuadrature

kQuadrature = <AlternateEncoderType.kQuadrature: 0>

property name

class EncoderType(*self: rev._rev.CANEncoder.EncoderType*, *value: int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kNoSensor

kHallSensor

kQuadrature

kSensorless

kHallSensor = <EncoderType.kHallSensor: 1>

kNoSensor = <EncoderType.kNoSensor: 0>

kQuadrature = <EncoderType.kQuadrature: 2>

kSensorless = <EncoderType.kSensorless: 3>

property name

getAverageDepth(*self:* rev.CANEncoder) → int
Get the averafe sampling depth for a quadrature encoder.

Returns The average sampling depth

getCPR(*self:* rev.CANEncoder) → int

getCountsPerRevolution(*self:* rev.CANEncoder) → int
Get the counts per revolution of the quadrature encoder.

For a description on the difference between CPR, PPR, etc. go to https://www.cuidevices.com/blog/what-is-encoder-ppr-cpr-and-lpr

Returns Counts per revolution

getInverted(*self:* rev.CANEncoder) → bool
Get the phase of the CANSensor. This will just return false if the user tries to get inverted while the SparkMax is Brushless and using the hall effect sensor.

Returns The phase of the encoder

getMeasurementPeriod(*self:* rev.CANEncoder) → int
Get the number of samples for reading from a quadrature encoder. This value sets the number of samples in the average for velocity readings.

Returns Measurement period in microseconds

getPosition(*self:* rev.CANEncoder) → float
Get the position of the motor. This returns the native units of 'rotations' by default, and can be changed by a scale factor using setPositionConversionFactor().

Returns Number of rotations of the motor

getPositionConversionFactor(*self:* rev.CANEncoder) → float
Get the conversion factor for position of the encoder. Multiplied by the native output units to give you position

Returns The conversion factor for position

getVelocity(*self:* rev.CANEncoder) → float
Get the velocity of the motor. This returns the native units of 'RPM' by default, and can be changed by a scale factor using setVelocityConversionFactor().

Returns Number the RPM of the motor

getVelocityConversionFactor(*self:* rev.CANEncoder) → float
Get the conversion factor for velocity of the encoder. Multiplied by the native output units to give you velocity

Returns The conversion factor for velocity

setAverageDepth(*self:* rev.CANEncoder, *depth: int*) → *rev.CANError*
Set the average sampling depth for a quadrature encoder. This value sets the number of samples in the average for velocity readings. This can be any value from 1 to 64.

When the SparkMax controller is in Brushless mode, this will not change any behavior.

Parameters **depth** – The average sampling depth between 1 and 64 (default)

Returns CANError.kOK if successful

setInverted(*self:* rev.CANEncoder, *inverted: bool*) → *rev.CANError*
Set the phase of the CANSensor so that it is set to be in phase with the motor itself. This only works for quadrature encoders. This will throw an error if the user tries to set inverted while the SparkMax is Brushless and using the hall effect sensor.

Parameters **inverted** – The phase of the encoder

Returns CANError.kOK if successful

setMeasurementPeriod(*self:* rev.CANEncoder, *period_ms: int*) → *rev.CANError*
Set the measurement period for velocity measurements of a quadrature encoder. When the SparkMax controller is in Brushless mode, this will not change any behavior.

The basic formula to calculate velocity is change in positon / change in time. This parameter sets the change in time for measurement.

Parameters **period_us** – Measurement period in milliseconds. This number may be between 1 and 100 (default).

Returns CANError.kOK if successful

setPosition(*self:* rev.CANEncoder, *position: float*) → *rev.CANError*
Set the position of the encoder.

Parameters **position** – Number of rotations of the motor

Returns CANError Set to CANError.kOK if successful

setPositionConversionFactor(*self:* rev.CANEncoder, *factor: float*) → *rev.CANError*
Set the conversion factor for position of the encoder. Multiplied by the native output units to give you position

Parameters **factor** – The conversion factor to multiply the native units by

Returns CANError Set to CANError.kOK if successful

setVelocityConversionFactor(*self:* rev.CANEncoder, *factor: float*) → *rev.CANError*
Set the conversion factor for velocity of the encoder. Multiplied by the native output units to give you velocity

Parameters **factor** – The conversion factor to multiply the native units by

Returns CANError Set to CANError.kOK if successful

1.1.4 CANError

class rev.**CANError**(*self:* rev.CANError, *value: int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kOk

kError

kTimeout

kNotImplmented

kHALError

kCantFindFirmware

kFirmwareTooOld

kFirmwareTooNew

kParamInvalidID

kParamMismatchType

kParamAccessMode

kParamInvalid

kParamNotImplementedDeprecated

kFollowConfigMismatch

kInvalid

kSetpointOutOfRange

kCantFindFirmware = <CANError.kCantFindFirmware: 5>

kError = <CANError.kError: 1>

kFirmwareTooNew = <CANError.kFirmwareTooNew: 7>

kFirmwareTooOld = <CANError.kFirmwareTooOld: 6>

kFollowConfigMismatch = <CANError.kFollowConfigMismatch: 13>

kHALError = <CANError.kHALError: 4>

kInvalid = <CANError.kInvalid: 14>

kNotImplmented = <CANError.kNotImplmented: 3>

kOk = <CANError.kOk: 0>

kParamAccessMode = <CANError.kParamAccessMode: 10>

kParamInvalid = <CANError.kParamInvalid: 11>

kParamInvalidID = <CANError.kParamInvalidID: 8>

kParamMismatchType = <CANError.kParamMismatchType: 9>

kParamNotImplementedDeprecated = <CANError.kParamNotImplementedDeprecated: 12>

kSetpointOutOfRange = <CANError.kSetpointOutOfRange: 15>

kTimeout = <CANError.kTimeout: 2>

property name

1.1.5 CANPIDController

class rev.**CANPIDController**
Bases: pybind11_builtins.pybind11_object

class AccelStrategy(*self: rev._rev.CANPIDController.AccelStrategy*, *value: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kTrapezoidal

kSCurve

kSCurve = <AccelStrategy.kSCurve: 1>

kTrapezoidal = <AccelStrategy.kTrapezoidal: 0>

property name

class ArbFFUnits(*self: rev._rev.CANPIDController.ArbFFUnits*, *value: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kVoltage

kPercentOut

kPercentOut = <ArbFFUnits.kPercentOut: 1>

kVoltage = <ArbFFUnits.kVoltage: 0>

property name

getD(*self:* rev.CANPIDController, *slotID: int = 0*) → float
Get the Derivative Gain constant of the PIDF controller on the SPARK MAX.

