RobotPy WPILib Documentation

Release 2020.3.2.4

RobotPy development team

Mar 02, 2020

Robot Programming

1 WPILib API **3**

1.1 wpilib Package . . . 3
1.2 wpilib.controller Package . . . 124
1.3 wpilib.drive Package . . . 138
1.4 wpilib.geometry Package . . . 146
1.5 wpilib.interfaces Package . . . 151
1.6 wpilib.kinematics Package . . . 158
1.7 wpilib.simulation Package . . . 174
1.8 wpilib.spline Package . . . 175
1.9 wpilib.trajectory Package . . . 179
1.10 wpilib.trajectory.constraint Package . . . 186

2 Indices and tables **191**

Index **193**

RobotPy WPILib is the source code for python wrappers around the C++ implementation of WPILib, the library used to interface with hardware for the FIRST Robotics Competition. Teams can use this library to write their robot code in Python, a powerful dynamic programming language.

Note: RobotPy is a community project and is not officially supported by FIRST. Please see the FAQ for more information.

CHAPTER 1

WPILib API

The WPI Robotics library (WPILib) is a set of classes that interfaces to the hardware in the FRC control system and your robot. There are classes to handle sensors, motors, the driver station, and a number of other utility functions like timing and field management. The library is designed to:

- Deal with all the low level interfacing to these components so you can concentrate on solving this year's "robot problem". This is a philosophical decision to let you focus on the higher-level design of your robot rather than deal with the details of the processor and the operating system.
- Understand everything at all levels by making the full source code of the library available. You can study (and modify) the algorithms used by the gyro class for oversampling and integration of the input signal or just ask the class for the current robot heading. You can work at any level.

1.1 wpilib Package

wpilib functions

wpilib.ADXL345_I2C(self, port, range, ...)	ADXL345 Accelerometer on I2C.
wpilib.ADXL345_SPI(self, port, range)	ADXL345 Accelerometer on SPI.
wpilib.ADXL362(*args, **kwargs)	ADXL362 SPI Accelerometer.
wpilib.ADXRS450_Gyro(*args, **kwargs)	Use a rate gyro to return the robots heading relative to a starting position.
wpilib.AddressableLED(self, port)	A class for driving addressable LEDs, such as WS2812s and NeoPixels.
wpilib.AnalogAccelerometer(*args, **kwargs)	Handle operation of an analog accelerometer.
wpilib.AnalogEncoder(self, analogInput)	Class for supporting continuous analog encoders, such as the US Digital MA3.
wpilib.AnalogGyro(*args, **kwargs)	Use a rate gyro to return the robots heading relative to a starting position.
wpilib.AnalogInput(self, channel)	Analog input class.

Continued on next page

Table 1 – continued from previous page

wpilib.AnalogOutput(self, channel)	MXP analog output class.
wpilib.AnalogPotentiometer(*args, **kwargs)	Class for reading analog potentiometers.
wpilib.AnalogTrigger(*args, **kwargs)	Overloaded function.
wpilib.AnalogTriggerOutput(self, trigger, ...)	Class to represent a specific output from an analog trigger.
wpilib.AnalogTriggerType(self, arg0)	Members:
wpilib.BuiltInAccelerometer(self, range)	Built-in accelerometer.
wpilib.CAN(*args, **kwargs)	High level class for interfacing with CAN devices conforming to the standard CAN spec.
wpilib.CANData(self)	
wpilib.CANStatus(self)	
wpilib.CameraServer	Provides a way to launch an out of process cscore-based camera service instance, for streaming or for image processing.
wpilib.Color(*args, **kwargs)	Represents colors that can be used with Addressable LEDs.
wpilib.Color8Bit(*args, **kwargs)	Represents colors that can be used with Addressable LEDs.
wpilib.Compressor(self, pcmID)	Class for operating a compressor connected to a %PCM (Pneumatic Control Module).
wpilib.Counter(*args, **kwargs)	Class for counting the number of ticks on a digital input channel.
wpilib.DMC60(self, channel)	Digilent DMC 60 Speed Controller.
wpilib.DigitalGlitchFilter(self)	Class to enable glitch filtering on a set of digital inputs.
wpilib.DigitalInput(self, channel)	Class to read a digital input.
wpilib.DigitalOutput(self, channel)	Class to write to digital outputs.
wpilib.DigitalSource(self)	DigitalSource Interface.
wpilib.DoubleSolenoid(*args, **kwargs)	DoubleSolenoid class for running 2 channels of high voltage Digital Output (PCM).
wpilib.DriverStation	Provide access to the network communication data to / from the Driver Station.
wpilib.DutyCycle(self, source)	Class to read a duty cycle PWM input.
wpilib.DutyCycleEncoder(*args, **kwargs)	Class for supporting duty cycle/PWM encoders, such as the US Digital MA3 with PWM Output, the CTRE Mag Encoder, the Rev Hex Encoder, and the AM Mag Encoder.
wpilib.Encoder(*args, **kwargs)	Class to read quad encoders.
wpilib.Error(*args, **kwargs)	Error object represents a library error.
wpilib.ErrorBase(self)	Base class for most objects.
wpilib.GyroBase(self)	GyroBase is the common base class for Gyro implementations such as AnalogGyro.
wpilib.I2C(self, port, deviceAddress)	I2C bus interface class.
wpilib.InterruptableSensorBase(self)	
wpilib.IterativeRobot(self)	IterativeRobot implements the IterativeRobotBase robot program framework.
wpilib.IterativeRobotBase(self, period)	IterativeRobotBase implements a specific type of robot program framework, extending the RobotBase class.
wpilib.Jaguar(self, channel)	Luminary Micro / Vex Robotics Jaguar Speed Controller with PWM control.

Continued on next page

Table 1 – continued from previous page

wpilib.Joystick(self, port)	Handle input from standard Joysticks connected to the Driver Station.
wpilib.LinearFilter(self, ffGains, fbGains)	This class implements a linear, digital filter.
wpilib.LiveWindow	The LiveWindow class is the public interface for putting sensors and actuators on the LiveWindow.
wpilib.MedianFilter(self, size)	A class that implements a moving-window median filter.
wpilib.MotorSafety(self)	This base class runs a watchdog timer and calls the subclass's StopMotor() function if the timeout expires.
wpilib.NidecBrushless(self, pwmChannel, ...)	Nidec Brushless Motor.
wpilib.Notifier(self, handler, None])	Create a Notifier for timer event notification.
wpilib.PWM(self, channel)	Class implements the PWM generation in the FPGA.
wpilib.PWMSparkMax(self, channel)	REV Robotics SPARK MAX Speed Controller.
wpilib.PWMSpeedController(self, channel)	Common base class for all PWM Speed Controllers.
wpilib.PWMTalonFX(self, channel)	Cross the Road Electronics (CTRE) Talon FX Speed Controller with PWM control.
wpilib.PWMTalonSRX(self, channel)	Cross the Road Electronics (CTRE) Talon SRX Speed Controller with PWM control.
wpilib.PWMVenom(self, channel)	Playing with Fusion Venom Smart Motor with PWM control.
wpilib.PWMVictorSPX(self, channel)	Cross the Road Electronics (CTRE) Victor SPX Speed Controller with PWM control.
wpilib.PowerDistributionPanel(*args, **kwargs)	Class for getting voltage, current, temperature, power and energy from the CAN PDP.
wpilib.Preferences	The preferences class provides a relatively simple way to save important values to the roboRIO to access the next time the roboRIO is booted.
wpilib.Relay(self, channel, direction)	Class for Spike style relay outputs.
wpilib.RobotBase(self)	Implement a Robot Program framework.
wpilib.RobotController	
wpilib.RobotState	
wpilib.SD540(self, channel)	Mindsensors SD540 Speed Controller.
wpilib.SPI(self, port)	SPI bus interface class.
wpilib.Sendable(self)	Interface for Sendable objects.
wpilib.SendableBase	
wpilib.SendableBuilder(self)	
wpilib.SendableBuilderImpl(self)	
wpilib.SendableChooser(self)	The SendableChooser class is a useful tool for presenting a selection of options to the SmartDashboard.
wpilib.SendableRegistry	The SendableRegistry class is the public interface for registering sensors and actuators for use on dashboards and LiveWindow.
wpilib.SensorUtil	Stores most recent status information as well as containing utility functions for checking channels and error processing.
wpilib.SerialPort(*args, **kwargs)	Driver for the RS-232 serial port on the roboRIO.
wpilib.Servo(self, channel)	Standard hobby style servo.
wpilib.SlewRateLimiter(self, rateLimit, ...)	A class that limits the rate of change of an input value.
wpilib.SmartDashboard	
wpilib.Solenoid(*args, **kwargs)	Solenoid class for running high voltage Digital Output (PCM).

Continued on next page

Table 1 – continued from previous page

wpilib.SolenoidBase(self, pcmID)	SolenoidBase class is the common base class for the Solenoid and DoubleSolenoid classes.
wpilib.Spark(self, channel)	REV Robotics SPARK Speed Controller.
wpilib.SpeedControllerGroup(self, *args)	
wpilib.Talon(self, channel)	Cross the Road Electronics (CTRE) Talon and Talon SR Speed Controller.
wpilib.TimedRobot(self, period)	TimedRobot implements the IterativeRobotBase robot program framework.
wpilib.Timer(self)	A wrapper for the frc::Timer class that returns unit-typed values.
wpilib.Ultrasonic(*args, **kwargs)	Ultrasonic rangefinder class.
wpilib.Victor(self, channel)	Vex Robotics Victor 888 Speed Controller.
wpilib.VictorSP(self, channel)	Vex Robotics Victor SP Speed Controller.
wpilib.Watchdog(self, timeout, callback, None])	A class that's a wrapper around a watchdog timer.
wpilib.XboxController(self, port)	Handle input from Xbox 360 or Xbox One controllers connected to the Driver Station.

1.1.1 wpilib functions

wpilib.**getCurrentThreadPriority**() → Tuple[int, bool]
Get the thread priority for the current thread

Parameters **isRealTime** – Set to true if thread is realtime, otherwise false.

Returns The current thread priority. Scaled 1-99.

wpilib.**getThreadPriority**(*thread: std::thread*) → Tuple[int, bool]
Get the thread priority for the specified thread.

Parameters

- **thread** – Reference to the thread to get the priority for.
- **isRealTime** – Set to true if thread is realtime, otherwise false.

Returns The current thread priority. Scaled 1-99, with 1 being highest.

wpilib.**getTime**() → seconds
@brief Gives real-time clock system time with nanosecond resolution

Returns The time, just in case you want the robot to start autonomous at 8pm on Saturday.

wpilib.**run**(*robot_class*, ***kwargs*)
This function gets called in robot.py like so:

```
if __name__ == '__main__':
    wpilib.run(MyRobot)
```

This function loads available entry points, parses arguments, and sets things up specific to RobotPy so that the robot can run. This function is used whether the code is running on the roboRIO or a simulation.

Parameters

- **robot_class** – A class that inherits from *RobotBase*
- ****kwargs** – Keyword arguments that will be passed to the executed entry points

Returns This function should never return

wpilib.**setCurrentThreadPriority**(*realTime: bool*, *priority: int*) → bool
Sets the thread priority for the current thread

Parameters

- **realTime** – Set to true to set a realtime priority, false for standard priority.
- **priority** – Priority to set the thread to. Scaled 1-99, with 1 being highest. On RoboRIO, priority is ignored for non realtime setting.

Returns The success state of setting the priority

wpilib.**setThreadPriority**(*thread: std::thread*, *realTime: bool*, *priority: int*) → bool
Sets the thread priority for the specified thread

Parameters

- **thread** – Reference to the thread to set the priority of.
- **realTime** – Set to true to set a realtime priority, false for standard priority.
- **priority** – Priority to set the thread to. Scaled 1-99, with 1 being highest. On RoboRIO, priority is ignored for non realtime setting.

Returns The success state of setting the priority

wpilib.**wait**(*seconds: seconds*) → None
Pause the task for a specified time.

Pause the execution of the program for a specified period of time given in seconds. Motors will continue to run at their last assigned values, and sensors will continue to update. Only the task containing the wait will pause until the wait time is expired.

Parameters **seconds** – Length of time to pause, in seconds.

1.1.2 ADXL345_I2C

class wpilib.**ADXL345_I2C**(*port: frc::I2C::Port*, *range: wpilib.interfaces._interfaces.Accelerometer.Range = Range.kRange_2G*, *deviceAddress: int = 29*) → None
Bases: *wpilib.ErrorBase*, *wpilib.interfaces.Accelerometer*, *wpilib.Sendable*

ADXL345 Accelerometer on I2C.

This class allows access to a Analog Devices ADXL345 3-axis accelerometer on an I2C bus. This class assumes the default (not alternate) sensor address of 0x1D (7-bit address).

Constructs the ADXL345 Accelerometer over I2C.

Parameters

- **port** – The I2C port the accelerometer is attached to
- **range** – The range (+ or -) that the accelerometer will measure
- **deviceAddress** – The I2C address of the accelerometer (0x1D or 0x53)

class AllAxes() → None
Bases: pybind11_builtins.pybind11_object

XAxis

YAxis

ZAxis

class Axes(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kAxis_X

kAxis_Y

kAxis_Z

kAxis_X = Axes.kAxis_X

kAxis_Y = Axes.kAxis_Y

kAxis_Z = Axes.kAxis_Z

name
(self: handle) -> str

getAcceleration(*axis: wpilib._wpilib.ADXL345_I2C.Axes*) → float
Get the acceleration of one axis in Gs.

Parameters **axis** – The axis to read from.

Returns Acceleration of the ADXL345 in Gs.

getAccelerations() → wpilib._wpilib.ADXL345_I2C.AllAxes
Get the acceleration of all axes in Gs.

Returns An object containing the acceleration measured on each axis of the ADXL345 in Gs.

getX() → float

getY() → float

getZ() → float

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

setRange(*range: wpilib.interfaces._interfaces.Accelerometer.Range*) → None

1.1.3 ADXL345_SPI

class wpilib.**ADXL345_SPI**(*port: frc::SPI::Port*, *range: wpilib.interfaces._interfaces.Accelerometer.Range* = *Range.kRange_2G*) → None
Bases: *wpilib.ErrorBase*, *wpilib.interfaces.Accelerometer*, *wpilib.Sendable*

ADXL345 Accelerometer on SPI.

This class allows access to an Analog Devices ADXL345 3-axis accelerometer via SPI. This class assumes the sensor is wired in 4-wire SPI mode.

Constructor.

Parameters

- **port** – The SPI port the accelerometer is attached to
- **range** – The range (+ or -) that the accelerometer will measure

class AllAxes() → None
Bases: pybind11_builtins.pybind11_object

XAxis

YAxis

ZAxis

class Axes(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kAxis_X

kAxis_Y

kAxis_Z

kAxis_X = Axes.kAxis_X

kAxis_Y = Axes.kAxis_Y

kAxis_Z = Axes.kAxis_Z

name
(self: handle) -> str

getAcceleration(*axis: wpilib._wpilib.ADXL345_SPI.Axes*) → float
Get the acceleration of one axis in Gs.

Parameters **axis** – The axis to read from.

Returns Acceleration of the ADXL345 in Gs.

getAccelerations() → wpilib._wpilib.ADXL345_SPI.AllAxes
Get the acceleration of all axes in Gs.

Returns An object containing the acceleration measured on each axis of the ADXL345 in Gs.

getX() → float

getY() → float

getZ() → float

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

setRange(*range: wpilib.interfaces._interfaces.Accelerometer.Range*) → None

1.1.4 ADXL362

class wpilib.**ADXL362**(**args*, ***kwargs*)
Bases: *wpilib.ErrorBase*, *wpilib.interfaces.Accelerometer*, *wpilib.Sendable*

ADXL362 SPI Accelerometer.

This class allows access to an Analog Devices ADXL362 3-axis accelerometer.

Overloaded function.

1. __init__(self: wpilib._wpilib.ADXL362, range: wpilib.interfaces._interfaces.Accelerometer.Range = Range.kRange_2G) -> None

Constructor. Uses the onboard CS1.

Parameters **range** – The range (+ or -) that the accelerometer will measure.

2. __init__(self: wpilib._wpilib.ADXL362, port: frc::SPI::Port, range: wpilib.interfaces._interfaces.Accelerometer.Range = Range.kRange_2G) -> None

Constructor.

Parameters

- **port** – The SPI port the accelerometer is attached to
- **range** – The range (+ or -) that the accelerometer will measure.

class AllAxes() → None
Bases: pybind11_builtins.pybind11_object

XAxis

YAxis

ZAxis

class Axes(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kAxis_X

kAxis_Y

kAxis_Z

kAxis_X = Axes.kAxis_X

kAxis_Y = Axes.kAxis_Y

kAxis_Z = Axes.kAxis_Z

name
(self: handle) -> str

getAcceleration(*axis: wpilib._wpilib.ADXL362.Axes*) → float
Get the acceleration of one axis in Gs.

Parameters **axis** – The axis to read from.

Returns Acceleration of the ADXL362 in Gs.

getAccelerations() → wpilib._wpilib.ADXL362.AllAxes
Get the acceleration of all axes in Gs.

Returns An object containing the acceleration measured on each axis of the ADXL362 in Gs.

getX() → float

getY() → float

getZ() → float

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

setRange(*range: wpilib.interfaces._interfaces.Accelerometer.Range*) → None

1.1.5 ADXRS450_Gyro

class wpilib.**ADXRS450_Gyro**(**args*, ***kwargs*)
Bases: *wpilib.GyroBase*

Use a rate gyro to return the robots heading relative to a starting position.

The Gyro class tracks the robots heading based on the starting position. As the robot rotates the new heading is computed by integrating the rate of rotation returned by the sensor. When the class is instantiated, it does a short

calibration routine where it samples the gyro while at rest to determine the default offset. This is subtracted from each sample to determine the heading.

This class is for the digital ADXRS450 gyro sensor that connects via SPI. Only one instance of an ADXRS Gyro is supported.

Overloaded function.

1. __init__(self: wpilib._wpilib.ADXRS450_Gyro) -> None

Gyro constructor on onboard CS0.

2. __init__(self: wpilib._wpilib.ADXRS450_Gyro, port: wpilib._wpilib.SPI.Port) -> None

Gyro constructor on the specified SPI port.

Parameters **port** – The SPI port the gyro is attached to.

calibrate() → None
Initialize the gyro.

Calibrate the gyro by running for a number of samples and computing the center value. Then use the center value as the Accumulator center value for subsequent measurements.

It's important to make sure that the robot is not moving while the centering calculations are in progress, this is typically done when the robot is first turned on while it's sitting at rest before the competition starts.

getAngle() → float
Return the actual angle in degrees that the robot is currently facing.

The angle is based on the current accumulator value corrected by the oversampling rate, the gyro type and the A/D calibration values. The angle is continuous, that is it will continue from 360->361 degrees. This allows algorithms that wouldn't want to see a discontinuity in the gyro output as it sweeps from 360 to 0 on the second time around.

Returns the current heading of the robot in degrees. This heading is based on integration of the returned rate from the gyro.

getRate() → float
Return the rate of rotation of the gyro

The rate is based on the most recent reading of the gyro analog value

Returns the current rate in degrees per second

reset() → None
Reset the gyro.

Resets the gyro to a heading of zero. This can be used if there is significant drift in the gyro and it needs to be recalibrated after it has been running.

1.1.6 AddressableLED

class wpilib.**AddressableLED**(*port: int*) → None
Bases: *wpilib.ErrorBase*

A class for driving addressable LEDs, such as WS2812s and NeoPixels.

Only 1 LED driver is currently supported by the roboRIO.

Constructs a new driver for a specific port.

Parameters **port** – the output port to use (Must be a PWM header)

class LEDData(**args*, ***kwargs*)

Bases: pybind11_builtins.pybind11_object

Overloaded function.

1. __init__(self: wpilib._wpilib.AddressableLED.LEDData) -> None
2. __init__(self: wpilib._wpilib.AddressableLED.LEDData, r: int, g: int, b: int) -> None

setHSV(*h: int*, *s: int*, *v: int*) → None

A helper method to set all values of the LED.

Parameters

- **h** – the h value [0-180]
- **s** – the s value [0-255]
- **v** – the v value [0-255]

setLED(**args*, ***kwargs*)

Overloaded function.

1. setLED(self: wpilib._wpilib.AddressableLED.LEDData, color: wpilib._wpilib.Color) -> None
2. setLED(self: wpilib._wpilib.AddressableLED.LEDData, color: wpilib._wpilib.Color8Bit) -> None

setRGB(*r: int*, *g: int*, *b: int*) → None

A helper method to set all values of the LED.

Parameters

- **r** – the r value [0-255]
- **g** – the g value [0-255]
- **b** – the b value [0-255]

setBitTiming(*lowTime0: nanoseconds*, *highTime0: nanoseconds*, *lowTime1: nanoseconds*, *highTime1: nanoseconds*) → None

Sets the bit timing.

By default, the driver is set up to drive WS2812s, so nothing needs to be set for those.

Parameters

- **lowTime0** – low time for 0 bit
- **highTime0** – high time for 0 bit
- **lowTime1** – low time for 1 bit
- **highTime1** – high time for 1 bit

setData() → None

Sets the led output data.

If the output is enabled, this will start writing the next data cycle. It is safe to call, even while output is enabled.

Parameters **ledData** – the buffer to write

setLength(*length: int*) → None

Sets the length of the LED strip.

Calling this is an expensive call, so its best to call it once, then just update data.