This uses the Get Parameter API and should be used infrequently. This function uses a non-blocking call and will return a cached value if the parameter is not returned by the timeout. The timeout can be changed by calling SetCANTimeout(int milliseconds)

Parameters **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns double D Gain value

getDFilter(*self:* rev.CANPIDController, *slotID: int = 0*) → float
Get the Derivative Filter constant of the PIDF controller on the SPARK MAX.

This uses the Get Parameter API and should be used infrequently. This function uses a non-blocking call and will return a cached value if the parameter is not returned by the timeout. The timeout can be changed by calling SetCANTimeout(int milliseconds)

Parameters **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns double D Filter value

getFF(*self:* rev.CANPIDController, *slotID: int = 0*) → float
Get the Feed-forward Gain constant of the PIDF controller on the SPARK MAX.

This uses the Get Parameter API and should be used infrequently. This function uses a non-blocking call and will return a cached value if the parameter is not returned by the timeout. The timeout can be changed by calling SetCANTimeout(int milliseconds)

Parameters **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns double F Gain value

getI(*self:* rev.CANPIDController, *slotID: int = 0*) → float
Get the Integral Gain constant of the PIDF controller on the SPARK MAX.

This uses the Get Parameter API and should be used infrequently. This function uses a non-blocking call and will return a cached value if the parameter is not returned by the timeout. The timeout can be changed by calling SetCANTimeout(int milliseconds)

Parameters **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns double I Gain value

getIAccum(*self:* rev.CANPIDController) → float
Get the I accumulator of the PID controller. This is useful when wishing to see what the I accumulator value is to help with PID tuning

Returns The value of the I accumulator

getIMaxAccum(*self:* rev.CANPIDController, *slotID: int = 0*) → float
Get the maximum I accumulator of the PID controller. This value is used to constrain the I accumulator to help manage integral wind-up

Parameters **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns The max value to contrain the I accumulator to

getIZone(*self:* rev.CANPIDController, *slotID: int = 0*) → float
Get the IZone constant of the PIDF controller on the SPARK MAX.

This uses the Get Parameter API and should be used infrequently. This function uses a non-blocking call and will return a cached value if the parameter is not returned by the timeout. The timeout can be changed by calling SetCANTimeout(int milliseconds)

Parameters **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns double IZone value

getOutputMax(*self:* rev.CANPIDController, *slotID: int = 0*) → float
Get the max output of the PIDF controller on the SPARK MAX.

This uses the Get Parameter API and should be used infrequently. This function uses a non-blocking call and will return a cached value if the parameter is not returned by the timeout. The timeout can be changed by calling SetCANTimeout(int milliseconds)

Parameters **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns double max value

getOutputMin (*self:* rev.CANPIDController, *slotID: int = 0*) → float
Get the min output of the PIDF controller on the SPARK MAX.

This uses the Get Parameter API and should be used infrequently. This function uses a non-blocking call and will return a cached value if the parameter is not returned by the timeout. The timeout can be changed by calling SetCANTimeout(int milliseconds)

Parameters **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns double min value

getP (*self:* rev.CANPIDController, *slotID: int = 0*) → float
Get the Proportional Gain constant of the PIDF controller on the SPARK MAX.

This uses the Get Parameter API and should be used infrequently. This function uses a non-blocking call and will return a cached value if the parameter is not returned by the timeout. The timeout can be changed by calling SetCANTimeout(int milliseconds)

Parameters **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns double P Gain value

getSmartMotionAccelStrategy (*self:* rev.CANPIDController, *slotID: int = 0*) → rev._rev.CANPIDController.AccelStrategy
Get the acceleration strategy used to control acceleration on the motor. The current strategy is trapezoidal motion profiling.

Parameters **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns The acceleration strategy to use for the automatically generated motion profile

getSmartMotionAllowedClosedLoopError (*self:* rev.CANPIDController, *slotID: int = 0*) → float
Get the allowed closed loop error of SmartMotion mode. This value is how much deviation from your setpoint is tolerated and is useful in preventing oscillation around your setpoint.

Parameters **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns The allowed deviation for your setpoint vs actual position in rotations

getSmartMotionMaxAccel (*self:* rev.CANPIDController, *slotID: int = 0*) → float
Get the maximum acceleration of the SmartMotion mode. This is the accleration that the motor velocity will increase at until the max velocity is reached

Parameters **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns The maxmimum acceleration for the motion profile in RPM per second

getSmartMotionMaxVelocity(*self:* rev.CANPIDController, *slotID: int = 0*) → float

Get the maximum velocity of the SmartMotion mode. This is the velocity that is reached in the middle of the profile and is what the motor should spend most of its time at

Parameters **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns The maxmimum cruise velocity for the motion profile in RPM

getSmartMotionMinOutputVelocity(*self:* rev.CANPIDController, *slotID: int = 0*) → float

Get the mimimum velocity of the SmartMotion mode. Any requested velocities below this value will be set to 0.

Parameters **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns The minimum velocity for the motion profile in RPM

setD(*self:* rev.CANPIDController, *gain: float*, *slotID: int = 0*) → *rev.CANError*

Set the Derivative Gain constant of the PIDF controller on the SPARK MAX. This uses the Set Parameter API and should be used infrequently. The parameter does not presist unless burnFlash() is called. The recommended method to configure this parameter is use to SPARK MAX GUI to tune and save parameters.

Parameters

- **gain** – The derivative gain value, must be positive
- **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns CANError Set to REV_OK if successful

setDFilter(*self:* rev.CANPIDController, *gain: float*, *slotID: int = 0*) → *rev.CANError*

Set the Derivative Filter constant of the PIDF controller on the SPARK MAX. This uses the Set Parameter API and should be used infrequently. The parameter does not presist unless burnFlash() is called.

Parameters

- **gain** – The derivative filter value, must be a positive number between 0 and 1
- **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns CANError Set to REV_OK if successful

setFF(*self:* rev.CANPIDController, *gain: float*, *slotID: int = 0*) → *rev.CANError*

Set the Feed-froward Gain constant of the PIDF controller on the SPARK MAX. This uses the Set Parameter API and should be used infrequently. The parameter does not presist unless burnFlash() is called. The recommended method to configure this parameter is use to SPARK MAX GUI to tune and save parameters.

Parameters

- **gain** – The feed-forward gain value
- **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns CANError Set to REV_OK if successful

setFeedbackDevice(*self:* rev.CANPIDController, *sensor:* rev.CANSensor) → *rev.CANError*

Set the controller's feedback device.

The default feedback device is assumed to be the integrated encoder. This is used to changed to another feedback device for the controller, such as an analog sensor.

If there is a limited range on the feedback sensor that should be observed by the PIDController, it can be set by calling SetFeedbackSensorRange() on the sensor object.

Parameters **sensor** – The sensor to be used as a feedback device

Returns CANError set to kOK if successful

setI(*self:* rev.CANPIDController, *gain: float*, *slotID: int = 0*) → rev.CANError

Set the Integral Gain constant of the PIDF controller on the SPARK MAX. This uses the Set Parameter API and should be used infrequently. The parameter does not presist unless burnFlash() is called. The recommended method to configure this parameter is use to SPARK MAX GUI to tune and save parameters.

Parameters

- **gain** – The integral gain value, must be positive
- **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns CANError Set to REV_OK if successful

setIAccum(*self:* rev.CANPIDController, *iAccum: float*) → rev.CANError

Set the I accumulator of the PID controller. This is useful when wishing to force a reset on the I accumulator of the PID controller. You can also preset values to see how it will respond to certain I characteristics

To use this function, the controller must be in a closed loop control mode by calling setReference()

Parameters **iAccum** – The value to set the I accumulator to

Returns CANError Set to kOK if successful

setIMaxAccum(*self:* rev.CANPIDController, *iMaxAccum: float*, *slotID: int = 0*) → rev.CANError

Configure the maximum I accumulator of the PID controller. This value is used to constrain the I accumulator to help manage integral wind-up

Parameters

- **iMaxAccum** – The max value to contrain the I accumulator to
- **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns CANError Set to kOK if successful

setIZone(*self:* rev.CANPIDController, *IZone: float*, *slotID: int = 0*) → rev.CANError

Set the IZone range of the PIDF controller on the SPARK MAX. This value specifies the range the |error| must be within for the integral constant to take effect.

This uses the Set Parameter API and should be used infrequently. The parameter does not presist unless burnFlash() is called. The recommended method to configure this parameter is to use the SPARK MAX GUI to tune and save parameters.

Parameters

- **gain** – The IZone value, must be positive. Set to 0 to disable
- **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns CANError Set to REV_OK if successful

setOutputRange(*self:* rev.CANPIDController, *min: float*, *max: float*, *slotID: int = 0*) → rev.CANError

Set the min amd max output for the closed loop mode.

This uses the Set Parameter API and should be used infrequently. The parameter does not presist unless burnFlash() is called. The recommended method to configure this parameter is to use the SPARK MAX GUI to tune and save parameters.

Parameters

- **min** – Reverse power minimum to allow the controller to output
- **max** – Forward power maximum to allow the controller to output
- **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns CANError Set to REV_OK if successful

setP(*self:* rev.CANPIDController, *gain: float*, *slotID: int = 0*) → rev.CANError

Set the Proportional Gain constant of the PIDF controller on the SPARK MAX. This uses the Set Parameter API and should be used infrequently. The parameter does not presist unless burnFlash() is called. The recommended method to configure this parameter is use to SPARK MAX GUI to tune and save parameters.

Parameters

- **gain** – The proportional gain value, must be positive
- **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns CANError Set to REV_OK if successful

setReference(*value: float*, *ctrl: rev._rev.ControlType*, *pidSlot: int = 0*, *arbFeedforward: float = 0*, *arbFFUnits: rev._rev.CANPIDController.ArbFFUnits = <ArbFFUnits.kVoltage: 0>*) → rev.CANError

Set the controller reference value based on the selected control mode.

Parameters

- **value** – The value to set depending on the control mode. For basic duty cycle control this should be a value between -1 and 1 Otherwise: Voltage Control: Voltage (volts) Velocity Control: Velocity (RPM) Position Control: Position (Rotations) Current Control: Current (Amps). The units can be changed for position and velocity by a scale factor using setPositionConversionFactor().
- **ctrl** – Is the control type
- **pidSlot** – for this command
- **arbFeedforward** – A value from -32.0 to 32.0 which is a voltage applied to the motor after the result of the specified control mode. The units for the parameter is Volts. This value is set after the control mode, but before any current limits or ramp rates.

Returns CANError Set to REV_OK if successful

setSmartMotionAccelStrategy(*self:* rev.CANPIDController, *accelStrategy: rev._rev.CANPIDController.AccelStrategy*, *slotID: int = 0*) → rev.CANError

Coming soon. Configure the acceleration strategy used to control acceleration on the motor. The current strategy is trapezoidal motion profiling.