The max length is 5460 LEDs.

Parameters **length** – the strip length

setSyncTime(*syncTime: microseconds*) → None

Sets the sync time.

The sync time is the time to hold output so LEDs enable. Default set for WS2812.

Parameters **syncTimeMicroSeconds** – the sync time

start() → None
Starts the output.

The output writes continously.

stop() → None
Stops the output.

1.1.7 AnalogAccelerometer

class wpilib.**AnalogAccelerometer**(**args*, ***kwargs*)
Bases: *wpilib.ErrorBase*, *wpilib.interfaces.PIDSource*, *wpilib.Sendable*

Handle operation of an analog accelerometer.

The accelerometer reads acceleration directly through the sensor. Many sensors have multiple axis and can be treated as multiple devices. Each is calibrated by finding the center value over a period of time.

Overloaded function.

1. __init__(self: wpilib._wpilib.AnalogAccelerometer, channel: int) -> None

Create a new instance of an accelerometer.

The constructor allocates desired analog input.

Parameters **channel** – The channel number for the analog input the accelerometer is connected to

2. __init__(self: wpilib._wpilib.AnalogAccelerometer, channel: frc::AnalogInput) -> None

Create a new instance of Accelerometer from an existing AnalogInput.

Make a new instance of accelerometer given an AnalogInput. This is particularly useful if the port is going to be read as an analog channel as well as through the Accelerometer class.

Parameters **channel** – The existing AnalogInput object for the analog input the accelerometer is connected to

getAcceleration() → float
Return the acceleration in Gs.

The acceleration is returned units of Gs.

Returns The current acceleration of the sensor in Gs.

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

pidGet() → float
Get the Acceleration for the PID Source parent.

Returns The current acceleration in Gs.

setSensitivity(*sensitivity: float*) → None
Set the accelerometer sensitivity.

This sets the sensitivity of the accelerometer used for calculating the acceleration. The sensitivity varies by accelerometer model. There are constants defined for various models.

Parameters **sensitivity** – The sensitivity of accelerometer in Volts per G.

setZero (*zero: float*) → None
Set the voltage that corresponds to 0 G.

The zero G voltage varies by accelerometer model. There are constants defined for various models.

Parameters **zero** – The zero G voltage.

1.1.8 AnalogEncoder

class wpilib.**AnalogEncoder** (*analogInput: frc::AnalogInput*) → None
Bases: *wpilib.ErrorBase*, *wpilib.Sendable*

Class for supporting continuous analog encoders, such as the US Digital MA3.

Construct a new AnalogEncoder attached to a specific AnalogInput.

Parameters **analogInput** – the analog input to attach to

get () → turns
Get the encoder value since the last reset.

This is reported in rotations since the last reset.

Returns the encoder value in rotations

getDistance () → float
Get the distance the sensor has driven since the last reset as scaled by the value from SetDistancePerRotation.

Returns The distance driven since the last reset

getDistancePerRotation () → float
Get the distance per rotation for this encoder.

Returns The scale factor that will be used to convert rotation to useful units.

getPositionOffset () → float
Get the offset of position relative to the last reset.

GetPositionInRotation() - GetPositionOffset() will give an encoder absolute position relative to the last reset. This could potentially be negative, which needs to be accounted for.

Returns the position offset

initSendable (*builder: wpilib._wpilib.SendableBuilder*) → None

reset () → None
Reset the Encoder distance to zero.

setDistancePerRotation (*distancePerRotation: float*) → None
Set the distance per rotation of the encoder. This sets the multiplier used to determine the distance driven based on the rotation value from the encoder. Set this value based on the how far the mechanism travels in 1 rotation of the encoder, and factor in gearing reductions following the encoder shaft. This distance can be in any units you like, linear or angular.

Parameters **distancePerRotation** – the distance per rotation of the encoder

1.1.9 AnalogGyro

class wpilib.**AnalogGyro** (**args*, ***kwargs*)
Bases: *wpilib.GyroBase*

Use a rate gyro to return the robots heading relative to a starting position. The Gyro class tracks the robots heading based on the starting position. As the robot rotates the new heading is computed by integrating the rate of rotation returned by the sensor. When the class is instantiated, it does a short calibration routine where it samples the gyro while at rest to determine the default offset. This is subtracted from each sample to determine the heading. This gyro class must be used with a channel that is assigned one of the Analog accumulators from the FPGA. See AnalogInput for the current accumulator assignments.

This class is for gyro sensors that connect to an analog input.

Overloaded function.

1. __init__(self: wpilib._wpilib.AnalogGyro, channel: int) -> None

Gyro constructor using the Analog Input channel number.

Parameters **channel** – The analog channel the gyro is connected to. Gyros can only be used on on-board Analog Inputs 0-1.

2. __init__(self: wpilib._wpilib.AnalogGyro, channel: wpilib._wpilib.AnalogInput) -> None

Gyro constructor with a precreated AnalogInput object.

Use this constructor when the analog channel needs to be shared. This object will not clean up the AnalogInput object when using this constructor.

Parameters **channel** – A pointer to the AnalogInput object that the gyro is connected to.

3. __init__(self: wpilib._wpilib.AnalogGyro, channel: int, center: int, offset: float) -> None

Gyro constructor using the Analog Input channel number with parameters for presetting the center and offset values. Bypasses calibration.

Parameters

- **channel** – The analog channel the gyro is connected to. Gyros can only be used on on-board Analog Inputs 0-1.
- **center** – Preset uncalibrated value to use as the accumulator center value.
- **offset** – Preset uncalibrated value to use as the gyro offset.

4. __init__(self: wpilib._wpilib.AnalogGyro, channel: wpilib._wpilib.AnalogInput, center: int, offset: float) -> None

Gyro constructor with a precreated AnalogInput object and calibrated parameters.

Use this constructor when the analog channel needs to be shared. This object will not clean up the AnalogInput object when using this constructor.

Parameters

- **channel** – A pointer to the AnalogInput object that the gyro is connected to.
- **center** – Preset uncalibrated value to use as the accumulator center value.
- **offset** – Preset uncalibrated value to use as the gyro offset.

calibrate() → None

getAngle() → float
Return the actual angle in degrees that the robot is currently facing.

The angle is based on the current accumulator value corrected by the oversampling rate, the gyro type and the A/D calibration values. The angle is continuous, that is it will continue from 360->361 degrees. This allows algorithms that wouldn't want to see a discontinuity in the gyro output as it sweeps from 360 to 0 on the second time around.

Returns The current heading of the robot in degrees. This heading is based on integration of the returned rate from the gyro.

getCenter() → int
Return the gyro center value. If run after calibration, the center value can be used as a preset later.

Returns the current center value

getOffset() → float
Return the gyro offset value. If run after calibration, the offset value can be used as a preset later.

Returns the current offset value

getRate() → float
Return the rate of rotation of the gyro

The rate is based on the most recent reading of the gyro analog value

Returns the current rate in degrees per second

initGyro() → None
Initialize the gyro.

Calibration is handled by Calibrate().

kAverageBits = 0

kCalibrationSampleTime = 5.0

kDefaultVoltsPerDegreePerSecond = 0.007

kOversampleBits = 10

kSamplesPerSecond = 50.0

reset() → None
Reset the gyro.

Resets the gyro to a heading of zero. This can be used if there is significant drift in the gyro and it needs to be recalibrated after it has been running.

setDeadband(*volts: float*) → None
Set the size of the neutral zone.

Any voltage from the gyro less than this amount from the center is considered stationary. Setting a deadband will decrease the amount of drift when the gyro isn't rotating, but will make it less accurate.

Parameters **volts** – The size of the deadband in volts

setSensitivity(*voltsPerDegreePerSecond: float*) → None
Set the gyro sensitivity.

This takes the number of volts/degree/second sensitivity of the gyro and uses it in subsequent calculations to allow the code to work with multiple gyros. This value is typically found in the gyro datasheet.

Parameters **voltsPerDegreePerSecond** – The sensitivity in Volts/degree/second

1.1.10 AnalogInput

class wpilib.**AnalogInput**(*channel: int*) → None
Bases: *wpilib.ErrorBase*, *wpilib.interfaces.PIDSource*, *wpilib.Sendable*

Analog input class.

Connected to each analog channel is an averaging and oversampling engine. This engine accumulates the specified (by SetAverageBits() and SetOversampleBits()) number of samples before returning a new value. This is not a sliding window average. The only difference between the oversampled samples and the averaged samples is that the oversampled samples are simply accumulated effectively increasing the resolution, while the averaged samples are divided by the number of samples to retain the resolution, but get more stable values.

Construct an analog input.

Parameters **channel** – The channel number on the roboRIO to represent. 0-3 are on-board 4-7 are on the MXP port.

getAccumulatorCount() → int
Read the number of accumulated values.

Read the count of the accumulated values since the accumulator was last Reset().

Returns The number of times samples from the channel were accumulated.

getAccumulatorOutput(*value: int*, *count: int*) → None
Read the accumulated value and the number of accumulated values atomically.

This function reads the value and count from the FPGA atomically. This can be used for averaging.

Parameters

- **value** – Reference to the 64-bit accumulated output.
- **count** – Reference to the number of accumulation cycles.

getAccumulatorValue() → int
Read the accumulated value.

Read the value that has been accumulating. The accumulator is attached after the oversample and average engine.

Returns The 64-bit value accumulated since the last Reset().

getAverageBits() → int
Get the number of averaging bits previously configured.

This gets the number of averaging bits from the FPGA. The actual number of averaged samples is 2^bits. The averaging is done automatically in the FPGA.

Returns Number of bits of averaging previously configured.

getAverageValue() → int
Get a sample from the output of the oversample and average engine for this channel.

The sample is 12-bit + the bits configured in SetOversampleBits(). The value configured in SetAverageBits() will cause this value to be averaged 2**bits number of samples.

This is not a sliding window. The sample will not change until 2**(OversampleBits + AverageBits) samples have been acquired from the module on this channel.

Use GetAverageVoltage() to get the analog value in calibrated units.

Returns A sample from the oversample and average engine for this channel.

getAverageVoltage() → float
Get a scaled sample from the output of the oversample and average engine for this channel.

The value is scaled to units of Volts using the calibrated scaling data from GetLSBWeight() and GetOffset().

Using oversampling will cause this value to be higher resolution, but it will update more slowly.

Using averaging will cause this value to be more stable, but it will update more slowly.

Returns A scaled sample from the output of the oversample and average engine for this channel.

getChannel() → int
Get the channel number.

Returns The channel number.

getLSBWeight() → int
Get the factory scaling least significant bit weight constant.

Volts = ((LSB_Weight * 1e-9) * raw) - (Offset * 1e-9)

Returns Least significant bit weight.

getOffset() → int
Get the factory scaling offset constant.

Volts = ((LSB_Weight * 1e-9) * raw) - (Offset * 1e-9)

Returns Offset constant.

getOversampleBits() → int
Get the number of oversample bits previously configured.

This gets the number of oversample bits from the FPGA. The actual number of oversampled values is 2^bits. The oversampling is done automatically in the FPGA.

Returns Number of bits of oversampling previously configured.

static getSampleRate() → float
Get the current sample rate for all channels

Returns Sample rate.

getValue() → int
Get a sample straight from this channel.

The sample is a 12-bit value representing the 0V to 5V range of the A/D converter in the module. The units are in A/D converter codes. Use GetVoltage() to get the analog value in calibrated units.

Returns A sample straight from this channel.

getVoltage() → float
Get a scaled sample straight from this channel.

The value is scaled to units of Volts using the calibrated scaling data from GetLSBWeight() and GetOffset().

Returns A scaled sample straight from this channel.

initAccumulator() → None
Initialize the accumulator.

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

isAccumulatorChannel() → bool
Is the channel attached to an accumulator.

Returns The analog input is attached to an accumulator.

kAccumulatorModuleNumber = 1

kAccumulatorNumChannels = 2

pidGet() → float
Get the Average value for the PID Source base object.

Returns The average voltage.

resetAccumulator() → None
Resets the accumulator to the initial value.

setAccumulatorCenter(*center: int*) → None
Set the center value of the accumulator.

The center value is subtracted from each A/D value before it is added to the accumulator. This is used for the center value of devices like gyros and accelerometers to take the device offset into account when integrating.

This center value is based on the output of the oversampled and averaged source from the accumulator channel. Because of this, any non-zero oversample bits will affect the size of the value for this field.

setAccumulatorDeadband(*deadband: int*) → None
Set the accumulator's deadband.

setAccumulatorInitialValue(*value: int*) → None
Set an initial value for the accumulator.

This will be added to all values returned to the user.

Parameters **initialValue** – The value that the accumulator should start from when reset.

setAverageBits(*bits: int*) → None
Set the number of averaging bits.

This sets the number of averaging bits. The actual number of averaged samples is 2^bits.

Use averaging to improve the stability of your measurement at the expense of sampling rate. The averaging is done automatically in the FPGA.

Parameters **bits** – Number of bits of averaging.

setOversampleBits(*bits: int*) → None
Set the number of oversample bits.

This sets the number of oversample bits. The actual number of oversampled values is 2^bits. Use oversampling to improve the resolution of your measurements at the expense of sampling rate. The oversampling is done automatically in the FPGA.

Parameters **bits** – Number of bits of oversampling.

static setSampleRate(*samplesPerSecond: float*) → None
Set the sample rate per channel for all analog channels.

The maximum rate is 500kS/s divided by the number of channels in use. This is 62500 samples/s per channel.

Parameters **samplesPerSecond** – The number of samples per second.

setSimDevice(*device: int*) → None
Indicates this input is used by a simulated device.

Parameters **device** – simulated device handle

1.1.11 AnalogOutput

class wpilib.**AnalogOutput**(*channel: int*) → None
Bases: *wpilib.ErrorBase*, *wpilib.Sendable*

MXP analog output class.

Construct an analog output on the given channel.

All analog outputs are located on the MXP port.

Parameters **channel** – The channel number on the roboRIO to represent.

getChannel() → int
Get the channel of this AnalogOutput.

getVoltage() → float
Get the voltage of the analog output

Returns The value in Volts, from 0.0 to +5.0

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

setVoltage(*voltage: float*) → None
Set the value of the analog output.

Parameters **voltage** – The output value in Volts, from 0.0 to +5.0

1.1.12 AnalogPotentiometer

class wpilib.**AnalogPotentiometer**(**args*, ***kwargs*)
Bases: *wpilib.ErrorBase*, *wpilib.interfaces.Potentiometer*, *wpilib.Sendable*

Class for reading analog potentiometers. Analog potentiometers read in an analog voltage that corresponds to a position. The position is in whichever units you choose, by way of the scaling and offset constants passed to the constructor.

Overloaded function.

1. __init__(self: wpilib._wpilib.AnalogPotentiometer, channel: int, fullRange: float = 1.0, offset: float = 0.0) -> None

Construct an Analog Potentiometer object from a channel number.

Use the fullRange and offset values so that the output produces meaningful values. I.E: you have a 270 degree potentiometer and you want the output to be degrees with the halfway point as 0 degrees. The fullRange value is 270.0 degrees and the offset is -135.0 since the halfway point after scaling is 135 degrees.

This will calculate the result from the fullRange times the fraction of the supply voltage, plus the offset.

Parameters

- **channel** – The channel number on the roboRIO to represent. 0-3 are on-board 4-7 are on the MXP port.
- **fullRange** – The angular value (in desired units) representing the full 0-5V range of the input.
- **offset** – The angular value (in desired units) representing the angular output at 0V.

2. __init__(self: wpilib._wpilib.AnalogPotentiometer, input: wpilib._wpilib.AnalogInput, fullRange: float = 1.0, offset: float = 0.0) -> None

Construct an Analog Potentiometer object from an existing Analog Input pointer.

Use the fullRange and offset values so that the output produces meaningful values. I.E: you have a 270 degree potentiometer and you want the output to be degrees with the halfway point as 0 degrees. The fullRange value is 270.0 degrees and the offset is -135.0 since the halfway point after scaling is 135 degrees.

This will calculate the result from the fullRange times the fraction of the supply voltage, plus the offset.

Parameters

- **channel** – The existing Analog Input pointer
- **fullRange** – The angular value (in desired units) representing the full 0-5V range of the input.
- **offset** – The angular value (in desired units) representing the angular output at 0V.

get() → float
Get the current reading of the potentiomer.

Returns The current position of the potentiometer (in the units used for fullRange and offset).

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

pidGet() → float
Implement the PIDSource interface.

Returns The current reading.

1.1.13 AnalogTrigger

class wpilib.**AnalogTrigger**(**args*, ***kwargs*)
Bases: *wpilib.ErrorBase*, *wpilib.Sendable*

Overloaded function.

1. __init__(self: wpilib._wpilib.AnalogTrigger, channel: int) -> None

Constructor for an analog trigger given a channel number.

Parameters **channel** – The channel number on the roboRIO to represent. 0-3 are on-board 4-7 are on the MXP port.

2. __init__(self: wpilib._wpilib.AnalogTrigger, channel: wpilib._wpilib.AnalogInput) -> None

Construct an analog trigger given an analog input.

This should be used in the case of sharing an analog channel between the trigger and an analog input object.

Parameters **channel** – The pointer to the existing AnalogInput object

3. __init__(self: wpilib._wpilib.AnalogTrigger, dutyCycle: frc::DutyCycle) -> None

Construct an analog trigger given a duty cycle input.

Parameters **channel** – The pointer to the existing DutyCycle object

createOutput(*type: wpilib._wpilib.AnalogTriggerType*) → frc::AnalogTriggerOutput
Creates an AnalogTriggerOutput object.

Gets an output object that can be used for routing. Caller is responsible for deleting the AnalogTriggerOutput object.

Parameters **type** – An enum of the type of output object to create.

Returns A pointer to a new AnalogTriggerOutput object.

getInWindow() → bool
Return the InWindow output of the analog trigger.

True if the analog input is between the upper and lower limits.

Returns True if the analog input is between the upper and lower limits.

getIndex() → int
Return the index of the analog trigger.

This is the FPGA index of this analog trigger instance.

Returns The index of the analog trigger.

getTriggerState() → bool
Return the TriggerState output of the analog trigger.

True if above upper limit. False if below lower limit. If in Hysteresis, maintain previous state.

Returns True if above upper limit. False if below lower limit. If in Hysteresis, maintain previous state.

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

setAveraged(*useAveragedValue: bool*) → None
Configure the analog trigger to use the averaged vs. raw values.

If the value is true, then the averaged value is selected for the analog trigger, otherwise the immediate value is used.

Parameters **useAveragedValue** – If true, use the Averaged value, otherwise use the instantaneous reading

setFiltered(*useFilteredValue: bool*) → None
Configure the analog trigger to use a filtered value.

The analog trigger will operate with a 3 point average rejection filter. This is designed to help with 360 degree pot applications for the period where the pot crosses through zero.

Parameters **useFilteredValue** – If true, use the 3 point rejection filter, otherwise use the unfiltered value

setLimitsDutyCycle(*lower: float*, *upper: float*) → None
Set the upper and lower duty cycle limits of the analog trigger.

The limits are given as floating point values between 0 and 1.

Parameters

- **lower** – The lower limit of the trigger in percentage.
- **upper** – The upper limit of the trigger in percentage.

setLimitsRaw(*lower: int*, *upper: int*) → None
Set the upper and lower limits of the analog trigger.

The limits are given in ADC codes. If oversampling is used, the units must be scaled appropriately.

Parameters

- **lower** – The lower limit of the trigger in ADC codes (12-bit values).
- **upper** – The upper limit of the trigger in ADC codes (12-bit values).

setLimitsVoltage(*lower: float*, *upper: float*) → None
Set the upper and lower limits of the analog trigger.

The limits are given as floating point voltage values.

Parameters

- **lower** – The lower limit of the trigger in Volts.
- **upper** – The upper limit of the trigger in Volts.

1.1.14 AnalogTriggerOutput

class wpilib.**AnalogTriggerOutput**(*trigger: wpilib._wpilib.AnalogTrigger*, *outputType: wpilib._wpilib.AnalogTriggerType*) → None
Bases: *wpilib.DigitalSource*, *wpilib.Sendable*

Class to represent a specific output from an analog trigger.

This class is used to get the current output value and also as a DigitalSource to provide routing of an output to digital subsystems on the FPGA such as Counter, Encoder, and Interrupt.

The TriggerState output indicates the primary output value of the trigger. If the analog signal is less than the lower limit, the output is false. If the analog value is greater than the upper limit, then the output is true. If the analog value is in between, then the trigger output state maintains its most recent value.

The InWindow output indicates whether or not the analog signal is inside the range defined by the limits.

The RisingPulse and FallingPulse outputs detect an instantaneous transition from above the upper limit to below the lower limit, and vise versa. These pulses represent a rollover condition of a sensor and can be routed to an up / down counter or to interrupts. Because the outputs generate a pulse, they cannot be read directly. To help ensure that a rollover condition is not missed, there is an average rejection filter available that operates on the upper 8 bits of a 12 bit number and selects the nearest outlyer of 3 samples. This will reject a sample that is (due to averaging or sampling) errantly between the two limits. This filter will fail if more than one sample in a row is errantly in between the two limits. You may see this problem if attempting to use this feature with a mechanical rollover sensor, such as a 360 degree no-stop potentiometer without signal conditioning, because the rollover transition is not sharp / clean enough. Using the averaging engine may help with this, but rotational speeds of the sensor will then be limited.

Create an object that represents one of the four outputs from an analog trigger.

Because this class derives from DigitalSource, it can be passed into routing functions for Counter, Encoder, etc.

Parameters

- **trigger** – A pointer to the trigger for which this is an output.
- **outputType** – An enum that specifies the output on the trigger to represent.

get() → bool
Get the state of the analog trigger output.

Returns The state of the analog trigger output.

getAnalogTriggerTypeForRouting() → wpilib._wpilib.AnalogTriggerType

Returns The type of analog trigger output to be used.

getChannel() → int

Returns The channel of the source.

getPortHandleForRouting() → int

Returns The HAL Handle to the specified source.

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

isAnalogTrigger() → bool
Is source an AnalogTrigger

1.1.15 AnalogTriggerType

class wpilib.**AnalogTriggerType**(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kInWindow

kState

kRisingPulse

kFallingPulse

kFallingPulse = AnalogTriggerType.kFallingPulse

kInWindow = AnalogTriggerType.kInWindow

kRisingPulse = AnalogTriggerType.kRisingPulse

kState = AnalogTriggerType.kState

name
(self: handle) -> str

1.1.16 BuiltInAccelerometer

class wpilib.**BuiltInAccelerometer**(*range: wpilib.interfaces._interfaces.Accelerometer.Range = Range.kRange_8G*) → None
Bases: *wpilib.ErrorBase*, *wpilib.interfaces.Accelerometer*, *wpilib.Sendable*

Built-in accelerometer.

This class allows access to the roboRIO's internal accelerometer.

Constructor.

Parameters **range** – The range the accelerometer will measure

getX() → float

Returns The acceleration of the roboRIO along the X axis in g-forces

getY() → float

Returns The acceleration of the roboRIO along the Y axis in g-forces

getZ() → float

Returns The acceleration of the roboRIO along the Z axis in g-forces

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

setRange(*range: wpilib.interfaces._interfaces.Accelerometer.Range*) → None
Set the measuring range of the accelerometer.

Parameters **range** – The maximum acceleration, positive or negative, that the accelerometer will measure. Not all accelerometers support all ranges.

1.1.17 CAN

class wpilib.**CAN**(**args*, ***kwargs*)

Bases: *wpilib.ErrorBase*

High level class for interfacing with CAN devices conforming to the standard CAN spec.

No packets that can be sent gets blocked by the RoboRIO, so all methods work identically in all robot modes.

All methods are thread save, however the buffer objects passed in by the user need to not be modified for the duration of their calls.

Overloaded function.

1. __init__(self: wpilib._wpilib.CAN, deviceId: int) -> None

Create a new CAN communication interface with the specific device ID. This uses the team manufacturer and device types. The device ID is 6 bits (0-63)

Parameters **deviceId** – The device id

2. __init__(self: wpilib._wpilib.CAN, deviceId: int, deviceManufacturer: int, deviceType: int) -> None

Create a new CAN communication interface with a specific device ID, manufacturer and device type. The device ID is 6 bits, the manufacturer is 8 bits, and the device type is 5 bits.

Parameters

- **deviceId** – The device ID
- **deviceManufacturer** – The device manufacturer
- **deviceType** – The device type

kTeamDeviceType = CANDeviceType.kMiscellaneous

kTeamManufacturer = CANManufacturer.kTeamUse

readPacketLatest(*apiId: int*, *data: wpilib._wpilib.CANData*) → bool

Read a CAN packet. The will continuously return the last packet received, without accounting for packet age.

Parameters

- **apiId** – The API ID to read.
- **data** – Storage for the received data.

Returns True if the data is valid, otherwise false.

readPacketNew(*apiId: int*, *data: wpilib._wpilib.CANData*) → bool

Read a new CAN packet. This will only return properly once per packet received. Multiple calls without receiving another packet will return false.

Parameters

- **apiId** – The API ID to read.
- **data** – Storage for the received data.

Returns True if the data is valid, otherwise false.

readPacketTimeout(*apiId: int*, *timeoutMs: int*, *data: wpilib._wpilib.CANData*) → bool
Read a CAN packet. The will return the last packet received until the packet is older then the requested timeout. Then it will return false.

Parameters

- **apiId** – The API ID to read.
- **timeoutMs** – The timeout time for the packet
- **data** – Storage for the received data.

Returns True if the data is valid, otherwise false.

stopPacketRepeating(*apiId: int*) → None
Stop a repeating packet with a specific ID. This ID is 10 bits.

Parameters **apiId** – The API ID to stop repeating

writePacket(*data: buffer*, *apiId: int*) → None
Write a packet to the CAN device with a specific ID. This ID is 10 bits.

Parameters

- **data** – The data to write (8 bytes max)
- **length** – The data length to write
- **apiId** – The API ID to write.

writePacketRepeating(*data: buffer*, *apiId: int*, *repeatMs: int*) → None
Write a repeating packet to the CAN device with a specific ID. This ID is 10 bits. The RoboRIO will automatically repeat the packet at the specified interval

Parameters

- **data** – The data to write (8 bytes max)
- **length** – The data length to write
- **apiId** – The API ID to write.
- **repeatMs** – The period to repeat the packet at.

writeRTRFrame(*length: int*, *apiId: int*) → None
Write an RTR frame to the CAN device with a specific ID. This ID is 10 bits. The length by spec must match what is returned by the responding device

Parameters

- **length** – The length to request (0 to 8)
- **apiId** – The API ID to write.

1.1.18 CANData

class wpilib.**CANData**() → None
Bases: pybind11_builtins.pybind11_object

data

length

timestamp

1.1.19 CANStatus

class wpilib.**CANStatus**() → None
Bases: pybind11_builtins.pybind11_object

busOffCount

percentBusUtilization

receiveErrorCount

transmitErrorCount

txFullCount

1.1.20 CameraServer

class wpilib.**CameraServer**
Bases: object

Provides a way to launch an out of process cscore-based camera service instance, for streaming or for image processing.

Note: This does not correspond directly to the wpilib CameraServer object; that can be found as `cscore.CameraServer`. However, you should not use cscore directly from your robot code, see the documentation for details

---

classmethod is_alive()

Return type bool

Returns True if the CameraServer is still alive

classmethod launch(*vision_py=None*)
Launches the CameraServer process in autocapture mode or using a user-specified python script

Parameters **vision_py** (Optional[str]) – If specified, this is the relative path to a filename with a function in it

Example usage:

```
wpilib.CameraServer.launch("vision.py:main")
```

Warning: You must have robotpy-cscore installed, or this function will fail without returning an error (you will see an error in the console).

Return type None

1.1.21 Color

class wpilib.**Color**(**args*, ***kwargs*)
Bases: pybind11_builtins.pybind11_object

Represents colors that can be used with Addressable LEDs.

Limited to 12 bits of precision.

Overloaded function.

1. __init__(self: wpilib._wpilib.Color) -> None
2. __init__(self: wpilib._wpilib.Color, red: float, green: float, blue: float) -> None

Constructs a Color.

Parameters

- **red** – Red value (0-1)
- **green** – Green value (0-1)
- **blue** – Blue value (0-1)

blue

green

kAliceBlue = Color(red=0.941284, green=0.972534, blue=1.000000)

kAntiqueWhite = Color(red=0.980347, green=0.921509, blue=0.843140)

kAqua = Color(red=0.000122, green=1.000000, blue=1.000000)

kAquamarine = Color(red=0.497925, green=1.000000, blue=0.831421)

kAzure = Color(red=0.941284, green=1.000000, blue=1.000000)

kBeige = Color(red=0.960815, green=0.960815, blue=0.862671)

kBisque = Color(red=1.000000, green=0.894165, blue=0.768677)

kBlack = Color(red=0.000122, green=0.000122, blue=0.000122)

kBlanchedAlmond = Color(red=1.000000, green=0.921509, blue=0.803833)

kBlue = Color(red=0.000122, green=0.000122, blue=1.000000)

kBlueViolet = Color(red=0.541138, green=0.168579, blue=0.886353)

kBrown = Color(red=0.647095, green=0.164673, blue=0.164673)

kBurlywood = Color(red=0.870483, green=0.721558, blue=0.529419)

kCadetBlue = Color(red=0.372437, green=0.619507, blue=0.627563)

kChartreuse = Color(red=0.497925, green=1.000000, blue=0.000122)

kChocolate = Color(red=0.823608, green=0.411743, blue=0.117554)

kCoral = Color(red=1.000000, green=0.497925, blue=0.313843)

kCornflowerBlue = Color(red=0.392212, green=0.584351, blue=0.929321)

kCornsilk = Color(red=1.000000, green=0.972534, blue=0.862671)

kCrimson = Color(red=0.862671, green=0.078491, blue=0.235229)

kCyan = Color(red=0.000122, green=1.000000, blue=1.000000)

kDarkBlue = Color(red=0.000122, green=0.000122, blue=0.545044)

kDarkCyan = Color(red=0.000122, green=0.545044, blue=0.545044)

kDarkGoldenrod = Color(red=0.721558, green=0.525513, blue=0.043091)

kDarkGray = Color(red=0.662720, green=0.662720, blue=0.662720)

kDarkGreen = Color(red=0.000122, green=0.392212, blue=0.000122)

kDarkKhaki = Color(red=0.741089, green=0.717651, blue=0.419556)

kDarkMagenta = Color(red=0.545044, green=0.000122, blue=0.545044)

kDarkOliveGreen = Color(red=0.333374, green=0.419556, blue=0.184204)

kDarkOrange = Color(red=1.000000, green=0.548950, blue=0.000122)

kDarkOrchid = Color(red=0.599976, green=0.196167, blue=0.799927)

kDarkRed = Color(red=0.545044, green=0.000122, blue=0.000122)

kDarkSalmon = Color(red=0.913696, green=0.588257, blue=0.478394)

kDarkSeaGreen = Color(red=0.560669, green=0.737183, blue=0.560669)

kDarkSlateBlue = Color(red=0.282349, green=0.239136, blue=0.545044)

kDarkSlateGray = Color(red=0.184204, green=0.309692, blue=0.309692)

kDarkTurquoise = Color(red=0.000122, green=0.807739, blue=0.819702)

kDarkViolet = Color(red=0.580444, green=0.000122, blue=0.827515)

kDeepPink = Color(red=1.000000, green=0.078491, blue=0.576538)

kDeepSkyBlue = Color(red=0.000122, green=0.748901, blue=1.000000)

kDenim = Color(red=0.082397, green=0.376587, blue=0.741089)

kDimGray = Color(red=0.411743, green=0.411743, blue=0.411743)

kDodgerBlue = Color(red=0.117554, green=0.564819, blue=1.000000)

kFirebrick = Color(red=0.698120, green=0.133423, blue=0.133423)

kFirstBlue = Color(red=0.000122, green=0.400024, blue=0.702026)

kFirstRed = Color(red=0.929321, green=0.109741, blue=0.141235)

kFloralWhite = Color(red=1.000000, green=0.980347, blue=0.941284)

kForestGreen = Color(red=0.133423, green=0.545044, blue=0.133423)

kFuchsia = Color(red=1.000000, green=0.000122, blue=1.000000)

kGainsboro = Color(red=0.862671, green=0.862671, blue=0.862671)

kGhostWhite = Color(red=0.972534, green=0.972534, blue=1.000000)

kGold = Color(red=1.000000, green=0.843140, blue=0.000122)

kGoldenrod = Color(red=0.854858, green=0.647095, blue=0.125610)

kGray = Color(red=0.502075, green=0.502075, blue=0.502075)

kGreen = Color(red=0.000122, green=0.502075, blue=0.000122)

kGreenYellow = Color(red=0.678345, green=1.000000, blue=0.184204)

kHoneydew = Color(red=0.941284, green=1.000000, blue=0.941284)

kHotPink = Color(red=1.000000, green=0.411743, blue=0.705933)

kIndianRed = Color(red=0.803833, green=0.360718, blue=0.360718)

kIndigo = Color(red=0.294067, green=0.000122, blue=0.509888)

kIvory = Color(red=1.000000, green=1.000000, blue=0.941284)

kKhaki = Color(red=0.941284, green=0.901978, blue=0.548950)

```
kLavender = Color(red=0.901978, green=0.901978, blue=0.980347)
kLavenderBlush = Color(red=1.000000, green=0.941284, blue=0.960815)
kLawnGreen = Color(red=0.486206, green=0.988159, blue=0.000122)
kLemonChiffon = Color(red=1.000000, green=0.980347, blue=0.803833)
kLightBlue = Color(red=0.678345, green=0.847046, blue=0.901978)
kLightCoral = Color(red=0.941284, green=0.502075, blue=0.502075)
kLightCyan = Color(red=0.878540, green=1.000000, blue=1.000000)
kLightGoldenrodYellow = Color(red=0.980347, green=0.980347, blue=0.823608)
kLightGray = Color(red=0.827515, green=0.827515, blue=0.827515)
kLightGreen = Color(red=0.564819, green=0.933228, blue=0.564819)
kLightPink = Color(red=1.000000, green=0.713745, blue=0.756958)
kLightSalmon = Color(red=1.000000, green=0.627563, blue=0.478394)
kLightSeaGreen = Color(red=0.125610, green=0.698120, blue=0.666626)
kLightSkyBlue = Color(red=0.529419, green=0.807739, blue=0.980347)
kLightSlateGray = Color(red=0.466675, green=0.533325, blue=0.599976)
kLightSteelBlue = Color(red=0.690308, green=0.768677, blue=0.870483)
kLightYellow = Color(red=1.000000, green=1.000000, blue=0.878540)
kLime = Color(red=0.000122, green=1.000000, blue=0.000122)
kLimeGreen = Color(red=0.196167, green=0.803833, blue=0.196167)
kLinen = Color(red=0.980347, green=0.941284, blue=0.901978)
kMagenta = Color(red=1.000000, green=0.000122, blue=1.000000)
kMaroon = Color(red=0.502075, green=0.000122, blue=0.000122)
kMediumAquamarine = Color(red=0.400024, green=0.803833, blue=0.666626)
kMediumBlue = Color(red=0.000122, green=0.000122, blue=0.803833)
kMediumOrchid = Color(red=0.729370, green=0.333374, blue=0.827515)
kMediumPurple = Color(red=0.576538, green=0.439331, blue=0.858765)
kMediumSeaGreen = Color(red=0.235229, green=0.702026, blue=0.443237)
kMediumSlateBlue = Color(red=0.482300, green=0.407837, blue=0.933228)
kMediumSpringGreen = Color(red=0.000122, green=0.980347, blue=0.603882)
kMediumTurquoise = Color(red=0.282349, green=0.819702, blue=0.799927)
kMediumVioletRed = Color(red=0.780396, green=0.082397, blue=0.521606)
kMidnightBlue = Color(red=0.098022, green=0.098022, blue=0.439331)
kMintcream = Color(red=0.960815, green=1.000000, blue=0.980347)
kMistyRose = Color(red=1.000000, green=0.894165, blue=0.882446)
kMoccasin = Color(red=1.000000, green=0.894165, blue=0.709839)
kNavajoWhite = Color(red=1.000000, green=0.870483, blue=0.678345)
```

kNavy = Color(red=0.000122, green=0.000122, blue=0.502075)

kOldLace = Color(red=0.992065, green=0.960815, blue=0.901978)

kOlive = Color(red=0.502075, green=0.502075, blue=0.000122)

kOliveDrab = Color(red=0.419556, green=0.556763, blue=0.137329)

kOrange = Color(red=1.000000, green=0.647095, blue=0.000122)

kOrangeRed = Color(red=1.000000, green=0.270630, blue=0.000122)

kOrchid = Color(red=0.854858, green=0.439331, blue=0.839233)

kPaleGoldenrod = Color(red=0.933228, green=0.909790, blue=0.666626)

kPaleGreen = Color(red=0.596069, green=0.984253, blue=0.596069)

kPaleTurquoise = Color(red=0.686157, green=0.933228, blue=0.933228)

kPaleVioletRed = Color(red=0.858765, green=0.439331, blue=0.576538)

kPapayaWhip = Color(red=1.000000, green=0.937134, blue=0.835327)

kPeachPuff = Color(red=1.000000, green=0.854858, blue=0.725464)

kPeru = Color(red=0.803833, green=0.521606, blue=0.246948)

kPink = Color(red=1.000000, green=0.753052, blue=0.796021)

kPlum = Color(red=0.866577, green=0.627563, blue=0.866577)

kPowderBlue = Color(red=0.690308, green=0.878540, blue=0.901978)

kPurple = Color(red=0.502075, green=0.000122, blue=0.502075)

kRed = Color(red=1.000000, green=0.000122, blue=0.000122)

kRosyBrown = Color(red=0.737183, green=0.560669, blue=0.560669)

kRoyalBlue = Color(red=0.255005, green=0.411743, blue=0.882446)

kSaddleBrown = Color(red=0.545044, green=0.270630, blue=0.074585)

kSalmon = Color(red=0.980347, green=0.502075, blue=0.447144)

kSandyBrown = Color(red=0.956909, green=0.643188, blue=0.376587)

kSeaGreen = Color(red=0.180298, green=0.545044, blue=0.341187)

kSeashell = Color(red=1.000000, green=0.960815, blue=0.933228)

kSienna = Color(red=0.627563, green=0.321655, blue=0.176392)

kSilver = Color(red=0.753052, green=0.753052, blue=0.753052)

kSkyBlue = Color(red=0.529419, green=0.807739, blue=0.921509)

kSlateBlue = Color(red=0.415649, green=0.352905, blue=0.803833)

kSlateGray = Color(red=0.439331, green=0.502075, blue=0.564819)

kSnow = Color(red=1.000000, green=0.980347, blue=0.980347)

kSpringGreen = Color(red=0.000122, green=1.000000, blue=0.497925)

kSteelBlue = Color(red=0.274536, green=0.509888, blue=0.705933)

kTan = Color(red=0.823608, green=0.705933, blue=0.548950)

kTeal = Color(red=0.000122, green=0.502075, blue=0.502075)

kThistle = Color(red=0.847046, green=0.748901, blue=0.847046)

kTomato = Color(red=1.000000, green=0.388306, blue=0.278442)

kTurquoise = Color(red=0.251099, green=0.878540, blue=0.815796)

kViolet = Color(red=0.933228, green=0.509888, blue=0.933228)

kWheat = Color(red=0.960815, green=0.870483, blue=0.702026)

kWhite = Color(red=1.000000, green=1.000000, blue=1.000000)

kWhiteSmoke = Color(red=0.960815, green=0.960815, blue=0.960815)

kYellow = Color(red=1.000000, green=1.000000, blue=0.000122)

kYellowGreen = Color(red=0.603882, green=0.803833, blue=0.196167)

red

1.1.22 Color8Bit

class wpilib.**Color8Bit**(**args*, ***kwargs*)

Bases: pybind11_builtins.pybind11_object

Represents colors that can be used with Addressable LEDs.

Overloaded function.

1. __init__(self: wpilib._wpilib.Color8Bit) -> None
2. __init__(self: wpilib._wpilib.Color8Bit, red: int, green: int, blue: int) -> None

Constructs a Color8Bit.

Parameters

- **red** – Red value (0-255)
- **green** – Green value (0-255)
- **blue** – Blue value (0-255)

3. __init__(self: wpilib._wpilib.Color8Bit, color: wpilib._wpilib.Color) -> None

Constructs a Color8Bit from a Color.

Parameters **color** – The color

blue

green

red

1.1.23 Compressor

class wpilib.**Compressor**(*pcmID: int = 0*) → None

Bases: *wpilib.ErrorBase*, *wpilib.Sendable*

Class for operating a compressor connected to a %PCM (Pneumatic Control Module).

The PCM will automatically run in closed loop mode by default whenever a Solenoid object is created. For most cases, a Compressor object does not need to be instantiated or used in a robot program. This class is only

required in cases where the robot program needs a more detailed status of the compressor or to enable/disable closed loop control.

Note: you cannot operate the compressor directly from this class as doing so would circumvent the safety provided by using the pressure switch and closed loop control. You can only turn off closed loop control, thereby stopping the compressor from operating.

Constructor. The default PCM ID is 0.

Parameters **module** – The PCM ID to use (0-62)

clearAllPCMStickyFaults() → None
Clear ALL sticky faults inside PCM that Compressor is wired to.

If a sticky fault is set, then it will be persistently cleared. Compressor drive maybe momentarily disable while flags are being cleared. Care should be taken to not call this too frequently, otherwise normal compressor functionality may be prevented.

If no sticky faults are set then this call will have no effect.

enabled() → bool
Check if compressor output is active.

Returns true if the compressor is on

getClosedLoopControl() → bool
Returns true if the compressor will automatically turn on when the pressure is low.

Returns True if closed loop control of the compressor is enabled. False if disabled.

getCompressorCurrent() → float
Query how much current the compressor is drawing.

Returns The current through the compressor, in amps

getCompressorCurrentTooHighFault() → bool
Query if the compressor output has been disabled due to high current draw.

Returns true if PCM is in fault state : Compressor Drive is disabled due to compressor current being too high.

getCompressorCurrentTooHighStickyFault() → bool
Query if the compressor output has been disabled due to high current draw (sticky).

A sticky fault will not clear on device reboot, it must be cleared through code or the webdash.

Returns true if PCM sticky fault is set : Compressor Drive is disabled due to compressor current being too high.

getCompressorNotConnectedFault() → bool
Query if the compressor output does not appear to be wired.

Returns true if PCM is in fault state : Compressor does not appear to be wired, i.e. compressor is not drawing enough current.

getCompressorNotConnectedStickyFault() → bool
Query if the compressor output does not appear to be wired (sticky).