Parameters

- **accelStrategy** – The acceleration strategy to use for the automatically generated motion profile
- **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns CANError Set to kOK if successful

setSmartMotionAllowedClosedLoopError(*self:* rev.CANPIDController, *allowedErr: float*, *slotID: int = 0*) → rev.CANError

Configure the allowed closed loop error of SmartMotion mode. This value is how much deviation from your setpoint is tolerated and is useful in preventing oscillation around your setpoint.

Parameters

- **allowedErr** – The allowed deviation for your setpoint vs actual position in rotations
- **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns CANError Set to kOK if successful

setSmartMotionMaxAccel(*self:* rev.CANPIDController, *maxAccel: float*, *slotID: int = 0*) → rev.CANError

Configure the maximum acceleration of the SmartMotion mode. This is the accleration that the motor velocity will increase at until the max velocity is reached

Parameters

- **maxAccel** – The maxmimum acceleration for the motion profile in RPM per second
- **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns CANError Set to kOK if successful

setSmartMotionMaxVelocity(*self:* rev.CANPIDController, *maxVel: float*, *slotID: int = 0*) → rev.CANError

Configure the maximum velocity of the SmartMotion mode. This is the velocity that is reached in the middle of the profile and is what the motor should spend most of its time at

Parameters

- **maxVel** – The maxmimum cruise velocity for the motion profile in RPM
- **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns CANError Set to kOK if successful

setSmartMotionMinOutputVelocity(*self:* rev.CANPIDController, *minVel: float*, *slotID: int = 0*) → rev.CANError

Configure the mimimum velocity of the SmartMotion mode. Any requested velocities below this value will be set to 0.

Parameters

- **minVel** – The minimum velocity for the motion profile in RPM
- **slotID** – Is the gain schedule slot, the value is a number between 0 and 3. Each slot has its own set of gain values and can be changed in each control frame using SetReference().

Returns CANError Set to kOK if successful

1.1.6 CANSensor

class rev.**CANSensor**
Bases: pybind11_builtins.pybind11_object

getInverted(*self:* rev.CANSensor) → bool
Get the phase of the CANSensor. This will just return false if the user tries to get the inversion of the hall effect.

setInverted(*self:* rev.CANSensor, *inverted: bool*) → *rev.CANError*
Set the phase of the CANSensor so that it is set to be in phase with the motor itself. This only works for quadrature encoders and analog sensors. This will throw an error if the user tries to set the inversion of the hall effect.

1.1.7 CANSparkMax

class rev.**CANSparkMax**(*self:* rev.CANSparkMax, *deviceID: int*, *type:* *rev._rev.CANSparkMaxLowLevel.MotorType*) → None
Bases: *rev.CANSparkMaxLowLevel*

Create a new SPARK MAX Controller

Parameters

- **deviceID** – The device ID.
- **type** – The motor type connected to the controller. Brushless motors must be connected to their matching color and the hall sensor plugged in. Brushed motors must be connected to the Red and Black terminals only.

class ExternalFollower(*self: rev._rev.CANSparkMax.ExternalFollower*) → None
Bases: pybind11_builtins.pybind11_object

property arbId

property configId

class FaultID(*self: rev._rev.CANSparkMax.FaultID*, *value: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kBrownout

kOvercurrent

kIWDTReset

kMotorFault

kSensorFault

kStall

kEEPROMCRC

kCANTX

kCANRX

kHasReset

kDRVFault

kOtherFault

kSoftLimitFwd

kSoftLimitRev

kHardLimitFwd

kHardLimitRev

kBrownout = <FaultID.kBrownout: 0>

kCANRX = <FaultID.kCANRX: 8>

kCANTX = <FaultID.kCANTX: 7>

kDRVFault = <FaultID.kDRVFault: 10>

kEEPROMCRC = <FaultID.kEEPROMCRC: 6>

kHardLimitFwd = <FaultID.kHardLimitFwd: 14>

kHardLimitRev = <FaultID.kHardLimitRev: 15>

kHasReset = <FaultID.kHasReset: 9>

kIWDTReset = <FaultID.kIWDTReset: 2>

kMotorFault = <FaultID.kMotorFault: 3>

kOtherFault = <FaultID.kOtherFault: 11>

kOvercurrent = <FaultID.kOvercurrent: 1>

kSensorFault = <FaultID.kSensorFault: 4>

kSoftLimitFwd = <FaultID.kSoftLimitFwd: 12>

kSoftLimitRev = <FaultID.kSoftLimitRev: 13>

kStall = <FaultID.kStall: 5>

property name

class IdleMode (*self: rev._rev.CANSparkMax.IdleMode*, *value: int*) → None

Bases: `pybind11_builtins.pybind11_object`

Members:

kCoast

kBrake

kBrake = <IdleMode.kBrake: 1>

kCoast = <IdleMode.kCoast: 0>

property name

class InputMode (*self: rev._rev.CANSparkMax.InputMode*, *value: int*) → None

Bases: `pybind11_builtins.pybind11_object`

Members:

kPWM

kCAN

kCAN = <InputMode.kCAN: 1>

kPWM = <InputMode.kPWM: 0>

property name

PIDWrite(*self:* rev.CANSparkMax, *output: float*) → None

class SoftLimitDirection(*self:* *rev._rev.CANSparkMax.SoftLimitDirection*, *value:* *int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kForward

kReverse

kForward = <SoftLimitDirection.kForward: 0>

kReverse = <SoftLimitDirection.kReverse: 1>

property name

burnFlash(*self:* rev.CANSparkMax) → *rev.CANError*

Writes all settings to flash.

clearFaults(*self:* rev.CANSparkMax) → *rev.CANError*

Clears all non-sticky faults.

Sticky faults must be cleared by resetting the motor controller.

disable(*self:* rev.CANSparkMax) → None

Common interface for disabling a motor.

disableVoltageCompensation(*self:* rev.CANSparkMax) → *rev.CANError*

Disables the voltage compensation setting for all modes on the SPARK MAX.

Returns CANError Set to CANError.kOK if successful

enableSoftLimit(*self:* rev.CANSparkMax, *direction: rev._rev.CANSparkMax.SoftLimitDirection*, *enable: bool*) → *rev.CANError*

Enable soft limits

Parameters

- **direction** – the direction of motion to restrict
- **enable** – set true to enable soft limits

enableVoltageCompensation(*self:* rev.CANSparkMax, *nominalVoltage:* *float*) → *rev.CANError*

Sets the voltage compensation setting for all modes on the SPARK MAX and enables voltage compensation.

Parameters **nominalVoltage** – Nominal voltage to compensate output to

Returns CANError Set to CANError.kOK if successful

follow(**args*, ***kwargs*)

Overloaded function.

1. follow(self: rev._rev.CANSparkMax, leader: rev._rev.CANSparkMax, invert: bool = False) -> rev._rev.CANError

Causes this controller's output to mirror the provided leader.

Only voltage output is mirrored. Settings changed on the leader do not affect the follower.

Following anything other than a CAN SPARK MAX is not officially supported.

Parameters

- **leader** – The motor controller to follow.

- **invert** – Set the follower to output opposite of the leader

2. follow(self: rev._rev.CANSparkMax, leader: rev._rev.CANSparkMax.ExternalFollower, deviceID: int, invert: bool = False) -> rev._rev.CANError

Causes this controller's output to mirror the provided leader.

Only voltage output is mirrored. Settings changed on the leader do not affect the follower.

Following anything other than a CAN SPARK MAX is not officially supported.

Parameters

- **leader** – The type of motor controller to follow (Talon SRX, Spark Max, etc.).
- **deviceID** – The CAN ID of the device to follow.
- **invert** – Set the follower to output opposite of the leader

get(*self:* rev.CANSparkMax) → float
Common interface for getting the current set speed of a speed controller.

Returns The current set speed. Value is between -1.0 and 1.0.

getAlternateEncoder(*self:* rev.CANSparkMax, *sensorType:* *rev._rev.CANEncoder.AlternateEncoderType*, *counts_per_rev:* *int*) → rev.CANEncoder
Returns an object for interfacing with a quadrature encoder connected to the alternate encoder mode data port pins. These are defined as:

Pin 4 (Forward Limit Switch): Index Pin 6 (Multi-function): Encoder A Pin 8 (Reverse Limit Switch): Encoder B

This call will disable support for the limit switch inputs.

getAnalog(*mode:* *rev._rev.CANAnalog.AnalogMode* = <*AnalogMode.kAbsolute:* *0*>) → rev.CANAnalog
Returns an object for interfacing with a connected analog sensor. By default, the AnalogMode is set to kAbsolute, thus treating the sensor as an absolute sensor.

getAppliedOutput(*self:* rev.CANSparkMax) → float
Returns motor controller's output duty cycle.

getBusVoltage(*self:* rev.CANSparkMax) → float
Returns the voltage fed into the motor controller.

getClosedLoopRampRate(*self:* rev.CANSparkMax) → float
Get the configured closed loop ramp rate

This is the maximum rate at which the motor controller's output is allowed to change.

Returns rampte rate time in seconds to go from 0 to full throttle.

getEncoder(*sensorType:* *rev._rev.CANEncoder.EncoderType* = <*EncoderType.kHallSensor:* *1*>, *counts_per_rev: int = 0*) → rev.CANEncoder
Returns an object for interfacing with the encoder connected to the encoder pins or front port of the SPARK MAX.

The default encoder type is assumed to be the hall effect for brushless. This can be modified for brushed DC to use an quadrature encoder.

getFault(*self:* rev.CANSparkMax, *faultID: rev._rev.CANSparkMax.FaultID*) → bool
Returns whether the fault with the given ID occurred.

getFaults(*self:* rev.CANSparkMax) → int
Returns fault bits.

getForwardLimitSwitch(*self:* rev.CANSparkMax, *polarity:* *rev._rev.CANDigitalInput.LimitSwitchPolarity*) → *rev.CANDigitalInput*
Returns an object for interfacing with the forward limit switch connected to the appropriate pins on the data port.

This call will disable support for the alternate encoder.

Parameters **polarity** – Whether the limit switch is normally open or normally closed.

getIdleMode(*self:* rev.CANSparkMax) → rev._rev.CANSparkMax.IdleMode
Gets the idle mode setting for the SPARK MAX.

This uses the Get Parameter API and should be used infrequently. This function uses a non-blocking call and will return a cached value if the parameter is not returned by the timeout. The timeout can be changed by calling SetCANTimeout(int milliseconds)

Returns IdleMode Idle mode setting

getInverted(*self:* rev.CANSparkMax) → bool
Common interface for returning the inversion state of a speed controller.

This call has no effect if the controller is a follower.

Returns isInverted The state of inversion, true is inverted.

getLastError(*self:* rev.CANSparkMax) → *rev.CANError*
All device errors are tracked on a per thread basis for all devices in that thread. This is meant to be called immediately following another call that has the possibility of throwing an error to validate if an error has occurred.

Returns the last error that was generated.

getMotorTemperature(*self:* rev.CANSparkMax) → float
Returns the motor temperature in Celsius.

getOpenLoopRampRate(*self:* rev.CANSparkMax) → float
Get the configured open loop ramp rate

This is the maximum rate at which the motor controller's output is allowed to change.