A sticky fault will not clear on device reboot, it must be cleared through code or the webdash.

Returns true if PCM sticky fault is set : Compressor does not appear to be wired, i.e. compressor is not drawing enough current.

getCompressorShortedFault() → bool
Query if the compressor output has been disabled due to a short circuit.

Returns true if PCM is in fault state : Compressor output appears to be shorted.

getCompressorShortedStickyFault() → bool
Query if the compressor output has been disabled due to a short circuit (sticky).

A sticky fault will not clear on device reboot, it must be cleared through code or the webdash.

Returns true if PCM sticky fault is set : Compressor output appears to be shorted.

getPressureSwitchValue() → bool
Check if the pressure switch is triggered.

Returns true if pressure is low

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

setClosedLoopControl(*on: bool*) → None
Enables or disables automatically turning the compressor on when the pressure is low.

Parameters **on** – Set to true to enable closed loop control of the compressor. False to disable.

start() → None
Starts closed-loop control. Note that closed loop control is enabled by default.

stop() → None
Stops closed-loop control. Note that closed loop control is enabled by default.

1.1.24 Counter

class wpilib.**Counter**(**args*, ***kwargs*)
Bases: *wpilib.ErrorBase*, *wpilib.interfaces.CounterBase*, *wpilib.Sendable*

Class for counting the number of ticks on a digital input channel.

This is a general purpose class for counting repetitive events. It can return the number of counts, the period of the most recent cycle, and detect when the signal being counted has stopped by supplying a maximum cycle time.

All counters will immediately start counting - Reset() them if you need them to be zeroed before use.

Overloaded function.

1. __init__(self: wpilib._wpilib.Counter, mode: wpilib._wpilib.Counter.Mode = Mode.kTwoPulse) -> None

Create an instance of a counter where no sources are selected.

They all must be selected by calling functions to specify the upsource and the downsource independently.

This creates a ChipObject counter and initializes status variables appropriately.

The counter will start counting immediately.

Parameters **mode** – The counter mode

2. __init__(self: wpilib._wpilib.Counter, channel: int) -> None

Create an instance of a Counter object.

Create an up-Counter instance given a channel.

The counter will start counting immediately.

Parameters **channel** – The DIO channel to use as the up source. 0-9 are on-board, 10-25 are on the MXP

3. __init__(self: wpilib._wpilib.Counter, source: frc::DigitalSource) -> None

Create an instance of a counter from a Digital Source (such as a Digital Input).

This is used if an existing digital input is to be shared by multiple other objects such as encoders or if the Digital Source is not a Digital Input channel (such as an Analog Trigger).

The counter will start counting immediately.

Parameters **source** – A pointer to the existing DigitalSource object. It will be set as the Up Source.

4. __init__(self: wpilib._wpilib.Counter, trigger: wpilib._wpilib.AnalogTrigger) -> None

Create an instance of a Counter object.

Create an instance of a simple up-Counter given an analog trigger. Use the trigger state output from the analog trigger.

The counter will start counting immediately.

Parameters **trigger** – The reference to the existing AnalogTrigger object.

5. __init__(self: wpilib._wpilib.Counter, encodingType: wpilib.interfaces._interfaces.CounterBase.EncodingType, upSource: frc::DigitalSource, downSource: frc::DigitalSource, inverted: bool) -> None

Create an instance of a Counter object.

Creates a full up-down counter given two Digital Sources.

Parameters

- **encodingType** – The quadrature decoding mode (1x or 2x)
- **upSource** – The pointer to the DigitalSource to set as the up source
- **downSource** – The pointer to the DigitalSource to set as the down source
- **inverted** – True to invert the output (reverse the direction)

class Mode(*arg0: int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kTwoPulse

kSemiperiod

kPulseLength

kExternalDirection

kExternalDirection = Mode.kExternalDirection

kPulseLength = Mode.kPulseLength

kSemiperiod = Mode.kSemiperiod

kTwoPulse = Mode.kTwoPulse

name

(self: handle) -> str

clearDownSource() → None

Disable the down counting source to the counter.

clearUpSource() → None
Disable the up counting source to the counter.

get() → int
Read the current counter value.

Read the value at this instant. It may still be running, so it reflects the current value. Next time it is read, it might have a different value.

getDirection() → bool
The last direction the counter value changed.

Returns The last direction the counter value changed.

getFPGAIndex() → int

getPeriod() → float
Get the Period of the most recent count.

Returns the time interval of the most recent count. This can be used for velocity calculations to determine shaft speed.

Returns The period between the last two pulses in units of seconds.

getSamplesToAverage() → int
Get the Samples to Average which specifies the number of samples of the timer to average when calculating the period.

Perform averaging to account for mechanical imperfections or as oversampling to increase resolution.

Returns The number of samples being averaged (from 1 to 127)

getStopped() → bool
Determine if the clock is stopped.

Determine if the clocked input is stopped based on the MaxPeriod value set using the SetMaxPeriod method. If the clock exceeds the MaxPeriod, then the device (and counter) are assumed to be stopped and it returns true.

Returns Returns true if the most recent counter period exceeds the MaxPeriod value set by SetMaxPeriod.

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

reset() → None
Reset the Counter to zero.

Set the counter value to zero. This doesn't effect the running state of the counter, just sets the current value to zero.

setDownSource(**args*, ***kwargs*)
Overloaded function.

1. setDownSource(self: wpilib._wpilib.Counter, channel: int) -> None

Set the down counting source to be a digital input channel.

Parameters **channel** – The DIO channel to use as the up source. 0-9 are on-board, 10-25 are on the MXP

2. setDownSource(self: wpilib._wpilib.Counter, analogTrigger: wpilib._wpilib.AnalogTrigger, triggerType: wpilib._wpilib.AnalogTriggerType) -> None

Set the down counting source to be an analog trigger.

Parameters

- **analogTrigger** – The analog trigger object that is used for the Down Source
- **triggerType** – The analog trigger output that will trigger the counter.

3. setDownSource(self: wpilib._wpilib.Counter, source: frc::DigitalSource) -> None

setDownSourceEdge(*risingEdge: bool*, *fallingEdge: bool*) → None
Set the edge sensitivity on a down counting source.

Set the down source to either detect rising edges or falling edges.

Parameters

- **risingEdge** – True to trigger on rising edges
- **fallingEdge** – True to trigger on falling edges

setExternalDirectionMode() → None
Set external direction mode on this counter.

Counts are sourced on the Up counter input. The Down counter input represents the direction to count.

setMaxPeriod(*maxPeriod: float*) → None
Set the maximum period where the device is still considered "moving".

Sets the maximum period where the device is considered moving. This value is used to determine the "stopped" state of the counter using the GetStopped method.

Parameters **maxPeriod** – The maximum period where the counted device is considered moving in seconds.

setPulseLengthMode(*threshold: float*) → None
Configure the counter to count in up or down based on the length of the input pulse.

This mode is most useful for direction sensitive gear tooth sensors.

Parameters **threshold** – The pulse length beyond which the counter counts the opposite direction. Units are seconds.

setReverseDirection(*reverseDirection: bool*) → None
Set the Counter to return reversed sensing on the direction.

This allows counters to change the direction they are counting in the case of 1X and 2X quadrature encoding only. Any other counter mode isn't supported.

Parameters **reverseDirection** – true if the value counted should be negated.

setSamplesToAverage(*samplesToAverage: int*) → None
Set the Samples to Average which specifies the number of samples of the timer to average when calculating the period. Perform averaging to account for mechanical imperfections or as oversampling to increase resolution.

Parameters **samplesToAverage** – The number of samples to average from 1 to 127.

setSemiPeriodMode(*highSemiPeriod: bool*) → None
Set Semi-period mode on this counter.

Counts up on both rising and falling edges.

setUpDownCounterMode() → None
Set standard up / down counting mode on this counter.

Up and down counts are sourced independently from two inputs.

setUpSource(**args*, ***kwargs*)
Overloaded function.

1. setUpSource(self: wpilib._wpilib.Counter, channel: int) -> None

Set the upsource for the counter as a digital input channel.

Parameters **channel** – The DIO channel to use as the up source. 0-9 are on-board, 10-25 are on the MXP

2. setUpSource(self: wpilib._wpilib.Counter, analogTrigger: wpilib._wpilib.AnalogTrigger, triggerType: wpilib._wpilib.AnalogTriggerType) -> None

Set the up counting source to be an analog trigger.

Parameters

- **analogTrigger** – The analog trigger object that is used for the Up Source
- **triggerType** – The analog trigger output that will trigger the counter.

3. setUpSource(self: wpilib._wpilib.Counter, source: frc::DigitalSource) -> None

Set the source object that causes the counter to count up.

Set the up counting DigitalSource.

Parameters **source** – Pointer to the DigitalSource object to set as the up source

setUpSourceEdge(*risingEdge: bool*, *fallingEdge: bool*) → None
Set the edge sensitivity on an up counting source.

Set the up source to either detect rising edges or falling edges or both.

Parameters

- **risingEdge** – True to trigger on rising edges
- **fallingEdge** – True to trigger on falling edges

setUpdateWhenEmpty(*enabled: bool*) → None
Select whether you want to continue updating the event timer output when there are no samples captured.

The output of the event timer has a buffer of periods that are averaged and posted to a register on the FPGA. When the timer detects that the event source has stopped (based on the MaxPeriod) the buffer of samples to be averaged is emptied. If you enable the update when empty, you will be notified of the stopped source and the event time will report 0 samples. If you disable update when empty, the most recent average will remain on the output until a new sample is acquired. You will never see 0 samples output (except when there have been no events since an FPGA reset) and you will likely not see the stopped bit become true (since it is updated at the end of an average and there are no samples to average).

Parameters **enabled** – True to enable update when empty

1.1.25 DMC60

class wpilib.**DMC60**(*channel: int*) → None
Bases: *wpilib.PWMSpeedController*

Digilent DMC 60 Speed Controller.

Note that the DMC 60 uses the following bounds for PWM values. These values should work reasonably well for most controllers, but if users experience issues such as asymmetric behavior around the deadband or inability

to saturate the controller in either direction, calibration is recommended. The calibration procedure can be found in the DMC 60 User Manual available from Digilent.

- 2.004ms = full "forward"
- 1.520ms = the "high end" of the deadband range
- 1.500ms = center of the deadband range (off)
- 1.480ms = the "low end" of the deadband range
- 0.997ms = full "reverse"

Constructor for a Digilent DMC 60.

Parameters **channel** – The PWM channel that the DMC 60 is attached to. 0-9 are on-board, 10-19 are on the MXP port

1.1.26 DigitalGlitchFilter

class wpilib.**DigitalGlitchFilter**() → None
Bases: *wpilib.ErrorBase*, *wpilib.Sendable*

Class to enable glitch filtering on a set of digital inputs.

This class will manage adding and removing digital inputs from a FPGA glitch filter. The filter lets the user configure the time that an input must remain high or low before it is classified as high or low.

add(**args*, ***kwargs*)
Overloaded function.

1. add(self: wpilib._wpilib.DigitalGlitchFilter, input: frc::DigitalSource) -> None

Assigns the DigitalSource to this glitch filter.

Parameters **input** – The DigitalSource to add.

2. add(self: wpilib._wpilib.DigitalGlitchFilter, input: frc::Encoder) -> None

Assigns the Encoder to this glitch filter.

Parameters **input** – The Encoder to add.

3. add(self: wpilib._wpilib.DigitalGlitchFilter, input: wpilib._wpilib.Counter) -> None

Assigns the Counter to this glitch filter.

Parameters **input** – The Counter to add.

getPeriodCycles() → int
Gets the number of cycles that the input must not change state for.

Returns The number of cycles.

getPeriodNanoSeconds() → int
Gets the number of nanoseconds that the input must not change state for.

Returns The number of nanoseconds.

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

remove(**args*, ***kwargs*)
Overloaded function.

1. remove(self: wpilib._wpilib.DigitalGlitchFilter, input: frc::DigitalSource) -> None

Removes a digital input from this filter.

Removes the DigitalSource from this glitch filter and re-assigns it to the default filter.

Parameters **input** – The DigitalSource to remove.

2. remove(self: wpilib._wpilib.DigitalGlitchFilter, input: frc::Encoder) -> None

Removes an encoder from this filter.

Removes the Encoder from this glitch filter and re-assigns it to the default filter.

Parameters **input** – The Encoder to remove.

3. remove(self: wpilib._wpilib.DigitalGlitchFilter, input: wpilib._wpilib.Counter) -> None

Removes a counter from this filter.

Removes the Counter from this glitch filter and re-assigns it to the default filter.

Parameters **input** – The Counter to remove.

setPeriodCycles (*fpgaCycles: int*) → None
Sets the number of cycles that the input must not change state for.

Parameters **fpgaCycles** – The number of FPGA cycles.

setPeriodNanoSeconds (*nanoseconds: int*) → None
Sets the number of nanoseconds that the input must not change state for.

Parameters **nanoseconds** – The number of nanoseconds.

1.1.27 DigitalInput

class wpilib.**DigitalInput** (*channel: int*) → None
Bases: *wpilib.DigitalSource*, *wpilib.Sendable*

Class to read a digital input.

This class will read digital inputs and return the current value on the channel. Other devices such as encoders, gear tooth sensors, etc. that are implemented elsewhere will automatically allocate digital inputs and outputs as required. This class is only for devices like switches etc. that aren't implemented anywhere else.

Create an instance of a Digital Input class.

Creates a digital input given a channel.

Parameters **channel** – The DIO channel 0-9 are on-board, 10-25 are on the MXP port

get () → bool
Get the value from a digital input channel.

Retrieve the value of a single digital input channel from the FPGA.

getAnalogTriggerTypeForRouting () → wpilib._wpilib.AnalogTriggerType

Returns The type of analog trigger output to be used. 0 for Digitals

getChannel () → int

Returns The GPIO channel number that this object represents.

getPortHandleForRouting () → int

Returns The HAL Handle to the specified source.

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

isAnalogTrigger() → bool
Is source an AnalogTrigger

setSimDevice(*device: int*) → None
Indicates this input is used by a simulated device.

Parameters **device** – simulated device handle

1.1.28 DigitalOutput

class wpilib.**DigitalOutput**(*channel: int*) → None
Bases: *wpilib.DigitalSource*, *wpilib.Sendable*

Class to write to digital outputs.

Write values to the digital output channels. Other devices implemented elsewhere will allocate channels automatically so for those devices it shouldn't be done here.

Create an instance of a digital output.

Create a digital output given a channel.

Parameters **channel** – The digital channel 0-9 are on-board, 10-25 are on the MXP port

disablePWM() → None
Change this line from a PWM output back to a static Digital Output line.

Free up one of the 6 DO PWM generator resources that were in use.

enablePWM(*initialDutyCycle: float*) → None
Enable a PWM Output on this line.

Allocate one of the 6 DO PWM generator resources from this module.

Supply the initial duty-cycle to output so as to avoid a glitch when first starting.

The resolution of the duty cycle is 8-bit for low frequencies (1kHz or less) but is reduced the higher the frequency of the PWM signal is.

Parameters **initialDutyCycle** – The duty-cycle to start generating. [0..1]

get() → bool
Gets the value being output from the Digital Output.

Returns the state of the digital output.

getAnalogTriggerTypeForRouting() → wpilib._wpilib.AnalogTriggerType

Returns The type of analog trigger output to be used. 0 for Digitals

getChannel() → int

Returns The GPIO channel number that this object represents.

getPortHandleForRouting() → int

Returns The HAL Handle to the specified source.

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

isAnalogTrigger() → bool
Is source an AnalogTrigger

isPulsing() → bool
Determine if the pulse is still going.

Determine if a previously started pulse is still going.

pulse(*length: float*) → None
Output a single pulse on the digital output line.

Send a single pulse on the digital output line where the pulse duration is specified in seconds. Maximum pulse length is 0.0016 seconds.

Parameters **length** – The pulse length in seconds

set(*value: bool*) → None
Set the value of a digital output.

Set the value of a digital output to either one (true) or zero (false).

Parameters **value** – 1 (true) for high, 0 (false) for disabled

setPWMRate(*rate: float*) → None
Change the PWM frequency of the PWM output on a Digital Output line.

The valid range is from 0.6 Hz to 19 kHz. The frequency resolution is logarithmic.

There is only one PWM frequency for all digital channels.

Parameters **rate** – The frequency to output all digital output PWM signals.

setSimDevice(*device: int*) → None
Indicates this output is used by a simulated device.

Parameters **device** – simulated device handle

updateDutyCycle(*dutyCycle: float*) → None
Change the duty-cycle that is being generated on the line.

The resolution of the duty cycle is 8-bit for low frequencies (1kHz or less) but is reduced the higher the frequency of the PWM signal is.

Parameters **dutyCycle** – The duty-cycle to change to. [0..1]

1.1.29 DigitalSource

class wpilib.**DigitalSource**() → None
Bases: *wpilib.InterruptableSensorBase*

DigitalSource Interface.

The DigitalSource represents all the possible inputs for a counter or a quadrature encoder. The source may be either a digital input or an analog input. If the caller just provides a channel, then a digital input will be constructed and freed when finished for the source. The source can either be a digital input or analog trigger but not both.

getAnalogTriggerTypeForRouting() → wpilib._wpilib.AnalogTriggerType

getChannel() → int

getPortHandleForRouting() → int

isAnalogTrigger() → bool

1.1.30 DoubleSolenoid

class wpilib.**DoubleSolenoid**(**args*, ***kwargs*)
Bases: *wpilib.SolenoidBase*, *wpilib.Sendable*

DoubleSolenoid class for running 2 channels of high voltage Digital Output (PCM).

The DoubleSolenoid class is typically used for pneumatics solenoids that have two positions controlled by two separate channels.

Overloaded function.

1. __init__(self: wpilib._wpilib.DoubleSolenoid, forwardChannel: int, reverseChannel: int) -> None

Constructor.

Uses the default PCM ID of 0.

Parameters

- **forwardChannel** – The forward channel number on the PCM (0..7).
- **reverseChannel** – The reverse channel number on the PCM (0..7).

2. __init__(self: wpilib._wpilib.DoubleSolenoid, moduleNumber: int, forwardChannel: int, reverseChannel: int) -> None

Constructor.

Parameters

- **moduleNumber** – The CAN ID of the PCM.
- **forwardChannel** – The forward channel on the PCM to control (0..7).
- **reverseChannel** – The reverse channel on the PCM to control (0..7).

class Value(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kOff

kForward

kReverse

kForward = Value.kForward

kOff = Value.kOff

kReverse = Value.kReverse

name
(self: handle) -> str

get() → wpilib._wpilib.DoubleSolenoid.Value
Read the current value of the solenoid.

Returns The current value of the solenoid.

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

isFwdSolenoidBlackListed() → bool
Check if the forward solenoid is blacklisted.

If a solenoid is shorted, it is added to the blacklist and disabled until power cycle, or until faults are cleared.

@see ClearAllPCMStickyFaults()

Returns If solenoid is disabled due to short.

isRevSolenoidBlackListed() → bool
Check if the reverse solenoid is blacklisted.

If a solenoid is shorted, it is added to the blacklist and disabled until power cycle, or until faults are cleared.

@see ClearAllPCMStickyFaults()

Returns If solenoid is disabled due to short.

set(*value: wpilib._wpilib.DoubleSolenoid.Value*) → None
Set the value of a solenoid.

Parameters **value** – The value to set (Off, Forward or Reverse)

1.1.31 DriverStation

class wpilib.**DriverStation**
Bases: *wpilib.ErrorBase*

Provide access to the network communication data to / from the Driver Station.

class Alliance(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kRed

kBlue

kInvalid

kBlue = Alliance.kBlue

kInvalid = Alliance.kInvalid

kRed = Alliance.kRed

name
(self: handle) -> str

class MatchType(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kNone

kPractice

kQualification

kElimination

kElimination = MatchType.kElimination

kNone = MatchType.kNone

kPractice = MatchType.kPractice

kQualification = MatchType.kQualification

name
(self: handle) -> str

getAlliance() → wpilib._wpilib.DriverStation.Alliance
Return the alliance that the driver station says it is on.

This could return kRed or kBlue.

Returns The Alliance enum (kRed, kBlue or kInvalid)

getBatteryVoltage() → float
Read the battery voltage.

Returns The battery voltage in Volts.

getControlState() → Tuple[bool, bool, bool]
More efficient way to determine what state the robot is in.

Returns booleans representing enabled, isautonomous, istest

New in version 2019.2.1.

Note: This function only exists in RobotPy

---

getEventName() → str
Returns the name of the competition event provided by the FMS.

Returns A string containing the event name

getGameSpecificMessage() → str
Returns the game specific message provided by the FMS.

Returns A string containing the game specific message.

static getInstance() → wpilib._wpilib.DriverStation
Return a reference to the singleton DriverStation.

Returns Reference to the DS instance

getJoystickAxisType(*stick: int*, *axis: int*) → int
Returns the types of Axes on a given joystick port.

Parameters **stick** – The joystick port number and the target axis

Returns What type of axis the axis is reporting to be

getJoystickIsXbox(*stick: int*) → bool
Returns a boolean indicating if the controller is an xbox controller.

Parameters **stick** – The joystick port number

Returns A boolean that is true if the controller is an xbox controller.

getJoystickName(*stick: int*) → str
Returns the name of the joystick at the given port.

Parameters **stick** – The joystick port number

Returns The name of the joystick at the given port

getJoystickType(*stick: int*) → int
Returns the type of joystick at a given port.

Parameters **stick** – The joystick port number

Returns The HID type of joystick at the given port

getLocation() → int
Return the driver station location on the field.

This could return 1, 2, or 3.

Returns The location of the driver station (1-3, 0 for invalid)

getMatchNumber() → int
Returns the match number provided by the FMS.

Returns The number of the match

getMatchTime() → float
Return the approximate match time.

The FMS does not send an official match time to the robots, but does send an approximate match time. The value will count down the time remaining in the current period (auto or teleop).

Warning: This is not an official time (so it cannot be used to dispute ref calls or guarantee that a function will trigger before the match ends).

The Practice Match function of the DS approximates the behaviour seen on the field.

Returns Time remaining in current match period (auto or teleop)

getMatchType() → wpilib._wpilib.DriverStation.MatchType
Returns the type of match being played provided by the FMS.

Returns The match type enum (kNone, kPractice, kQualification, kElimination)

getReplayNumber() → int
Returns the number of times the current match has been replayed from the FMS.

Returns The number of replays

getStickAxis(*stick: int*, *axis: int*) → float
Get the value of the axis on a joystick.

This depends on the mapping of the joystick connected to the specified port.

Parameters

- **stick** – The joystick to read.
- **axis** – The analog axis value to read from the joystick.

Returns The value of the axis on the joystick.

getStickAxisCount(*stick: int*) → int
Returns the number of axes on a given joystick port.

Parameters **stick** – The joystick port number

Returns The number of axes on the indicated joystick

getStickButton(*stick: int*, *button: int*) → bool
The state of one joystick button. Button indexes begin at 1.

Parameters

- **stick** – The joystick to read.

- **button** – The button index, beginning at 1.

Returns The state of the joystick button.

getStickButtonCount(*stick: int*) → int
Returns the number of buttons on a given joystick port.

Parameters **stick** – The joystick port number

Returns The number of buttons on the indicated joystick

getStickButtonPressed(*stick: int*, *button: int*) → bool
Whether one joystick button was pressed since the last check. Button indexes begin at 1.

Parameters

- **stick** – The joystick to read.
- **button** – The button index, beginning at 1.

Returns Whether the joystick button was pressed since the last check.

getStickButtonReleased(*stick: int*, *button: int*) → bool
Whether one joystick button was released since the last check. Button indexes begin at 1.

Parameters

- **stick** – The joystick to read.
- **button** – The button index, beginning at 1.

Returns Whether the joystick button was released since the last check.

getStickButtons(*stick: int*) → int
The state of the buttons on the joystick.

Parameters **stick** – The joystick to read.

Returns The state of the buttons on the joystick.

getStickPOV(*stick: int*, *pov: int*) → int
Get the state of a POV on the joystick.

Returns the angle of the POV in degrees, or -1 if the POV is not pressed.

getStickPOVCount(*stick: int*) → int
Returns the number of POVs on a given joystick port.

Parameters **stick** – The joystick port number

Returns The number of POVs on the indicated joystick

inAutonomous(*entering: bool*) → None
Only to be used to tell the Driver Station what code you claim to be executing for diagnostic purposes only.

Parameters **entering** – If true, starting autonomous code; if false, leaving autonomous code.

inDisabled(*entering: bool*) → None
Only to be used to tell the Driver Station what code you claim to be executing for diagnostic purposes only.

Parameters **entering** – If true, starting disabled code; if false, leaving disabled code.

inOperatorControl(*entering: bool*) → None
Only to be used to tell the Driver Station what code you claim to be executing for diagnostic purposes only.

Parameters **entering** – If true, starting teleop code; if false, leaving teleop code.

inTest(*entering: bool*) → None
Only to be used to tell the Driver Station what code you claim to be executing for diagnostic purposes only.

Parameters **entering** – If true, starting test code; if false, leaving test code.

isAutonomous() → bool
Check if the DS is commanding autonomous mode.

Returns True if the robot is being commanded to be in autonomous mode

isAutonomousEnabled() → bool
Equivalent to calling `isAutonomous() and isEnabled()` but more efficient.

Returns True if the robot is in autonomous mode and is enabled, False otherwise.

New in version 2019.2.1.

Note: This function only exists in RobotPy

---

isDSAttached() → bool
Check if the DS is attached.

Returns True if the DS is connected to the robot

isDisabled() → bool
Check if the robot is disabled.

Returns True if the robot is explicitly disabled or the DS is not connected

isEStopped() → bool
Check if the robot is e-stopped.

Returns True if the robot is e-stopped

isEnabled() → bool
Check if the DS has enabled the robot.

Returns True if the robot is enabled and the DS is connected

isFMSAttached() → bool
Is the driver station attached to a Field Management System?

Returns True if the robot is competing on a field being controlled by a Field Management System

isNewControlData() → bool
Has a new control packet from the driver station arrived since the last time this function was called?

Warning: If you call this function from more than one place at the same time, you will not get the intended behavior.

Returns True if the control data has been updated since the last call.

isOperatorControl() → bool
Check if the DS is commanding teleop mode.

Returns True if the robot is being commanded to be in teleop mode

isOperatorControlEnabled() → bool
Equivalent to calling `isOperatorControl() and isEnabled()` but more efficient.

Returns True if the robot is in operator-controlled mode and is enabled, False otherwise.

New in version 2019.2.1.

Note: This function only exists in RobotPy

isTest() → bool
Check if the DS is commanding test mode.

Returns True if the robot is being commanded to be in test mode

kJoystickPorts = 6

static reportError(*error: str*, *printTrace: bool_*) → object
Report error to Driver Station, and also prints error to *sys.stderr*. Optionally appends stack trace to error message.

Parameters **printTrace** – If True, append stack trace to error string

The error is also printed to the program console.

static reportWarning(*error: str*, *printTrace: bool_*) → object
Report warning to Driver Station, and also prints error to *sys.stderr*. Optionally appends stack trace to error message.

Parameters **printTrace** – If True, append stack trace to error string

The error is also printed to the program console.

waitForData(**args*, ***kwargs*)
Overloaded function.

1. waitForData(self: wpilib._wpilib.DriverStation) -> None

Wait until a new packet comes from the driver station.

This blocks on a semaphore, so the waiting is efficient.

This is a good way to delay processing until there is new driver station data to act on.

2. waitForData(self: wpilib._wpilib.DriverStation, timeout: float) -> bool

Wait until a new packet comes from the driver station, or wait for a timeout.

If the timeout is less then or equal to 0, wait indefinitely.

Timeout is in milliseconds

This blocks on a semaphore, so the waiting is efficient.

This is a good way to delay processing until there is new driver station data to act on.

Parameters **timeout** – Timeout time in seconds

Returns true if new data, otherwise false

wakeupWaitForData() → None
Forces WaitForData() to return immediately.

1.1.32 DutyCycle

class wpilib.**DutyCycle**(*source: frc::DigitalSource*) → None
Bases: *wpilib.ErrorBase*, *wpilib.Sendable*

Class to read a duty cycle PWM input.

PWM input signals are specified with a frequency and a ratio of high to low in that frequency. There are 8 of these in the roboRIO, and they can be attached to any DigitalSource.

These can be combined as the input of an AnalogTrigger to a Counter in order to implement rollover checking.

Constructs a DutyCycle input from a DigitalSource input.

This class does not own the inputted source.

Parameters **source** – The DigitalSource to use.

getFPGAIndex() → int
Get the FPGA index for the DutyCycle.

Returns the FPGA index

getFrequency() → int
Get the frequency of the duty cycle signal.

Returns frequency in Hertz

getOutput() → float
Get the output ratio of the duty cycle signal.

0 means always low, 1 means always high.

Returns output ratio between 0 and 1

getOutputRaw() → int
Get the raw output ratio of the duty cycle signal.

0 means always low, an output equal to GetOutputScaleFactor() means always high.

Returns output ratio in raw units

getOutputScaleFactor() → int
Get the scale factor of the output.

An output equal to this value is always high, and then linearly scales down to 0. Divide the result of getOutputRaw by this in order to get the percentage between 0 and 1.

Returns the output scale factor

getSourceChannel() → int
Get the channel of the source.

Returns the source channel

1.1.33 DutyCycleEncoder

class wpilib.**DutyCycleEncoder**(**args*, ***kwargs*)
Bases: *wpilib.ErrorBase*, *wpilib.Sendable*

Class for supporting duty cycle/PWM encoders, such as the US Digital MA3 with PWM Output, the CTRE Mag Encoder, the Rev Hex Encoder, and the AM Mag Encoder.

Overloaded function.

1. __init__(self: wpilib._wpilib.DutyCycleEncoder, channel: int) -> None

Construct a new DutyCycleEncoder on a specific channel.

Parameters **channel** – the channel to attach to

2. __init__(self: wpilib._wpilib.DutyCycleEncoder, dutyCycle: wpilib._wpilib.DutyCycle) -> None

Construct a new DutyCycleEncoder attached to an existing DutyCycle object.

Parameters **dutyCycle** – the duty cycle to attach to

3. __init__(self: wpilib._wpilib.DutyCycleEncoder, digitalSource: frc::DigitalSource) -> None

Construct a new DutyCycleEncoder attached to a DigitalSource object.

Parameters **source** – the digital source to attach to

get() → turns
Get the encoder value since the last reset.

This is reported in rotations since the last reset.

Returns the encoder value in rotations

getDistance() → float
Get the distance the sensor has driven since the last reset as scaled by the value from SetDistancePerRotation.

Returns The distance driven since the last reset

getDistancePerRotation() → float
Get the distance per rotation for this encoder.

Returns The scale factor that will be used to convert rotation to useful units.

getFrequency() → int
Get the frequency in Hz of the duty cycle signal from the encoder.

Returns duty cycle frequency in Hz

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

isConnected() → bool
Get if the sensor is connected

This uses the duty cycle frequency to determine if the sensor is connected. By default, a value of 100 Hz is used as the threshold, and this value can be changed with SetConnectedFrequencyThreshold.

Returns true if the sensor is connected

reset() → None
Reset the Encoder distance to zero.

setConnectedFrequencyThreshold(*frequency: int*) → None
Change the frequency threshold for detecting connection used by IsConnected.

Parameters **frequency** – the minimum frequency in Hz.

setDistancePerRotation(*distancePerRotation: float*) → None
Set the distance per rotation of the encoder. This sets the multiplier used to determine the distance driven based on the rotation value from the encoder. Set this value based on the how far the mechanism travels in 1 rotation of the encoder, and factor in gearing reductions following the encoder shaft. This distance can be in any units you like, linear or angular.

Parameters **distancePerRotation** – the distance per rotation of the encoder

1.1.34 Encoder

class wpilib.**Encoder**(**args*, ***kwargs*)

Bases: *wpilib.ErrorBase*, *wpilib.interfaces.CounterBase*, *wpilib.interfaces.PIDSource*, *wpilib.Sendable*

Class to read quad encoders.

Quadrature encoders are devices that count shaft rotation and can sense direction. The output of the QuadEncoder class is an integer that can count either up or down, and can go negative for reverse direction counting. When creating QuadEncoders, a direction is supplied that changes the sense of the output to make code more readable if the encoder is mounted such that forward movement generates negative values. Quadrature encoders have two digital outputs, an A Channel and a B Channel that are out of phase with each other to allow the FPGA to do direction sensing.

All encoders will immediately start counting - Reset() them if you need them to be zeroed before use.

Overloaded function.

1. __init__(self: wpilib._wpilib.Encoder, aChannel: int, bChannel: int, reverseDirection: bool = False, encodingType: wpilib.interfaces._interfaces.CounterBase.EncodingType = EncodingType.k4X) -> None

Encoder constructor.

Construct a Encoder given a and b channels.

The counter will start counting immediately.

Parameters

- **aChannel** – The a channel DIO channel. 0-9 are on-board, 10-25 are on the MXP port
- **bChannel** – The b channel DIO channel. 0-9 are on-board, 10-25 are on the MXP port
- **reverseDirection** – represents the orientation of the encoder and inverts the output values if necessary so forward represents positive values.
- **encodingType** – either k1X, k2X, or k4X to indicate 1X, 2X or 4X decoding. If 4X is selected, then an encoder FPGA object is used and the returned counts will be 4x the encoder spec'd value since all rising and falling edges are counted. If 1X or 2X are selected then a counter object will be used and the returned value will either exactly match the spec'd count or be double (2x) the spec'd count.

2. __init__(self: wpilib._wpilib.Encoder, aSource: frc::DigitalSource, bSource: frc::DigitalSource, reverseDirection: bool = False, encodingType: wpilib.interfaces._interfaces.CounterBase.EncodingType = EncodingType.k4X) -> None

class IndexingType(*arg0: int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kResetWhileHigh

kResetWhileLow

kResetOnFallingEdge

kResetOnRisingEdge

kResetOnFallingEdge = IndexingType.kResetOnFallingEdge

kResetOnRisingEdge = IndexingType.kResetOnRisingEdge

kResetWhileHigh = IndexingType.kResetWhileHigh

kResetWhileLow = IndexingType.kResetWhileLow

name
(self: handle) -> str

get() → int
Gets the current count.

Returns the current count on the Encoder. This method compensates for the decoding type.

Returns Current count from the Encoder adjusted for the 1x, 2x, or 4x scale factor.

getDirection() → bool
The last direction the encoder value changed.

Returns The last direction the encoder value changed.

getDistance() → float
Get the distance the robot has driven since the last reset.

Returns The distance driven since the last reset as scaled by the value from SetDistancePerPulse().

getDistancePerPulse() → float
Get the distance per pulse for this encoder.

Returns The scale factor that will be used to convert pulses to useful units.

getEncodingScale() → int
The encoding scale factor 1x, 2x, or 4x, per the requested encodingType.

Used to divide raw edge counts down to spec'd counts.

getFPGAIndex() → int

getPeriod() → float
Returns the period of the most recent pulse.

Returns the period of the most recent Encoder pulse in seconds. This method compensates for the decoding type.

Warning: This returns unscaled periods. Use GetRate() for rates that are scaled using the value from SetDistancePerPulse().

Returns Period in seconds of the most recent pulse.

getRate() → float
Get the current rate of the encoder.

Units are distance per second as scaled by the value from SetDistancePerPulse().

Returns The current rate of the encoder.

getRaw() → int
Gets the raw value from the encoder.

The raw value is the actual count unscaled by the 1x, 2x, or 4x scale factor.

Returns Current raw count from the encoder

getSamplesToAverage() → int
Get the Samples to Average which specifies the number of samples of the timer to average when calculating the period.

Perform averaging to account for mechanical imperfections or as oversampling to increase resolution.

Returns The number of samples being averaged (from 1 to 127)

getStopped() → bool
Determine if the encoder is stopped.

Using the MaxPeriod value, a boolean is returned that is true if the encoder is considered stopped and false if it is still moving. A stopped encoder is one where the most recent pulse width exceeds the MaxPeriod.

Returns True if the encoder is considered stopped.

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

pidGet() → float

reset() → None
Reset the Encoder distance to zero.

Resets the current count to zero on the encoder.

setDistancePerPulse(*distancePerPulse: float*) → None
Set the distance per pulse for this encoder.

This sets the multiplier used to determine the distance driven based on the count value from the encoder.

Do not include the decoding type in this scale. The library already compensates for the decoding type.

Set this value based on the encoder's rated Pulses per Revolution and factor in gearing reductions following the encoder shaft.

This distance can be in any units you like, linear or angular.

Parameters **distancePerPulse** – The scale factor that will be used to convert pulses to useful units.

setIndexSource(**args*, ***kwargs*)
Overloaded function.

1. setIndexSource(self: wpilib._wpilib.Encoder, channel: int, type: wpilib._wpilib.Encoder.IndexingType = IndexingType.kResetOnRisingEdge) -> None

Set the index source for the encoder.

When this source is activated, the encoder count automatically resets.

Parameters

- **channel** – A DIO channel to set as the encoder index
- **type** – The state that will cause the encoder to reset

2. setIndexSource(self: wpilib._wpilib.Encoder, source: frc::DigitalSource, type: wpilib._wpilib.Encoder.IndexingType = IndexingType.kResetOnRisingEdge) -> None

Set the index source for the encoder.

When this source is activated, the encoder count automatically resets.

Parameters

- **channel** – A digital source to set as the encoder index
- **type** – The state that will cause the encoder to reset

setMaxPeriod(*maxPeriod: float*) → None
Sets the maximum period for stopped detection.

Sets the value that represents the maximum period of the Encoder before it will assume that the attached device is stopped. This timeout allows users to determine if the wheels or other shaft has stopped rotating. This method compensates for the decoding type.

@deprecated Use SetMinRate() in favor of this method. This takes unscaled periods and SetMinRate() scales using value from SetDistancePerPulse().

Parameters **maxPeriod** – The maximum time between rising and falling edges before the FPGA will report the device stopped. This is expressed in seconds.

setMinRate(*minRate: float*) → None
Set the minimum rate of the device before the hardware reports it stopped.

Parameters **minRate** – The minimum rate. The units are in distance per second as scaled by the value from SetDistancePerPulse().

setReverseDirection(*reverseDirection: bool*) → None
Set the direction sensing for this encoder.

This sets the direction sensing on the encoder so that it could count in the correct software direction regardless of the mounting.

Parameters **reverseDirection** – true if the encoder direction should be reversed

setSamplesToAverage(*samplesToAverage: int*) → None
Set the Samples to Average which specifies the number of samples of the timer to average when calculating the period.

Perform averaging to account for mechanical imperfections or as oversampling to increase resolution.

Parameters **samplesToAverage** – The number of samples to average from 1 to 127.

setSimDevice(*device: int*) → None
Indicates this encoder is used by a simulated device.

Parameters **device** – simulated device handle

1.1.35 Error

class wpilib.**Error**(**args*, ***kwargs*)
Bases: pybind11_builtins.pybind11_object

Error object represents a library error.

Overloaded function.

1. __init__(self: wpilib._wpilib.Error) -> None
2. __init__(self: wpilib._wpilib.Error, code: int, contextMessage: str, filename: str, function: str, lineNumber: int, originatingObject: frc::ErrorBase) -> None

clear() → None

getCode() → int

getFilename() → str

getFunction() → str

getLineNumber() → int

getMessage() → str

getOriginatingObject() → frc::ErrorBase

getTimestamp() → float

set(*code: int*, *contextMessage: str*, *filename: str*, *function: str*, *lineNumber: int*, *originatingObject: frc::ErrorBase*) → None

1.1.36 ErrorBase

class wpilib.**ErrorBase**() → None
Bases: pybind11_builtins.pybind11_object

Base class for most objects.

ErrorBase is the base class for most objects since it holds the generated error for that object. In addition, there is a single instance of a global error object.

clearError() → None
@brief Clear the current error information associated with this sensor.

clearGlobalErrors() → None
Clear global errors.

cloneError(*rhs: wpilib._wpilib.ErrorBase*) → None

getError(**args*, ***kwargs*)
Overloaded function.

1. getError(self: wpilib._wpilib.ErrorBase) -> wpilib._wpilib.Error

@brief Retrieve the current error.

Get the current error information associated with this sensor.

2. getError(self: wpilib._wpilib.ErrorBase) -> wpilib._wpilib.Error

@brief Retrieve the current error.

Get the current error information associated with this sensor.

static getGlobalError() → wpilib._wpilib.Error
Retrieve the last global error.

static getGlobalErrors() → List[wpilib._wpilib.Error]
Retrieve all global errors.

setErrnoError(*contextMessage: str*, *filename: str*, *function: str*, *lineNumber: int*) → None
@brief Set error information associated with a C library call that set an error to the "errno" global variable.

Parameters

- **contextMessage** – A custom message from the code that set the error.
- **filename** – Filename of the error source
- **function** – Function of the error source
- **lineNumber** – Line number of the error source

setError(*code: int*, *contextMessage: str*, *filename: str*, *function: str*, *lineNumber: int*) → None
@brief Set the current error information associated with this sensor.

Parameters

- **code** – The error code
- **contextMessage** – A custom message from the code that set the error.

- **filename** – Filename of the error source
- **function** – Function of the error source
- **lineNumber** – Line number of the error source

setErrorRange(*code: int*, *minRange: int*, *maxRange: int*, *requestedValue: int*, *contextMessage: str*, *filename: str*, *function: str*, *lineNumber: int*) → None

@brief Set the current error information associated with this sensor. Range versions use for initialization code.

Parameters

- **code** – The error code
- **minRange** – The minimum allowed allocation range
- **maxRange** – The maximum allowed allocation range
- **requestedValue** – The requested value to allocate
- **contextMessage** – A custom message from the code that set the error.
- **filename** – Filename of the error source
- **function** – Function of the error source
- **lineNumber** – Line number of the error source

static setGlobalError(*code: int*, *contextMessage: str*, *filename: str*, *function: str*, *lineNumber: int*) → None

static setGlobalWPIError(*errorMessage: str*, *contextMessage: str*, *filename: str*, *function: str*, *lineNumber: int*) → None

setImaqError(*success: int*, *contextMessage: str*, *filename: str*, *function: str*, *lineNumber: int*) → None

@brief Set the current error information associated from the nivision Imaq API.

Parameters

- **success** – The return from the function
- **contextMessage** – A custom message from the code that set the error.
- **filename** – Filename of the error source
- **function** – Function of the error source
- **lineNumber** – Line number of the error source

setWPIError(*errorMessage: str*, *code: int*, *contextMessage: str*, *filename: str*, *function: str*, *lineNumber: int*) → None

@brief Set the current error information associated with this sensor.