Returns rampte rate time in seconds to go from 0 to full throttle.

getOutputCurrent(*self:* rev.CANSparkMax) → float
Returns motor controller's output current in Amps.

getPIDController(*self:* rev.CANSparkMax) → *rev.CANPIDController*
Returns an object for interfacing with the integrated PID controller.

getReverseLimitSwitch(*self:* rev.CANSparkMax, *polarity:* *rev._rev.CANDigitalInput.LimitSwitchPolarity*) → *rev.CANDigitalInput*
Returns an object for interfacing with the reverse limit switch connected to the appropriate pins on the data port.

This call will disable support for the alternate encoder.

Parameters **polarity** – Whether the limit switch is normally open or normally closed.

getSoftLimit(*self:* rev.CANSparkMax, *direction:* *rev._rev.CANSparkMax.SoftLimitDirection*) → float
Get the soft limit setting in the controller

Parameters **direction** – the direction of motion to restrict

Returns position soft limit setting of the controller

getStickyFault (*self:* rev.CANSparkMax, *faultID: rev._rev.CANSparkMax.FaultID*) → bool
Returns whether the sticky fault with the given ID occurred.

getStickyFaults (*self:* rev.CANSparkMax) → int
Returns sticky fault bits.

getVoltageCompensationNominalVoltage (*self:* rev.CANSparkMax) → float
Get the configured voltage compensation nominal voltage value

Returns The nominal voltage for voltage compensation mode.

isFollower (*self:* rev.CANSparkMax) → bool
Returns whether the controller is following another controller

Returns True if this device is following another controller false otherwise

isSoftLimitEnabled (*self:* rev.CANSparkMax, *direction: rev._rev.CANSparkMax.SoftLimitDirection*) → bool
Returns true if the soft limit is enabled.

set (*self:* rev.CANSparkMax, *speed: float*) → None
Common interface for setting the speed of a speed controller.

Parameters **speed** – The speed to set. Value should be between -1.0 and 1.0.

setCANTimeout (*self:* rev.CANSparkMax, *milliseconds: int*) → *rev.CANError*
Sets timeout for sending CAN messages. A timeout of 0 also means that error handling will be done automatically by registering calls and waiting for responses, rather than needing to call GetLastError().

Parameters **milliseconds** – The timeout in milliseconds.

setClosedLoopRampRate (*self:* rev.CANSparkMax, *rate: float*) → *rev.CANError*
Sets the ramp rate for closed loop control modes.

This is the maximum rate at which the motor controller's output is allowed to change.

Parameters **rate** – Time in seconds to go from 0 to full throttle.

setIdleMode (*self:* rev.CANSparkMax, *mode: rev._rev.CANSparkMax.IdleMode*) → *rev.CANError*
Sets the idle mode setting for the SPARK MAX.

Parameters **mode** – Idle mode (coast or brake).

setInverted (*self:* rev.CANSparkMax, *isInverted: bool*) → None
Common interface for inverting direction of a speed controller.

This call has no effect if the controller is a follower. To invert a follower, see the follow() method.

Parameters **isInverted** – The state of inversion, true is inverted.

setOpenLoopRampRate (*self:* rev.CANSparkMax, *rate: float*) → *rev.CANError*
Sets the ramp rate for open loop control modes.

This is the maximum rate at which the motor controller's output is allowed to change.

Parameters **rate** – Time in seconds to go from 0 to full throttle.

setSecondaryCurrentLimit (*self:* rev.CANSparkMax, *limit: float*, *limitCycles: int = 0*) → *rev.CANError*
Sets the secondary current limit in Amps.

The motor controller will disable the output of the controller briefly if the current limit is exceeded to reduce the current. This limit is a simplified 'on/off' controller. This limit is enabled by default but is set higher than the default Smart Current Limit.

The time the controller is off after the current limit is reached is determined by the parameter limitCycles, which is the number of PWM cycles (20kHz). The recommended value is the default of 0 which is the minimum time and is part of a PWM cycle from when the over current is detected. This allows the controller to regulate the current close to the limit value.

The total time is set by the equation

@code t = (50us - t0) + 50us * limitCycles t = total off time after over current t0 = time from the start of the PWM cycle until over current is detected @endcode

Parameters

- **limit** – The current limit in Amps.
- **limitCycles** – The number of additional PWM cycles to turn the driver off after over-current is detected.

setSmartCurrentLimit(**args*, ***kwargs*)

Overloaded function.

1. setSmartCurrentLimit(self: rev._rev.CANSparkMax, limit: int) -> rev._rev.CANError

Sets the current limit in Amps.

The motor controller will reduce the controller voltage output to avoid surpassing this limit. This limit is enabled by default and used for brushless only. This limit is highly recommended when using the NEO brushless motor.

The NEO Brushless Motor has a low internal resistance, which can mean large current spikes that could be enough to cause damage to the motor and controller. This current limit provides a smarter strategy to deal with high current draws and keep the motor and controller operating in a safe region.

Parameters **limit** – The current limit in Amps.

2. setSmartCurrentLimit(self: rev._rev.CANSparkMax, stallLimit: int, freeLimit: int, limitRPM: int = 20000) -> rev._rev.CANError

Sets the current limit in Amps.

The motor controller will reduce the controller voltage output to avoid surpassing this limit. This limit is enabled by default and used for brushless only. This limit is highly recommended when using the NEO brushless motor.

The NEO Brushless Motor has a low internal resistance, which can mean large current spikes that could be enough to cause damage to the motor and controller. This current limit provides a smarter strategy to deal with high current draws and keep the motor and controller operating in a safe region.

The controller can also limit the current based on the RPM of the motor in a linear fashion to help with controllability in closed loop control. For a response that is linear the entire RPM range leave limit RPM at 0.

Parameters

- **stallLimit** – The current limit in Amps at 0 RPM.
- **freeLimit** – The current limit at free speed (5700RPM for NEO).
- **limitRPM** – RPM less than this value will be set to the stallLimit, RPM values greater than limitRPM will scale linearly to freeLimit

setSoftLimit(*self:* rev.CANSparkMax, *direction:* rev._rev.CANSparkMax.SoftLimitDirection, *limit: float*) → rev.CANError

Set the soft limit based on position. The default unit is rotations, but will match the unit scaling set by the user.

Note that this value is not scaled internally so care must be taken to make sure these units match the desired conversion

Parameters

- **direction** – the direction of motion to restrict
- **limit** – position soft limit of the controller

setVoltage(*self:* rev.CANSparkMax, *output: volts*) → None

Sets the voltage output of the SpeedController. This is equivillant to a call to SetReference(output, rev::ControlType::kVoltage). The behavior of this call differs slightly from the WPILib documetation for this call since the device internally sets the desired voltage (not a compensation value). That means that this *can* be a 'set-and-forget' call.

Parameters **output** – The voltage to output.

stopMotor(*self:* rev.CANSparkMax) → None

Common interface to stop the motor until Set is called again.

1.1.8 CANSparkMaxLowLevel

class rev.**CANSparkMaxLowLevel**(*self:* rev.CANSparkMaxLowLevel, *deviceID: int*, *type:* rev._rev.CANSparkMaxLowLevel.MotorType) → None

Bases: wpilib.ErrorBase, wpilib.interfaces.SpeedController

Create a new SPARK MAX Controller

Parameters

- **deviceID** – The device ID.
- **type** – The motor type connected to the controller. Brushless motors must be connected to their matching color and the hall sensor plugged in. Brushed motors must be connected to the Red and Black terminals only.

class MotorType(*self:* rev._rev.CANSparkMaxLowLevel.MotorType, *value: int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kBrushed

kBrushless

kBrushed = <MotorType.kBrushed: 0>

kBrushless = <MotorType.kBrushless: 1>

property name

class ParameterStatus(*self:* rev._rev.CANSparkMaxLowLevel.ParameterStatus, *value: int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kOK

kInvalidID

kMismatchType

kAccessMode

kInvalid

kNotImplementedDeprecated

kAccessMode = <ParameterStatus.kAccessMode: 3>

kInvalid = <ParameterStatus.kInvalid: 4>

kInvalidID = <ParameterStatus.kInvalidID: 1>

kMismatchType = <ParameterStatus.kMismatchType: 2>

kNotImplementedDeprecated = <ParameterStatus.kNotImplementedDeprecated: 5>

kOK = <ParameterStatus.kOK: 0>

property name

class PeriodicFrame (*self: rev._rev.CANSparkMaxLowLevel.PeriodicFrame*, *value: int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kStatus0

kStatus1

kStatus2

kStatus0 = <PeriodicFrame.kStatus0: 0>

kStatus1 = <PeriodicFrame.kStatus1: 1>

kStatus2 = <PeriodicFrame.kStatus2: 2>

property name

class PeriodicStatus0 (*self: rev._rev.CANSparkMaxLowLevel.PeriodicStatus0*) → None

Bases: pybind11_builtins.pybind11_object

property appliedOutput

property faults

property isFollower

property isInverted

property lock

property motorType

property roboRIO

property stickyFaults

property timestamp

class PeriodicStatus1 (*self: rev._rev.CANSparkMaxLowLevel.PeriodicStatus1*) → None

Bases: pybind11_builtins.pybind11_object

property busVoltage

property motorTemperature

property outputCurrent

property sensorVelocity

property timestamp

class PeriodicStatus2 (*self: rev._rev.CANSparkMaxLowLevel.PeriodicStatus2*) → None

Bases: pybind11_builtins.pybind11_object

property iAccum

property sensorPosition

property timestamp

class TelemetryID (*self: rev._rev.CANSparkMaxLowLevel.TelemetryID*, *value: int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kBusVoltage

kOutputCurrent

kVelocity

kPosition

kIAccum

kAppliedOutput

kMotorTemp

kFaults

kStickyFaults

kAnalogVoltage

kAnalogPosition

kAnalogVelocity

kAltEncPosition

kAltEncVelocity

kTotalStreams

kAltEncPosition = <TelemetryID.kAltEncPosition: 12>

kAltEncVelocity = <TelemetryID.kAltEncVelocity: 13>

kAnalogPosition = <TelemetryID.kAnalogPosition: 10>

kAnalogVelocity = <TelemetryID.kAnalogVelocity: 11>

kAnalogVoltage = <TelemetryID.kAnalogVoltage: 9>

kAppliedOutput = <TelemetryID.kAppliedOutput: 5>

kBusVoltage = <TelemetryID.kBusVoltage: 0>

kFaults = <TelemetryID.kFaults: 7>

kIAccum = <TelemetryID.kIAccum: 4>

kMotorTemp = <TelemetryID.kMotorTemp: 6>

kOutputCurrent = <TelemetryID.kOutputCurrent: 1>

kPosition = <TelemetryID.kPosition: 3>

kStickyFaults = <TelemetryID.kStickyFaults: 8>

kTotalStreams = <TelemetryID.kTotalStreams: 14>

kVelocity = <TelemetryID.kVelocity: 2>

property name

class TelemetryMessage(*self: rev._rev.CANSparkMaxLowLevel.TelemetryMessage*) → None
Bases: pybind11_builtins.pybind11_object

property id

property lowerBnd

property name

property timestamp

property units

property upperBnd

property value

static enableExternalUSBControl(*enable: bool*) → None
Allow external controllers to recieve control commands over USB. For example, a configuration where the heartbeat (and enable/disable) is sent by the main controller, but control frames are sent by other CAN devices over USB.

This is global for all controllers on the same bus.

This does not disable sending control frames from this device. To prevent conflicts, do not enable this feature and also send Set() for SetReference() from the controllers you wish to control.

Parameters **enable** – Enable or disable external control

getDeviceId(*self:* rev.CANSparkMaxLowLevel) → int
Get the configured Device ID of the SPARK MAX.