Parameters

- **errorMessage** – The error message from WPIErrors.h
- **contextMessage** – A custom message from the code that set the error.
- **filename** – Filename of the error source
- **function** – Function of the error source
- **lineNumber** – Line number of the error source

statusIsFatal() → bool

@brief Check if the current error code represents a fatal error.

Returns true if the current error is fatal.

1.1.37 GyroBase

class wpilib.**GyroBase**() → None
Bases: *wpilib.interfaces.Gyro*, *wpilib.ErrorBase*, *wpilib.interfaces.PIDSource*, *wpilib.Sendable*

GyroBase is the common base class for Gyro implementations such as AnalogGyro.

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

pidGet() → float
Get the PIDOutput for the PIDSource base object. Can be set to return angle or rate using SetPIDSourceType(). Defaults to angle.

Returns The PIDOutput (angle or rate, defaults to angle)

1.1.38 I2C

class wpilib.**I2C**(*port: wpilib._wpilib.I2C.Port*, *deviceAddress: int*) → None
Bases: *wpilib.ErrorBase*

I2C bus interface class.

This class is intended to be used by sensor (and other I2C device) drivers. It probably should not be used directly.

Constructor.

Parameters

- **port** – The I2C port to which the device is connected.
- **deviceAddress** – The address of the device on the I2C bus.

class Port(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kOnboard

kMXP

kMXP = Port.kMXP

kOnboard = Port.kOnboard

name
(self: handle) -> str

addressOnly() → bool
Attempt to address a device on the I2C bus.

This allows you to figure out if there is a device on the I2C bus that responds to the address specified in the constructor.

Returns Transfer Aborted... false for success, true for aborted.

read(*registerAddress: int*, *data: buffer*) → bool
Execute a read transaction with the device.

Read bytes from a device. Most I2C devices will auto-increment the register pointer internally allowing you to read consecutive registers on a device in a single transaction.

Parameters

- **registerAddress** – The register to read first in the transaction.
- **count** – The number of bytes to read in the transaction.
- **buffer** – A pointer to the array of bytes to store the data read from the device.

Returns Transfer Aborted... false for success, true for aborted.

readOnly(*buffer: buffer*) → bool
Execute a read only transaction with the device.

Read bytes from a device. This method does not write any data to prompt the device.

Parameters

- **buffer** – A pointer to the array of bytes to store the data read from the device.
- **count** – The number of bytes to read in the transaction.

Returns Transfer Aborted... false for success, true for aborted.

transaction(*dataToSend: buffer*, *dataReceived: buffer*) → bool
Generic transaction.

This is a lower-level interface to the I2C hardware giving you more control over each transaction. If you intend to write multiple bytes in the same transaction and do not plan to receive anything back, use writeBulk() instead. Calling this with a receiveSize of 0 will result in an error.

Parameters

- **dataToSend** – Buffer of data to send as part of the transaction.
- **sendSize** – Number of bytes to send as part of the transaction.
- **dataReceived** – Buffer to read data into.
- **receiveSize** – Number of bytes to read from the device.

Returns Transfer Aborted... false for success, true for aborted.

verifySensor(*registerAddress: int*, *expected: buffer*) → bool
Verify that a device's registers contain expected values.

Most devices will have a set of registers that contain a known value that can be used to identify them. This allows an I2C device driver to easily verify that the device contains the expected value.

@pre The device must support and be configured to use register auto-increment.

Parameters

- **registerAddress** – The base register to start reading from the device.
- **count** – The size of the field to be verified.
- **expected** – A buffer containing the values expected from the device.

write(*registerAddress: int*, *data: int*) → bool
Execute a write transaction with the device.

Write a single byte to a register on a device and wait until the transaction is complete.

Parameters

- **registerAddress** – The address of the register on the device to be written.

- **data** – The byte to write to the register on the device.

Returns Transfer Aborted... false for success, true for aborted.

writeBulk(*data: buffer*) → bool
Execute a bulk write transaction with the device.

Write multiple bytes to a device and wait until the transaction is complete.

Parameters

- **data** – The data to write to the register on the device.
- **count** – The number of bytes to be written.

Returns Transfer Aborted... false for success, true for aborted.

1.1.39 InterruptableSensorBase

class wpilib.**InterruptableSensorBase**() → None
Bases: *wpilib.ErrorBase*

class WaitResult(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kTimeout

kRisingEdge

kFallingEdge

kBoth

kBoth = WaitResult.kBoth

kFallingEdge = WaitResult.kFallingEdge

kRisingEdge = WaitResult.kRisingEdge

kTimeout = WaitResult.kTimeout

name
(self: handle) -> str

cancelInterrupts() → None
Cancel interrupts on this device.

This deallocates all the chipobject structures and disables any interrupts.

disableInterrupts() → None
Disable Interrupts without without deallocating structures.

enableInterrupts() → None
Enable interrupts to occur on this input.

Interrupts are disabled when the RequestInterrupt call is made. This gives time to do the setup of the other options before starting to field interrupts.

getAnalogTriggerTypeForRouting() → wpilib._wpilib.AnalogTriggerType

getPortHandleForRouting() → int

readFallingTimestamp() → float
Return the timestamp for the falling interrupt that occurred most recently.

This is in the same time domain as GetClock(). The falling-edge interrupt should be enabled with `DigitalInput.SetUpSourceEdge()`

Returns Timestamp in seconds since boot.

readRisingTimestamp() → float
Return the timestamp for the rising interrupt that occurred most recently.

This is in the same time domain as GetClock(). The rising-edge interrupt should be enabled with `DigitalInput.SetUpSourceEdge()`

Returns Timestamp in seconds since boot.

requestInterrupts(**args*, ***kwargs*)
Overloaded function.

1. requestInterrupts(self: wpilib._wpilib.InterruptableSensorBase, handler: Callable[[wpilib._wpilib.InterruptableSensorBase.WaitResult], None]) -> None

Request one of the 8 interrupts asynchronously on this digital input.

Request interrupts in asynchronous mode where the user's interrupt handler will be called when the interrupt fires. Users that want control over the thread priority should use the synchronous method with their own spawned thread. The default is interrupt on rising edges only.

2. requestInterrupts(self: wpilib._wpilib.InterruptableSensorBase) -> None

Request one of the 8 interrupts synchronously on this digital input.

Request interrupts in synchronous mode where the user program will have to explicitly wait for the interrupt to occur using WaitForInterrupt. The default is interrupt on rising edges only.

setUpSourceEdge(*risingEdge: bool*, *fallingEdge: bool*) → None
Set which edge to trigger interrupts on

Parameters

- **risingEdge** – true to interrupt on rising edge
- **fallingEdge** – true to interrupt on falling edge

waitForInterrupt(*timeout: float*, *ignorePrevious: bool = True*) → wpilib._wpilib.InterruptableSensorBase.WaitResult
In synchronous mode, wait for the defined interrupt to occur.

You should **NOT** attempt to read the sensor from another thread while waiting for an interrupt. This is not threadsafe, and can cause memory corruption

Parameters

- **timeout** – Timeout in seconds
- **ignorePrevious** – If true, ignore interrupts that happened before WaitForInterrupt was called.

Returns What interrupts fired

1.1.40 IterativeRobot

class wpilib.**IterativeRobot**() → None
Bases: *wpilib.IterativeRobotBase*

IterativeRobot implements the IterativeRobotBase robot program framework.

The IterativeRobot class is intended to be subclassed by a user creating a robot program.

Periodic() functions from the base class are called each time a new packet is received from the driver station.

endCompetition() → None
Ends the main loop in StartCompetition().

startCompetition() → None
Provide an alternate "main loop" via StartCompetition().

This specific StartCompetition() implements "main loop" behaviour synced with the DS packets.

1.1.41 IterativeRobotBase

class wpilib.**IterativeRobotBase**(*period: seconds*) → None
Bases: *wpilib.RobotBase*

IterativeRobotBase implements a specific type of robot program framework, extending the RobotBase class.

The IterativeRobotBase class does not implement StartCompetition(), so it should not be used by teams directly.

This class provides the following functions which are called by the main loop, StartCompetition(), at the appropriate times:

RobotInit() – provide for initialization at robot power-on

Init() functions – each of the following functions is called once when the appropriate mode is entered: - DisabledInit() – called each and every time disabled is entered from another mode - AutonomousInit() – called each and every time autonomous is entered from another mode - TeleopInit() – called each and every time teleop is entered from another mode - TestInit() – called each and every time test is entered from another mode

Periodic() functions – each of these functions is called on an interval: - RobotPeriodic() - DisabledPeriodic() - AutonomousPeriodic() - TeleopPeriodic() - TestPeriodic()

Constructor for IterativeRobotBase.

Parameters **period** – Period.

autonomousInit() → None
Initialization code for autonomous mode should go here.

Users should override this method for initialization code which will be called each time the robot enters autonomous mode.

autonomousPeriodic() → None
Periodic code for autonomous mode should go here.

Users should override this method for code which will be called each time a new packet is received from the driver station and the robot is in autonomous mode.

disabledInit() → None
Initialization code for disabled mode should go here.

Users should override this method for initialization code which will be called each time the robot enters disabled mode.

disabledPeriodic() → None
Periodic code for disabled mode should go here.

Users should override this method for code which will be called each time a new packet is received from the driver station and the robot is in disabled mode.

robotInit() → None
Robot-wide initialization code should go here.

Users should override this method for default Robot-wide initialization which will be called when the robot is first powered on. It will be called exactly one time.

Warning: the Driver Station "Robot Code" light and FMS "Robot Ready" indicators will be off until RobotInit() exits. Code in RobotInit() that waits for enable will cause the robot to never indicate that the code is ready, causing the robot to be bypassed in a match.

robotPeriodic() → None
Periodic code for all modes should go here.

This function is called each time a new packet is received from the driver station.

teleopInit() → None
Initialization code for teleop mode should go here.

Users should override this method for initialization code which will be called each time the robot enters teleop mode.

teleopPeriodic() → None
Periodic code for teleop mode should go here.

Users should override this method for code which will be called each time a new packet is received from the driver station and the robot is in teleop mode.

testInit() → None
Initialization code for test mode should go here.

Users should override this method for initialization code which will be called each time the robot enters test mode.

testPeriodic() → None
Periodic code for test mode should go here.

Users should override this method for code which will be called each time a new packet is received from the driver station and the robot is in test mode.

1.1.42 Jaguar

class wpilib.**Jaguar**(*channel: int*) → None
Bases: *wpilib.PWMSpeedController*

Luminary Micro / Vex Robotics Jaguar Speed Controller with PWM control.

Note that the Jaguar uses the following bounds for PWM values. These values should work reasonably well for most controllers, but if users experience issues such as asymmetric behavior around the deadband or inability to saturate the controller in either direction, calibration is recommended. The calibration procedure can be found in the Jaguar User Manual available from Vex.

- 2.310ms = full "forward"
- 1.550ms = the "high end" of the deadband range
- 1.507ms = center of the deadband range (off)
- 1.454ms = the "low end" of the deadband range
- 0.697ms = full "reverse"

Constructor for a Jaguar connected via PWM.

Parameters **channel** – The PWM channel that the Jaguar is attached to. 0-9 are on-board, 10-19 are on the MXP port

1.1.43 Joystick

class wpilib.**Joystick**(*port: int*) → None
Bases: *wpilib.interfaces.GenericHID*

Handle input from standard Joysticks connected to the Driver Station.

This class handles standard input that comes from the Driver Station. Each time a value is requested the most recent value is returned. There is a single class instance for each joystick and the mapping of ports to hardware buttons depends on the code in the Driver Station.

Construct an instance of a joystick.

The joystick index is the USB port on the Driver Station.

Parameters **port** – The port on the Driver Station that the joystick is plugged into (0-5).

class **AxisType**(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kXAxis

kYAxis

kZAxis

kTwistAxis

kThrottleAxis

kThrottleAxis = AxisType.kThrottleAxis

kTwistAxis = AxisType.kTwistAxis

kXAxis = AxisType.kXAxis

kYAxis = AxisType.kYAxis

kZAxis = AxisType.kZAxis

name
(self: handle) -> str

class **ButtonType**(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kTriggerButton

kTopButton

kTopButton = ButtonType.kTopButton

kTriggerButton = ButtonType.kTriggerButton

name
(self: handle) -> str

getDirectionDegrees() → float
Get the direction of the vector formed by the joystick and its origin in degrees.

Returns The direction of the vector in degrees

getDirectionRadians() → float
Get the direction of the vector formed by the joystick and its origin in radians.

Returns The direction of the vector in radians

getMagnitude() → float
Get the magnitude of the direction vector formed by the joystick's current position relative to its origin.

Returns The magnitude of the direction vector

getThrottle() → float
Get the throttle value of the current joystick.

This depends on the mapping of the joystick connected to the current port.

getThrottleChannel() → int
Get the channel currently associated with the throttle axis.

Returns The channel for the axis.

getTop() → bool
Read the state of the top button on the joystick.

Look up which button has been assigned to the top and read its state.

Returns The state of the top button.

getTopPressed() → bool
Whether the top button was pressed since the last check.

Returns Whether the button was pressed since the last check.

getTopReleased() → bool
Whether the top button was released since the last check.

Returns Whether the button was released since the last check.

getTrigger() → bool
Read the state of the trigger on the joystick.

Look up which button has been assigned to the trigger and read its state.

Returns The state of the trigger.

getTriggerPressed() → bool
Whether the trigger was pressed since the last check.

Returns Whether the button was pressed since the last check.

getTriggerReleased() → bool
Whether the trigger was released since the last check.

Returns Whether the button was released since the last check.

getTwist() → float
Get the twist value of the current joystick.

This depends on the mapping of the joystick connected to the current port.

getTwistChannel() → int
Get the channel currently associated with the twist axis.

Returns The channel for the axis.

getX(*hand: wpilib.interfaces._interfaces.GenericHID.Hand = Hand.kRightHand*) → float
Get the X value of the joystick.

This depends on the mapping of the joystick connected to the current port.

Parameters **hand** – This parameter is ignored for the Joystick class and is only here to complete the GenericHID interface.

getXChannel() → int
Get the channel currently associated with the X axis.

Returns The channel for the axis.

getY(*hand: wpilib.interfaces._interfaces.GenericHID.Hand = Hand.kRightHand*) → float
Get the Y value of the joystick.

This depends on the mapping of the joystick connected to the current port.

Parameters **hand** – This parameter is ignored for the Joystick class and is only here to complete the GenericHID interface.

getYChannel() → int
Get the channel currently associated with the Y axis.

Returns The channel for the axis.

getZ() → float
Get the Z value of the current joystick.

This depends on the mapping of the joystick connected to the current port.

getZChannel() → int
Get the channel currently associated with the Z axis.

Returns The channel for the axis.

kDefaultThrottleChannel = 3

kDefaultTwistChannel = 2

kDefaultXChannel = 0

kDefaultYChannel = 1

kDefaultZChannel = 2

setThrottleChannel(*channel: int*) → None
Set the channel associated with the throttle axis.

Parameters

- **axis** – The axis to set the channel for.
- **channel** – The channel to set the axis to.

setTwistChannel(*channel: int*) → None
Set the channel associated with the twist axis.

Parameters

- **axis** – The axis to set the channel for.
- **channel** – The channel to set the axis to.

setXChannel(*channel: int*) → None
Set the channel associated with the X axis.

Parameters **channel** – The channel to set the axis to.

setYChannel(*channel: int*) → None
Set the channel associated with the Y axis.

Parameters

- **axis** – The axis to set the channel for.
- **channel** – The channel to set the axis to.

setZChannel(*channel: int*) → None
Set the channel associated with the Z axis.

Parameters

- **axis** – The axis to set the channel for.
- **channel** – The channel to set the axis to.

1.1.44 LinearFilter

class wpilib.**LinearFilter**() → None
Bases: pybind11_builtins.pybind11_object

This class implements a linear, digital filter. All types of FIR and IIR filters are supported. Static factory methods are provided to create commonly used types of filters.

Filters are of the form: y[n] = (b0 * x[n] + b1 * x[n-1] + … + bP * x[n-P]) - (a0 * y[n-1] + a2 * y[n-2] + … + aQ * y[n-Q])

Where: y[n] is the output at time "n" x[n] is the input at time "n" y[n-1] is the output from the LAST time step ("n-1") x[n-1] is the input from the LAST time step ("n-1") b0 … bP are the "feedforward" (FIR) gains a0 … aQ are the "feedback" (IIR) gains IMPORTANT! Note the "-" sign in front of the feedback term! This is a common convention in signal processing.

What can linear filters do? Basically, they can filter, or diminish, the effects of undesirable input frequencies. High frequencies, or rapid changes, can be indicative of sensor noise or be otherwise undesirable. A "low pass" filter smooths out the signal, reducing the impact of these high frequency components. Likewise, a "high pass" filter gets rid of slow-moving signal components, letting you detect large changes more easily.

Example FRC applications of filters: - Getting rid of noise from an analog sensor input (note: the roboRIO's FPGA can do this faster in hardware) - Smoothing out joystick input to prevent the wheels from slipping or the robot from tipping - Smoothing motor commands so that unnecessary strain isn't put on electrical or mechanical components - If you use clever gains, you can make a PID controller out of this class!

For more on filters, we highly recommend the following articles: https://en.wikipedia.org/wiki/Linear_filter https://en.wikipedia.org/wiki/Iir_filter https://en.wikipedia.org/wiki/Fir_filter

Note 1: Calculate() should be called by the user on a known, regular period. You can use a Notifier for this or do it "inline" with code in a periodic function.

Note 2: For ALL filters, gains are necessarily a function of frequency. If you make a filter that works well for you at, say, 100Hz, you will most definitely need to adjust the gains if you then want to run it at 200Hz! Combining this with Note 1 - the impetus is on YOU as a developer to make sure Calculate() gets called at the desired, constant frequency!

Create a linear FIR or IIR filter.

Parameters

- **ffGains** – The "feed forward" or FIR gains.
- **fbGains** – The "feed back" or IIR gains.

calculate(*input: float*) → float
Calculates the next value of the filter.

Parameters **input** – Current input value.

Returns The filtered value at this step

static highPass(*timeConstant: float*, *period: seconds*) → wpilib._wpilib.LinearFilter
Creates a first-order high-pass filter of the form: y[n] = gain * x[n] + (-gain) * x[n-1] + gain * y[n-1] where gain = e^{-dt / T}, T is the time constant in seconds

This filter is stable for time constants greater than zero.

Parameters

- **timeConstant** – The discrete-time time constant in seconds.
- **period** – The period in seconds between samples taken by the user.

static movingAverage(*taps: int*) → wpilib._wpilib.LinearFilter
Creates a K-tap FIR moving average filter of the form: y[n] = 1/k * (x[k] + x[k-1] + ... + x[0])

This filter is always stable.

Parameters **taps** – The number of samples to average over. Higher = smoother but slower

reset() → None
Reset the filter state.

static singlePoleIIR(*timeConstant: float*, *period: seconds*) → wpilib._wpilib.LinearFilter
Creates a one-pole IIR low-pass filter of the form: y[n] = (1 - gain) * x[n] + gain * y[n-1] where gain = e^{-dt / T}, T is the time constant in seconds

This filter is stable for time constants greater than zero.

Parameters

- **timeConstant** – The discrete-time time constant in seconds.
- **period** – The period in seconds between samples taken by the user.

1.1.45 LiveWindow

class wpilib.**LiveWindow**
Bases: pybind11_builtins.pybind11_object

The LiveWindow class is the public interface for putting sensors and actuators on the LiveWindow.

disableAllTelemetry() → None
Disable ALL telemetry.

disableTelemetry(*component: frc::Sendable*) → None
Disable telemetry for a single component.

Parameters **sendable** – component

disabled

enableTelemetry(*component: frc::Sendable*) → None
Enable telemetry for a single component.

Parameters **sendable** – component

enabled

static getInstance() → wpilib._wpilib.LiveWindow
Get an instance of the LiveWindow main class.

This is a singleton to guarantee that there is only a single instance regardless of how many times GetInstance is called.

isEnabled() → bool

setEnabled(*enabled: bool*) → None
Change the enabled status of LiveWindow.

If it changes to enabled, start livewindow running otherwise stop it

updateValues() → None
Tell all the sensors to update (send) their values.

Actuators are handled through callbacks on their value changing from the SmartDashboard widgets.

1.1.46 MedianFilter

class wpilib.**MedianFilter**(*size: int*) → None
Bases: pybind11_builtins.pybind11_object

A class that implements a moving-window median filter. Useful for reducing measurement noise, especially with processes that generate occasional, extreme outliers (such as values from vision processing, LIDAR, or ultrasonic sensors).

Creates a new MedianFilter.

Parameters **size** – The number of samples in the moving window.

calculate(*next: float*) → float
Calculates the moving-window median for the next value of the input stream.

Parameters **next** – The next input value.

Returns The median of the moving window, updated to include the next value.

reset() → None
Resets the filter, clearing the window of all elements.

1.1.47 MotorSafety

class wpilib.**MotorSafety**() → None
Bases: *wpilib.ErrorBase*

This base class runs a watchdog timer and calls the subclass's StopMotor() function if the timeout expires.

The subclass should call Feed() whenever the motor value is updated.

check() → None
Check if this motor has exceeded its timeout.

This method is called periodically to determine if this motor has exceeded its timeout value. If it has, the stop method is called, and the motor is shut down until its value is updated again.

static checkMotors() → None
Check the motors to see if any have timed out.

This static method is called periodically to poll all the motors and stop any that have timed out.

feed() → None
Feed the motor safety object.

Resets the timer on this object that is used to do the timeouts.

getDescription(*desc: wpi::raw_ostream*) → None

getExpiration() → float
Retrieve the timeout value for the corresponding motor safety object.

Returns the timeout value in seconds.

isAlive() → bool
Determine if the motor is still operating or has timed out.

Returns true if the motor is still operating normally and hasn't timed out.

isSafetyEnabled() → bool
Return the state of the motor safety enabled flag.

Return if the motor safety is currently enabled for this device.

Returns True if motor safety is enforced for this device.

setExpiration(*expirationTime: float*) → None
Set the expiration time for the corresponding motor safety object.

Parameters **expirationTime** – The timeout value in seconds.

setSafetyEnabled(*enabled: bool*) → None
Enable/disable motor safety for this device.

Turn on and off the motor safety option for this PWM object.

Parameters **enabled** – True if motor safety is enforced for this object.

stopMotor() → None

1.1.48 NidecBrushless

class wpilib.**NidecBrushless**(*pwmChannel: int*, *dioChannel: int*) → None
Bases: *wpilib.interfaces.SpeedController*, *wpilib.MotorSafety*, *wpilib.Sendable*

Nidec Brushless Motor.

Constructor.

Parameters

- **pwmChannel** – The PWM channel that the Nidec Brushless controller is attached to. 0-9 are on-board, 10-19 are on the MXP port.
- **dioChannel** – The DIO channel that the Nidec Brushless controller is attached to. 0-9 are on-board, 10-25 are on the MXP port.

disable() → None
Disable the motor. The Enable() function must be called to re-enable the motor.

enable() → None
Re-enable the motor after Disable() has been called. The Set() function must be called to set a new motor speed.

get() → float
Get the recently set value of the PWM.

Returns The most recently set value for the PWM between -1.0 and 1.0.

getChannel() → int
Gets the channel number associated with the object.

Returns The channel number.

getDescription(*desc: wpi::raw_ostream*) → None

getInverted() → bool

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

pidWrite(*output: float*) → None
Write out the PID value as seen in the PIDOutput base object.

Parameters **output** – Write out the PWM value as was found in the PIDController

set(*speed: float*) → None
Set the PWM value.

The PWM value is set using a range of -1.0 to 1.0, appropriately scaling the value for the FPGA.

Parameters **speed** – The speed value between -1.0 and 1.0 to set.

setInverted(*isInverted: bool*) → None

stopMotor() → None

1.1.49 Notifier

class wpilib.**Notifier**() → None
Bases: *wpilib.ErrorBase*

Create a Notifier for timer event notification.

Parameters **handler** – The handler is called at the notification time which is set using StartSingle or StartPeriodic.

setHandler() → None
Change the handler function.

Parameters **handler** – Handler

setName(*name: str*) → None
Sets the name of the notifier. Used for debugging purposes only.

Parameters **name** – Name

startPeriodic(*period: seconds*) → None
Register for periodic event notification.

A timer event is queued for periodic event notification. Each time the interrupt occurs, the event will be immediately requeued for the same time interval.

Parameters **period** – Period to call the handler starting one period after the call to this method.

startSingle(*delay: seconds*) → None
Register for single event notification.

A timer event is queued for a single event after the specified delay.

Parameters **delay** – Amount of time to wait before the handler is called.

stop() → None
Stop timer events from occuring.

Stop any repeating timer events from occuring. This will also remove any single notification events from the queue.

If a timer-based call to the registered handler is in progress, this function will block until the handler call is complete.

1.1.50 PWM

class wpilib.**PWM**(*channel: int*) → None
Bases: *wpilib.MotorSafety*, *wpilib.Sendable*

Class implements the PWM generation in the FPGA.

The values supplied as arguments for PWM outputs range from -1.0 to 1.0. They are mapped to the hardware dependent values, in this case 0-2000 for the FPGA. Changes are immediately sent to the FPGA, and the update occurs at the next FPGA cycle (5.005ms). There is no delay.

As of revision 0.1.10 of the FPGA, the FPGA interprets the 0-2000 values as follows: - 2000 = maximum pulse width - 1999 to 1001 = linear scaling from "full forward" to "center" - 1000 = center value - 999 to 2 = linear scaling from "center" to "full reverse" - 1 = minimum pulse width (currently 0.5ms) - 0 = disabled (i.e. PWM output is held low)

Allocate a PWM given a channel number.

Checks channel value range and allocates the appropriate channel. The allocation is only done to help users ensure that they don't double assign channels.

Parameters **channel** – The PWM channel number. 0-9 are on-board, 10-19 are on the MXP port

class **PeriodMultiplier**(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Represents the amount to multiply the minimum servo-pulse pwm period by.

Members:

kPeriodMultiplier_1X : Don't skip pulses. PWM pulses occur every 5.005 ms

kPeriodMultiplier_2X : Skip every other pulse. PWM pulses occur every 10.010 ms

kPeriodMultiplier_4X : Skip three out of four pulses. PWM pulses occur every 20.020 ms

kPeriodMultiplier_1X = PeriodMultiplier.kPeriodMultiplier_1X

kPeriodMultiplier_2X = PeriodMultiplier.kPeriodMultiplier_2X

kPeriodMultiplier_4X = PeriodMultiplier.kPeriodMultiplier_4X

name
(self: handle) -> str

enableDeadbandElimination(*eliminateDeadband: bool*) → None
Optionally eliminate the deadband from a speed controller.

Parameters **eliminateDeadband** – If true, set the motor curve on the Jaguar to eliminate the deadband in the middle of the range. Otherwise, keep the full range without modifying any values.

getChannel() → int

getDescription(*desc: wpi::raw_ostream*) → None

getPosition() → float
Get the PWM value in terms of a position.

This is intended to be used by servos.

@pre SetMaxPositivePwm() called. @pre SetMinNegativePwm() called.

Returns The position the servo is set to between 0.0 and 1.0.

getRaw() → int
Get the PWM value directly from the hardware.

Read a raw value from a PWM channel.

Returns Raw PWM control value.

getRawBounds() → Tuple[int, int, int, int, int]
Get the bounds on the PWM values.

This Gets the bounds on the PWM values for a particular each type of controller. The values determine the upper and lower speeds as well as the deadband bracket.

Parameters

- **max** – The Minimum pwm value
- **deadbandMax** – The high end of the deadband range
- **center** – The center speed (off)
- **deadbandMin** – The low end of the deadband range
- **min** – The minimum pwm value

getSpeed() → float
Get the PWM value in terms of speed.

This is intended to be used by speed controllers.

@pre SetMaxPositivePwm() called. @pre SetMinPositivePwm() called. @pre SetMaxNegativePwm() called. @pre SetMinNegativePwm() called.

Returns The most recently set speed between -1.0 and 1.0.

setBounds(*max: float*, *deadbandMax: float*, *center: float*, *deadbandMin: float*, *min: float*) → None
Set the bounds on the PWM pulse widths.

This sets the bounds on the PWM values for a particular type of controller. The values determine the upper and lower speeds as well as the deadband bracket.

Parameters

- **max** – The max PWM pulse width in ms
- **deadbandMax** – The high end of the deadband range pulse width in ms
- **center** – The center (off) pulse width in ms
- **deadbandMin** – The low end of the deadband pulse width in ms
- **min** – The minimum pulse width in ms

setDisabled() → None
Temporarily disables the PWM output. The next set call will reenable the output.

setPeriodMultiplier(*mult: wpilib._wpilib.PWM.PeriodMultiplier*) → None
Slow down the PWM signal for old devices.

Parameters **mult** – The period multiplier to apply to this channel

setPosition(*pos: float*) → None
Set the PWM value based on a position.

This is intended to be used by servos.

@pre SetMaxPositivePwm() called. @pre SetMinNegativePwm() called.

Parameters **pos** – The position to set the servo between 0.0 and 1.0.

setRaw(*value: int*) → None
Set the PWM value directly to the hardware.

Write a raw value to a PWM channel.

Parameters **value** – Raw PWM value.

setRawBounds(*max: int*, *deadbandMax: int*, *center: int*, *deadbandMin: int*, *min: int*) → None
Set the bounds on the PWM values.

This sets the bounds on the PWM values for a particular each type of controller. The values determine the upper and lower speeds as well as the deadband bracket.

Parameters

- **max** – The Minimum pwm value
- **deadbandMax** – The high end of the deadband range
- **center** – The center speed (off)
- **deadbandMin** – The low end of the deadband range
- **min** – The minimum pwm value

setSpeed(*speed: float*) → None
Set the PWM value based on a speed.

This is intended to be used by speed controllers.

@pre SetMaxPositivePwm() called. @pre SetMinPositivePwm() called. @pre SetCenterPwm() called. @pre SetMaxNegativePwm() called. @pre SetMinNegativePwm() called.

Parameters **speed** – The speed to set the speed controller between -1.0 and 1.0.

setZeroLatch() → None

stopMotor() → None

1.1.51 PWMSparkMax

class wpilib.**PWMSparkMax**(*channel: int*) → None
Bases: *wpilib.PWMSpeedController*

REV Robotics SPARK MAX Speed Controller.

Note that the SPARK MAX uses the following bounds for PWM values. These values should work reasonably well for most controllers, but if users experience issues such as asymmetric behavior around the deadband or inability to saturate the controller in either direction, calibration is recommended. The calibration procedure can be found in the SPARK MAX User Manual available from REV Robotics.

- 2.003ms = full "forward"
- 1.550ms = the "high end" of the deadband range

- 1.500ms = center of the deadband range (off)
- 1.460ms = the "low end" of the deadband range
- 0.999ms = full "reverse"

Constructor for a SPARK MAX.

Parameters **channel** – The PWM channel that the SPARK MAX is attached to. 0-9 are on-board, 10-19 are on the MXP port

1.1.52 PWMSpeedController

class wpilib.**PWMSpeedController**(*channel: int*) → None
Bases: *wpilib.PWM*, *wpilib.interfaces.SpeedController*

Common base class for all PWM Speed Controllers.

Constructor for a PWM Speed Controller connected via PWM.

Parameters **channel** – The PWM channel that the controller is attached to. 0-9 are on-board, 10-19 are on the MXP port

disable() → None

get() → float
Get the recently set value of the PWM.

Returns The most recently set value for the PWM between -1.0 and 1.0.

getInverted() → bool

pidWrite(*output: float*) → None
Write out the PID value as seen in the PIDOutput base object.

Parameters **output** – Write out the PWM value as was found in the PIDController

set(*value: float*) → None
Set the PWM value.

The PWM value is set using a range of -1.0 to 1.0, appropriately scaling the value for the FPGA.

Parameters **speed** – The speed value between -1.0 and 1.0 to set.

setInverted(*isInverted: bool*) → None

stopMotor() → None

1.1.53 PWMTalonFX

class wpilib.**PWMTalonFX**(*channel: int*) → None
Bases: *wpilib.PWMSpeedController*

Cross the Road Electronics (CTRE) Talon FX Speed Controller with PWM control.

Note that the Talon FX uses the following bounds for PWM values. These values should work reasonably well for most controllers, but if users experience issues such as asymmetric behavior around the deadband or inability to saturate the controller in either direction, calibration is recommended. The calibration procedure can be found in the Talon FX User Manual available from Cross The Road Electronics.

- 2.004ms = full "forward"
- 1.520ms = the "high end" of the deadband range

- 1.500ms = center of the deadband range (off)
- 1.480ms = the "low end" of the deadband range
- 0.997ms = full "reverse"

Construct a Talon FX connected via PWM.

Parameters **channel** – The PWM channel that the Talon FX is attached to. 0-9 are on-board, 10-19 are on the MXP port

1.1.54 PWMTalonSRX

class wpilib.**PWMTalonSRX**(*channel: int*) → None
Bases: *wpilib.PWMSpeedController*

Cross the Road Electronics (CTRE) Talon SRX Speed Controller with PWM control.

Note that the Talon SRX uses the following bounds for PWM values. These values should work reasonably well for most controllers, but if users experience issues such as asymmetric behavior around the deadband or inability to saturate the controller in either direction, calibration is recommended. The calibration procedure can be found in the Talon SRX User Manual available from Cross The Road Electronics.

- 2.004ms = full "forward"
- 1.520ms = the "high end" of the deadband range
- 1.500ms = center of the deadband range (off)
- 1.480ms = the "low end" of the deadband range
- 0.997ms = full "reverse"

Construct a Talon SRX connected via PWM.

Parameters **channel** – The PWM channel that the Talon SRX is attached to. 0-9 are on-board, 10-19 are on the MXP port

1.1.55 PWMVenom

class wpilib.**PWMVenom**(*channel: int*) → None
Bases: *wpilib.PWMSpeedController*

Playing with Fusion Venom Smart Motor with PWM control.

Note that the Venom uses the following bounds for PWM values. These values should work reasonably well for most controllers, but if users experience issues such as asymmetric behavior around the deadband or inability to saturate the controller in either direction, calibration is recommended.

- 2.004ms = full "forward"
- 1.520ms = the "high end" of the deadband range
- 1.500ms = center of the deadband range (off)
- 1.480ms = the "low end" of the deadband range
- 0.997ms = full "reverse"

Construct a Venom connected via PWM.

Parameters **channel** – The PWM channel that the Venom is attached to. 0-9 are on-board, 10-19 are on the MXP port

1.1.56 PWMVictorSPX

class wpilib.**PWMVictorSPX**(*channel: int*) → None
Bases: *wpilib.PWMSpeedController*

Cross the Road Electronics (CTRE) Victor SPX Speed Controller with PWM control.

Note that the Victor SPX uses the following bounds for PWM values. These values should work reasonably well for most controllers, but if users experience issues such as asymmetric behavior around the deadband or inability to saturate the controller in either direction, calibration is recommended. The calibration procedure can be found in the Victor SPX User Manual available from Cross The Road Electronics.

- 2.004ms = full "forward"
- 1.520ms = the "high end" of the deadband range
- 1.500ms = center of the deadband range (off)
- 1.480ms = the "low end" of the deadband range
- 0.997ms = full "reverse"

Construct a Victor SPX connected via PWM.

Parameters **channel** – The PWM channel that the Victor SPX is attached to. 0-9 are on-board, 10-19 are on the MXP port

1.1.57 PowerDistributionPanel

class wpilib.**PowerDistributionPanel**(**args*, ***kwargs*)
Bases: *wpilib.ErrorBase*, *wpilib.Sendable*

Class for getting voltage, current, temperature, power and energy from the CAN PDP.

Overloaded function.

1. __init__(self: wpilib._wpilib.PowerDistributionPanel) -> None
2. __init__(self: wpilib._wpilib.PowerDistributionPanel, module: int) -> None

clearStickyFaults() → None
Remove all of the fault flags on the PDP.

getCurrent(*channel: int*) → float
Query the current of a single channel of the PDP.

Returns The current of one of the PDP channels (channels 0-15) in Amperes

getTemperature() → float
Query the temperature of the PDP.

Returns The temperature of the PDP in degrees Celsius

getTotalCurrent() → float
Query the total current of all monitored PDP channels (0-15).

Returns The the total current drawn from the PDP channels in Amperes

getTotalEnergy() → float
Query the total energy drawn from the monitored PDP channels.

Returns The the total energy drawn from the PDP channels in Joules

getTotalPower() → float
Query the total power drawn from the monitored PDP channels.

Returns The the total power drawn from the PDP channels in Watts

getVoltage() → float
Query the input voltage of the PDP.

Returns The voltage of the PDP in volts

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

resetTotalEnergy() → None
Reset the total energy drawn from the PDP.

@see PowerDistributionPanel#GetTotalEnergy

1.1.58 Preferences

class wpilib.**Preferences**
Bases: pybind11_builtins.pybind11_object

The preferences class provides a relatively simple way to save important values to the roboRIO to access the next time the roboRIO is booted.

This class loads and saves from a file inside the roboRIO. The user cannot access the file directly, but may modify values at specific fields which will then be automatically periodically saved to the file by the NetworkTable server.

This class is thread safe.

This will also interact with {@link NetworkTable} by creating a table called "Preferences" with all the key-value pairs.

containsKey(*key: str*) → bool
Returns whether or not there is a key with the given name.

Parameters **key** – the key

Returns if there is a value at the given key

getBoolean(*key: str*, *defaultValue: bool = False*) → bool
Returns the boolean at the given key. If this table does not have a value for that position, then the given defaultValue value will be returned.

Parameters

- **key** – the key
- **defaultValue** – the value to return if none exists in the table

Returns either the value in the table, or the defaultValue

getDouble(*key: str*, *defaultValue: float = 0.0*) → float
Returns the double at the given key. If this table does not have a value for that position, then the given defaultValue value will be returned.

Parameters

- **key** – the key
- **defaultValue** – the value to return if none exists in the table

Returns either the value in the table, or the defaultValue

getFloat (*key: str*, *defaultValue: float = 0.0*) → float
Returns the float at the given key. If this table does not have a value for that position, then the given defaultValue value will be returned.

Parameters

- **key** – the key
- **defaultValue** – the value to return if none exists in the table

Returns either the value in the table, or the defaultValue

static getInstance () → wpilib._wpilib.Preferences
Get the one and only {@link Preferences} object.

Returns pointer to the {@link Preferences}

getInt (*key: str*, *defaultValue: int = 0*) → int
Returns the int at the given key. If this table does not have a value for that position, then the given defaultValue value will be returned.

Parameters

- **key** – the key
- **defaultValue** – the value to return if none exists in the table

Returns either the value in the table, or the defaultValue

getKeys () → List[str]
Returns a vector of all the keys.

Returns a vector of the keys

getLong (*key: str*, *defaultValue: int = 0*) → int
Returns the long (int64_t) at the given key. If this table does not have a value for that position, then the given defaultValue value will be returned.

Parameters

- **key** – the key
- **defaultValue** – the value to return if none exists in the table

Returns either the value in the table, or the defaultValue

getString (*key: str*, *defaultValue: str = ''*) → str
Returns the string at the given key. If this table does not have a value for that position, then the given defaultValue will be returned.

Parameters

- **key** – the key
- **defaultValue** – the value to return if none exists in the table

Returns either the value in the table, or the defaultValue

putBoolean (*key: str*, *value: bool*) → None
Puts the given boolean into the preferences table.

The key may not have any whitespace nor an equals sign.

Parameters

- **key** – the key
- **value** – the value

putDouble (*key: str*, *value: float*) → None
Puts the given double into the preferences table.

The key may not have any whitespace nor an equals sign.

Parameters

- **key** – the key
- **value** – the value

putFloat (*key: str*, *value: float*) → None
Puts the given float into the preferences table.

The key may not have any whitespace nor an equals sign.

Parameters

- **key** – the key
- **value** – the value

putInt (*key: str*, *value: int*) → None
Puts the given int into the preferences table.

The key may not have any whitespace nor an equals sign.

Parameters

- **key** – the key
- **value** – the value

putLong (*key: str*, *value: int*) → None
Puts the given long (int64_t) into the preferences table.

The key may not have any whitespace nor an equals sign.

Parameters

- **key** – the key
- **value** – the value

putString (*key: str*, *value: str*) → None
Puts the given string into the preferences table.

The value may not have quotation marks, nor may the key have any whitespace nor an equals sign.

Parameters

- **key** – the key
- **value** – the value

remove (*key: str*) → None
Remove a preference.

Parameters **key** – the key

removeAll () → None
Remove all preferences.

1.1.59 Relay

class wpilib.**Relay**(*channel: int*, *direction: wpilib._wpilib.Relay.Direction = Direction.kBothDirections*) → None

Bases: *wpilib.MotorSafety*, *wpilib.Sendable*

Class for Spike style relay outputs.

Relays are intended to be connected to spikes or similar relays. The relay channels controls a pair of pins that are either both off, one on, the other on, or both on. This translates into two spike outputs at 0v, one at 12v and one at 0v, one at 0v and the other at 12v, or two spike outputs at 12V. This allows off, full forward, or full reverse control of motors without variable speed. It also allows the two channels (forward and reverse) to be used independently for something that does not care about voltage polarity (like a solenoid).

Relay constructor given a channel.

This code initializes the relay and reserves all resources that need to be locked. Initially the relay is set to both lines at 0v.

Parameters

- **channel** – The channel number (0-3).
- **direction** – The direction that the Relay object will control.

class Direction(*arg0: int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kBothDirections

kForwardOnly

kReverseOnly

kBothDirections = Direction.kBothDirections

kForwardOnly = Direction.kForwardOnly

kReverseOnly = Direction.kReverseOnly

name

(self: handle) -> str

class Value(*arg0: int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kOff

kOn

kForward

kReverse

kForward = Value.kForward

kOff = Value.kOff

kOn = Value.kOn

kReverse = Value.kReverse

name

(self: handle) -> str

get() → wpilib._wpilib.Relay.Value
Get the Relay State

Gets the current state of the relay.

When set to kForwardOnly or kReverseOnly, value is returned as kOn/kOff not kForward/kReverse (per the recommendation in Set).

Returns The current state of the relay as a Relay::Value

getChannel() → int

getDescription(*desc: wpi::raw_ostream*) → None

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

set(*value: wpilib._wpilib.Relay.Value*) → None
Set the relay state.

Valid values depend on which directions of the relay are controlled by the object.

When set to kBothDirections, the relay can be any of the four states: 0v-0v, 0v-12v, 12v-0v, 12v-12v

When set to kForwardOnly or kReverseOnly, you can specify the constant for the direction or you can simply specify kOff and kOn. Using only kOff and kOn is recommended.