Returns int device ID

getFirmwareString(*self:* rev.CANSparkMaxLowLevel) → str
Get the firmware version of the SPARK MAX as a string.

Returns std::string Human readable firmware version string

getFirmwareVersion(*self:* rev.CANSparkMaxLowLevel) → int
Get the firmware version of the SPARK MAX.

Returns uint32_t Firmware version integer. Value is represented as 4 bytes, Major.Minor.Build H.Build L

getInitialMotorType(*self:* rev.CANSparkMaxLowLevel) → rev._rev.CANSparkMaxLowLevel.MotorType
Get the motor type setting from when the SparkMax was created.

This does not use the Get Parameter API which means it does not read what motor type is stored on the SparkMax itself. Instead, it reads the stored motor type from when the SparkMax object was first created.

Returns MotorType Motor type setting

getMotorType(*self:* rev.CANSparkMaxLowLevel) → rev._rev.CANSparkMaxLowLevel.MotorType
Get the motor type setting for the SPARK MAX.

This uses the Get Parameter API and should be used infrequently. This function uses a non-blocking call and will return a cached value if the parameter is not returned by the timeout. The timeout can be changed by calling SetCANTimeout(int milliseconds)

Returns MotorType Motor type setting

getSerialNumber(*self:* rev.CANSparkMaxLowLevel) → List[int]
Get the unique serial number of the SPARK MAX. Currently not implemented.

Returns std::vector<uint8_t> Vector of bytes representig the unique serial number

kAPIBuildVersion = 4

kAPIMajorVersion = 1

kAPIMinorVersion = 5

kAPIVersion = 17104900

restoreFactoryDefaults(*self:* rev.CANSparkMaxLowLevel, *persist: bool = False*) → *rev.CANError*
Restore motor controller parameters to factory default

Parameters **persist** – If true, burn the flash with the factory default parameters

Returns CANError Set to CANError::kOk if successful

setControlFramePeriodMs(*self:* rev.CANSparkMaxLowLevel, *periodMs: int*) → None
Set the control frame send period for the native CAN Send thread. To disable periodic sends, set periodMs to 0.

Parameters **periodMs** – The send period in milliseconds between 1ms and 100ms or set to 0 to disable periodic sends. Note this is not updated until the next call to Set() or SetReference().

static setEnable(*enable: bool*) → None
Send enabled or disabled command to controllers. This is global for all controllers on the same bus, and will only work for non-roboRIO targets in non-competiton use. This function will also not work if a roboRIO is present on the CAN bus.

This does not disable sending control frames from this device. To prevent conflicts, do not enable this feature and also send Set() for SetReference() from the controllers you wish to control.

Parameters **enable** – Enable or disable external control

setMotorType(*self:* rev.CANSparkMaxLowLevel, *type: rev._rev.CANSparkMaxLowLevel.MotorType*) → *rev.CANError*

setPeriodicFramePeriod(*self:* rev.CANSparkMaxLowLevel, *frame: rev._rev.CANSparkMaxLowLevel.PeriodicFrame*, *periodMs: int*) → *rev.CANError*
Set the rate of transmission for periodic frames from the SPARK MAX

Each motor controller sends back three status frames with different data at set rates. Use this function to change the default rates.

Defaults: Status0 - 10ms Status1 - 20ms Status2 - 50ms

This value is not stored in the FLASH after calling burnFlash() and is reset on powerup.

Refer to the SPARK MAX reference manual on details for how and when to configure this parameter.

Parameters

- **frameID** – The frame ID can be one of PeriodicFrame type
- **periodMs** – The rate the controller sends the frame to the controller.

Returns CANError Set to CANError::kOk if successful

1.1.9 ControlType

class rev.**ControlType**(*self:* rev.ControlType, *value: int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kDutyCycle

kVelocity

kVoltage

kPosition

kSmartMotion

kCurrent

kSmartVelocity

kCurrent = <ControlType.kCurrent: 5>

kDutyCycle = <ControlType.kDutyCycle: 0>

kPosition = <ControlType.kPosition: 3>

kSmartMotion = <ControlType.kSmartMotion: 4>

kSmartVelocity = <ControlType.kSmartVelocity: 6>

kVelocity = <ControlType.kVelocity: 1>

kVoltage = <ControlType.kVoltage: 2>

property name

1.1.10 IdleMode

class rev.**IdleMode**(*self: rev._rev.CANSparkMax.IdleMode*, *value: int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kCoast

kBrake

kBrake = <IdleMode.kBrake: 1>

kCoast = <IdleMode.kCoast: 0>

property name

1.1.11 LimitSwitch

class rev.**LimitSwitch**(*self: rev._rev.CANDigitalInput.LimitSwitch*, *value: int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kForward

kReverse

kForward = <LimitSwitch.kForward: 0>

kReverse = <LimitSwitch.kReverse: 1>

property name

1.1.12 LimitSwitchPolarity

class rev.**LimitSwitchPolarity**(*self: rev._rev.CANDigitalInput.LimitSwitchPolarity*, *value: int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kNormallyOpen

kNormallyClosed

kNormallyClosed = <LimitSwitchPolarity.kNormallyClosed: 1>

kNormallyOpen = <LimitSwitchPolarity.kNormallyOpen: 0>

property name

1.1.13 MotorType

class rev.**MotorType**(*self: rev._rev.CANSparkMaxLowLevel.MotorType*, *value: int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kBrushed

kBrushless

kBrushed = <MotorType.kBrushed: 0>

kBrushless = <MotorType.kBrushless: 1>

property name

1.1.14 SparkMax

class rev.**SparkMax**(*self:* rev.SparkMax, *channel: int*) → None
Bases: wpilib.PWMSpeedController

REV Robotics CAN speed controller controlled via PWM.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

I

K

L

M

N

O

P

R

S

T

U

V