Parameters **value** – The state to set the relay.

stopMotor() → None

1.1.60 RobotBase

class wpilib.**RobotBase**() → None
Bases: pybind11_builtins.pybind11_object

Implement a Robot Program framework.

The RobotBase class is intended to be subclassed by a user creating a robot program. Overridden Autonomous() and OperatorControl() methods are called at the appropriate time as the match proceeds. In the current implementation, the Autonomous code will run to completion before the OperatorControl code could start. In the future the Autonomous code might be spawned as a task, then killed at the end of the Autonomous period.

Constructor for a generic robot program.

User code should be placed in the constructor that runs before the Autonomous or Operator Control period starts. The constructor will run to completion before Autonomous is entered.

This must be used to ensure that the communications code starts. In the future it would be nice to put this code into it's own task that loads on boot so ensure that it runs.

ds

endCompetition() → None

getControlState() → Tuple[bool, bool, bool]
More efficient way to determine what state the robot is in.

Returns booleans representing enabled, isautonomous, istest

New in version 2019.2.1.

Note: This function only exists in RobotPy

---

static getThreadId() → std::thread::id
Gets the ID of the main robot thread.

isAutonomous() → bool
Determine if the robot is currently in Autonomous mode.

Returns True if the robot is currently operating Autonomously as determined by the field controls.

isAutonomousEnabled() → bool
Equivalent to calling `isAutonomous() and isEnabled()` but more efficient.

Returns True if the robot is in autonomous mode and is enabled, False otherwise.

New in version 2019.2.1.

Note: This function only exists in RobotPy

---

isDisabled() → bool
Determine if the Robot is currently disabled.

Returns True if the Robot is currently disabled by the field controls.

isEnabled() → bool
Determine if the Robot is currently enabled.

Returns True if the Robot is currently enabled by the field controls.

isNewDataAvailable() → bool
Indicates if new data is available from the driver station.

Returns Has new data arrived over the network since the last time this function was called?

isOperatorControl() → bool
Determine if the robot is currently in Operator Control mode.

Returns True if the robot is currently operating in Tele-Op mode as determined by the field controls.

isOperatorControlEnabled() → bool
Equivalent to calling `isOperatorControl() and isEnabled()` but more efficient.

Returns True if the robot is in operator-controlled mode and is enabled, False otherwise.

New in version 2019.2.1.

Note: This function only exists in RobotPy

---

static isReal() → bool

static isSimulation() → bool

isTest() → bool
Determine if the robot is currently in Test mode.

Returns True if the robot is currently running tests as determined by the field controls.

logger = <Logger robot (WARNING)>

static main(*robot_cls: object*) → object
Starting point for the application

startCompetition() → None

1.1.61 RobotController

class wpilib.**RobotController**
Bases: pybind11_builtins.pybind11_object

static getCANStatus() → wpilib._wpilib.CANStatus

static getCurrent3V3() → float
Get the current output of the 3.3V rail.

Returns The controller 3.3V rail output current value in Amps

static getCurrent5V() → float
Get the current output of the 5V rail.

Returns The controller 5V rail output current value in Amps

static getCurrent6V() → float
Get the current output of the 6V rail.

Returns The controller 6V rail output current value in Amps

static getEnabled3V3() → bool
Get the enabled state of the 3.3V rail. The rail may be disabled due to a controller brownout, a short circuit on the rail, or controller over-voltage.

Returns The controller 3.3V rail enabled value. True for enabled.

static getEnabled5V() → bool
Get the enabled state of the 5V rail. The rail may be disabled due to a controller brownout, a short circuit on the rail, or controller over-voltage.

Returns The controller 5V rail enabled value. True for enabled.

static getEnabled6V() → bool
Get the enabled state of the 6V rail. The rail may be disabled due to a controller brownout, a short circuit on the rail, or controller over-voltage.

Returns The controller 6V rail enabled value. True for enabled.

static getFPGARevision() → int
Return the FPGA Revision number.

The format of the revision is 3 numbers. The 12 most significant bits are the Major Revision. The next 8 bits are the Minor Revision. The 12 least significant bits are the Build Number.

Returns FPGA Revision number.

static getFPGATime() → int
Read the microsecond-resolution timer on the FPGA.

Returns The current time in microseconds according to the FPGA (since FPGA reset).

static getFPGAVersion() → int
Return the FPGA Version number.

For now, expect this to be competition year.

Returns FPGA Version number.

static getFaultCount3V3() → int
Get the count of the total current faults on the 3.3V rail since the controller has booted.

Returns The number of faults

static getFaultCount5V() → int
Get the count of the total current faults on the 5V rail since the controller has booted.

Returns The number of faults

static getFaultCount6V() → int
Get the count of the total current faults on the 6V rail since the controller has booted.

Returns The number of faults.

static getInputCurrent() → float
Get the input current to the robot controller.

Returns The controller input current value in Amps

static getInputVoltage() → float
Get the input voltage to the robot controller.

Returns The controller input voltage value in Volts

static getUserButton() → bool
Get the state of the "USER" button on the roboRIO.

Returns True if the button is currently pressed down

static getVoltage3V3() → float
Get the voltage of the 3.3V rail.

Returns The controller 3.3V rail voltage value in Volts

static getVoltage5V() → float
Get the voltage of the 5V rail.

Returns The controller 5V rail voltage value in Volts

static getVoltage6V() → float
Get the voltage of the 6V rail.

Returns The controller 6V rail voltage value in Volts

static isBrownedOut() → bool
Check if the system is browned out.

Returns True if the system is browned out

static isSysActive() → bool
Check if the FPGA outputs are enabled.

The outputs may be disabled if the robot is disabled or e-stopped, the watchdog has expired, or if the roboRIO browns out.

Returns True if the FPGA outputs are enabled.

1.1.62 RobotState

class wpilib.**RobotState**
Bases: pybind11_builtins.pybind11_object

static isAutonomous() → bool

static isDisabled() → bool

static isEStopped() → bool

static isEnabled() → bool

static isOperatorControl() → bool

static isTest() → bool

1.1.63 SD540

class wpilib.**SD540**(*channel: int*) → None
Bases: *wpilib.PWMSpeedController*

Mindsensors SD540 Speed Controller.

Note that the SD540 uses the following bounds for PWM values. These values should work reasonably well for most controllers, but if users experience issues such as asymmetric behavior around the deadband or inability to saturate the controller in either direction, calibration is recommended. The calibration procedure can be found in the SD540 User Manual available from Mindsensors.

- 2.05ms = full "forward"
- 1.55ms = the "high end" of the deadband range
- 1.50ms = center of the deadband range (off)
- 1.44ms = the "low end" of the deadband range
- 0.94ms = full "reverse"

Constructor for a SD540.

Parameters **channel** – The PWM channel that the SD540 is attached to. 0-9 are on-board, 10-19 are on the MXP port

1.1.64 SPI

class wpilib.**SPI**(*port: wpilib._wpilib.SPI.Port*) → None
Bases: *wpilib.ErrorBase*

SPI bus interface class.

This class is intended to be used by sensor (and other SPI device) drivers. It probably should not be used directly.

Constructor

Parameters **port** – the physical SPI port

class Port(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kOnboardCS0

kOnboardCS1

kOnboardCS2

kOnboardCS3

kMXP

kMXP = Port.kMXP

kOnboardCS0 = Port.kOnboardCS0

kOnboardCS1 = Port.kOnboardCS1

kOnboardCS2 = Port.kOnboardCS2

kOnboardCS3 = Port.kOnboardCS3

name
(self: handle) -> str

configureAutoStall(*port: hal._wpiHal.SPIPort*, *csToSclkTicks: int*, *stallTicks: int*, *pow2BytesPerRead: int*) → None
Configure the Auto SPI Stall time between reads.

Parameters

- **port** – The number of the port to use. 0-3 for Onboard CS0-CS2, 4 for MXP.
- **csToSclkTicks** – the number of ticks to wait before asserting the cs pin
- **stallTicks** – the number of ticks to stall for
- **pow2BytesPerRead** – the number of bytes to read before stalling

forceAutoRead() → None
Force the engine to make a single transfer.

freeAccumulator() → None
Frees the accumulator.

freeAuto() → None
Frees the automatic SPI transfer engine.

getAccumulatorAverage() → float
Read the average of the accumulated value.

Returns The accumulated average value (value / count).

getAccumulatorCount() → int
Read the number of accumulated values.

Read the count of the accumulated values since the accumulator was last Reset().

Returns The number of times samples from the channel were accumulated.

getAccumulatorIntegratedAverage() → float
Read the average of the integrated value. This is the sum of (each value times the time between values), divided by the count.

Returns The average of the integrated value accumulated since the last Reset().

getAccumulatorIntegratedValue() → float
Read the integrated value. This is the sum of (each value * time between values).

Returns The integrated value accumulated since the last Reset().

getAccumulatorLastValue() → int
Read the last value read by the accumulator engine.

getAccumulatorOutput(*value: int*, *count: int*) → None
Read the accumulated value and the number of accumulated values atomically.

This function reads the value and count atomically. This can be used for averaging.

Parameters

- **value** – Pointer to the 64-bit accumulated output.
- **count** – Pointer to the number of accumulation cycles.

getAccumulatorValue() → int
Read the accumulated value.

Returns The 64-bit value accumulated since the last Reset().

getAutoDroppedCount() → int
Get the number of bytes dropped by the automatic SPI transfer engine due to the receive buffer being full.

Returns Number of bytes dropped

initAccumulator(*period: seconds*, *cmd: int*, *xferSize: int*, *validMask: int*, *validValue: int*, *dataShift: int*, *dataSize: int*, *isSigned: bool*, *bigEndian: bool*) → None
Initialize the accumulator.

Parameters

- **period** – Time between reads
- **cmd** – SPI command to send to request data
- **xferSize** – SPI transfer size, in bytes
- **validMask** – Mask to apply to received data for validity checking
- **validData** – After valid_mask is applied, required matching value for validity checking
- **dataShift** – Bit shift to apply to received data to get actual data value
- **dataSize** – Size (in bits) of data field
- **isSigned** – Is data field signed?
- **bigEndian** – Is device big endian?

initAuto(*bufferSize: int*) → None
Initialize automatic SPI transfer engine.

Only a single engine is available, and use of it blocks use of all other chip select usage on the same physical SPI port while it is running.

Parameters **bufferSize** – buffer size in bytes

read(*initiate: bool*, *dataReceived: buffer*) → int
Read a word from the receive FIFO.

Waits for the current transfer to complete if the receive FIFO is empty.

If the receive FIFO is empty, there is no active transfer, and initiate is false, errors.

Parameters **initiate** – If true, this function pushes "0" into the transmit buffer and initiates a transfer. If false, this function assumes that data is already in the receive FIFO from a previous write.

readAutoReceivedData(*buffer: buffer*, *timeout: seconds*) → int
Read data that has been transferred by the automatic SPI transfer engine.

Transfers may be made a byte at a time, so it's necessary for the caller to handle cases where an entire transfer has not been completed.

Each received data sequence consists of a timestamp followed by the received data bytes, one byte per word (in the least significant byte). The length of each received data sequence is the same as the combined size of the data and zeroSize set in SetAutoTransmitData().

Blocks until numToRead words have been read or timeout expires. May be called with numToRead=0 to retrieve how many words are available.

Parameters

- **buffer** – buffer where read words are stored
- **numToRead** – number of words to read
- **timeout** – timeout (ms resolution)

Returns Number of words remaining to be read

resetAccumulator() → None
Resets the accumulator to zero.

setAccumulatorCenter(*center: int*) → None
Set the center value of the accumulator.

The center value is subtracted from each value before it is added to the accumulator. This is used for the center value of devices like gyros and accelerometers to make integration work and to take the device offset into account when integrating.

setAccumulatorDeadband(*deadband: int*) → None
Set the accumulator's deadband.

setAccumulatorIntegratedCenter(*center: float*) → None
Set the center value of the accumulator integrator.

The center value is subtracted from each value*dt before it is added to the integrated value. This is used for the center value of devices like gyros and accelerometers to take the device offset into account when integrating.

setAutoTransmitData() → None
Set the data to be transmitted by the engine.

Up to 16 bytes are configurable, and may be followed by up to 127 zero bytes.

Parameters

- **dataToSend** – data to send (maximum 16 bytes)
- **zeroSize** – number of zeros to send after the data

setChipSelectActiveHigh() → None
Configure the chip select line to be active high.

setChipSelectActiveLow() → None
Configure the chip select line to be active low.

setClockActiveHigh() → None
Configure the clock output line to be active high. This is sometimes called clock polarity low or clock idle low.

setClockActiveLow() → None
Configure the clock output line to be active low. This is sometimes called clock polarity high or clock idle high.

setClockRate(*hz: int*) → None
Configure the rate of the generated clock signal.

The default value is 500,000Hz. The maximum value is 4,000,000Hz.

Parameters **hz** – The clock rate in Hertz.

setLSBFirst() → None
Configure the order that bits are sent and received on the wire to be least significant bit first.

setMSBFirst() → None
Configure the order that bits are sent and received on the wire to be most significant bit first.

setSampleDataOnFalling() → None

setSampleDataOnLeadingEdge() → None
Configure that the data is stable on the leading edge and the data changes on the trailing edge.

setSampleDataOnRising() → None

setSampleDataOnTrailingEdge() → None
Configure that the data is stable on the trailing edge and the data changes on the leading edge.

startAutoRate(*period: seconds*) → None
Start running the automatic SPI transfer engine at a periodic rate.

InitAuto() and SetAutoTransmitData() must be called before calling this function.

Parameters **period** – period between transfers (us resolution)

startAutoTrigger(*source: frc::DigitalSource*, *rising: bool*, *falling: bool*) → None
Start running the automatic SPI transfer engine when a trigger occurs.

InitAuto() and SetAutoTransmitData() must be called before calling this function.

Parameters

- **source** – digital source for the trigger (may be an analog trigger)
- **rising** – trigger on the rising edge
- **falling** – trigger on the falling edge

stopAuto() → None
Stop running the automatic SPI transfer engine.

transaction(*dataToSend: buffer*, *dataReceived: buffer*) → int
Perform a simultaneous read/write transaction with the device

Parameters

- **dataToSend** – The data to be written out to the device
- **dataReceived** – Buffer to receive data from the device
- **size** – The length of the transaction, in bytes

write(*data: buffer*) → int
Write data to the slave device. Blocks until there is space in the output FIFO.

If not running in output only mode, also saves the data received on the MISO input during the transfer into the receive FIFO.

1.1.65 Sendable

class wpilib.**Sendable**() → None
Bases: pybind11_builtins.pybind11_object

Interface for Sendable objects.

initSendable(*builder: frc::SendableBuilder*) → None
Initializes this Sendable object.

Parameters **builder** – sendable builder

1.1.66 SendableBase

class wpilib.**SendableBase**
Bases: *wpilib.Sendable*

1.1.67 SendableBuilder

class wpilib.**SendableBuilder**() → None
Bases: pybind11_builtins.pybind11_object

addBooleanArrayProperty() → None
Add a boolean array property.

Parameters

- **key** – property name
- **getter** – getter function (returns current value)
- **setter** – setter function (sets new value)

addBooleanProperty() → None
Add a boolean property.

Parameters

- **key** – property name
- **getter** – getter function (returns current value)
- **setter** – setter function (sets new value)

addDoubleArrayProperty() → None
Add a double array property.

Parameters

- **key** – property name
- **getter** – getter function (returns current value)
- **setter** – setter function (sets new value)

addDoubleProperty() → None
Add a double property.

Parameters

- **key** – property name
- **getter** – getter function (returns current value)
- **setter** – setter function (sets new value)

addRawProperty() → None
Add a raw property.

Parameters

- **key** – property name
- **getter** – getter function (returns current value)
- **setter** – setter function (sets new value)

addSmallBooleanArrayProperty() → None
Add a boolean array property (SmallVector form).

Parameters

- **key** – property name
- **getter** – getter function (returns current value)
- **setter** – setter function (sets new value)

addSmallDoubleArrayProperty() → None
Add a double array property (SmallVector form).

Parameters

- **key** – property name
- **getter** – getter function (returns current value)
- **setter** – setter function (sets new value)

addSmallRawProperty() → None
Add a raw property (SmallVector form).

Parameters

- **key** – property name
- **getter** – getter function (returns current value)
- **setter** – setter function (sets new value)

addSmallStringArrayProperty() → None
Add a string array property (SmallVector form).

Parameters

- **key** – property name
- **getter** – getter function (returns current value)
- **setter** – setter function (sets new value)

addSmallStringProperty() → None
Add a string property (SmallString form).

Parameters

- **key** – property name
- **getter** – getter function (returns current value)
- **setter** – setter function (sets new value)

addStringArrayProperty() → None
Add a string array property.

Parameters

- **key** – property name
- **getter** – getter function (returns current value)
- **setter** – setter function (sets new value)

addStringProperty() → None
Add a string property.

Parameters

- **key** – property name
- **getter** – getter function (returns current value)
- **setter** – setter function (sets new value)

addValueProperty() → None
Add a NetworkTableValue property.

Parameters

- **key** – property name
- **getter** – getter function (returns current value)
- **setter** – setter function (sets new value)

getEntry(*key: str*) → _pyntcore._ntcore.NetworkTableEntry
Add a property without getters or setters. This can be used to get entry handles for the function called by SetUpdateTable().

Parameters **key** – property name

Returns Network table entry

setActuator(*value: bool*) → None
Set a flag indicating if this sendable should be treated as an actuator. By default this flag is false.

Parameters **value** – true if actuator, false if not

setSafeState() → None
Set the function that should be called to set the Sendable into a safe state. This is called when entering and exiting Live Window mode.

Parameters **func** – function

setSmartDashboardType(*type: str*) → None
Set the string representation of the named data type that will be used by the smart dashboard for this sendable.

Parameters **type** – data type

setUpdateTable() → None
Set the function that should be called to update the network table for things other than properties. Note this function is not passed the network table object; instead it should use the entry handles returned by GetEntry().

Parameters **func** – function

1.1.68 SendableBuilderImpl

class wpilib.**SendableBuilderImpl**() → None
Bases: *wpilib.SendableBuilder*

addBooleanArrayProperty() → None

addBooleanProperty() → None

addDoubleArrayProperty() → None

addDoubleProperty() → None

addRawProperty() → None

addSmallBooleanArrayProperty() → None

addSmallDoubleArrayProperty() → None

addSmallRawProperty() → None

addSmallStringArrayProperty() → None

addSmallStringProperty() → None

addStringArrayProperty() → None

addStringProperty() → None

addValueProperty() → None

clearProperties() → None
Clear properties.

getEntry(*key: str*) → _pyntcore._ntcore.NetworkTableEntry

getTable() → _pyntcore._ntcore.NetworkTable
Get the network table.

Returns The network table

hasTable() → bool
Return whether this sendable has an associated table.

Returns True if it has a table, false if not.

isActuator() → bool
Return whether this sendable should be treated as an actuator.

Returns True if actuator, false if not.

setActuator(*value: bool*) → None

setSafeState() → None

setSmartDashboardType(*type: str*) → None

setTable(*table: _pyntcore._ntcore.NetworkTable*) → None
Set the network table. Must be called prior to any Add* functions being called.

Parameters **table** – Network table

setUpdateTable() → None

startListeners() → None
Hook setters for all properties.

startLiveWindowMode() → None
Start LiveWindow mode by hooking the setters for all properties. Also calls the SafeState function if one was provided.

stopListeners() → None
Unhook setters for all properties.

stopLiveWindowMode() → None
Stop LiveWindow mode by unhooking the setters for all properties. Also calls the SafeState function if one was provided.

updateTable() → None
Update the network table values by calling the getters for all properties.

1.1.69 SendableChooser

class wpilib.**SendableChooser**() → None
Bases: wpilib._wpilib.SendableChooserBase

The SendableChooser class is a useful tool for presenting a selection of options to the SmartDashboard.

For instance, you may wish to be able to select between multiple autonomous modes. You can do this by putting every possible Command you want to run as an autonomous into a SendableChooser and then put it into the SmartDashboard to have a list of options appear on the laptop. Once autonomous starts, simply ask the SendableChooser what the selected value is.

@tparam T The type of values to be stored @see SmartDashboard

addOption(*name: str*, *object: object*) → None

getSelected() → object

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

setDefaultOption(*name: str*, *object: object*) → None

1.1.70 SendableRegistry

class wpilib.**SendableRegistry**
Bases: pybind11_builtins.pybind11_object

The SendableRegistry class is the public interface for registering sensors and actuators for use on dashboards and LiveWindow.

add(**args*, ***kwargs*)
Overloaded function.

1. add(self: wpilib._wpilib.SendableRegistry, sendable: wpilib._wpilib.Sendable, name: str) -> None

Adds an object to the registry.

Parameters

- **sendable** – object to add
- **name** – component name

2. add(self: wpilib._wpilib.SendableRegistry, sendable: wpilib._wpilib.Sendable, moduleType: str, channel: int) -> None

Adds an object to the registry.

Parameters

- **sendable** – object to add
- **moduleType** – A string that defines the module name in the label for the value
- **channel** – The channel number the device is plugged into

3. add(self: wpilib._wpilib.SendableRegistry, sendable: wpilib._wpilib.Sendable, moduleType: str, moduleNumber: int, channel: int) -> None

Adds an object to the registry.

Parameters

- **sendable** – object to add
- **moduleType** – A string that defines the module name in the label for the value
- **moduleNumber** – The number of the particular module type
- **channel** – The channel number the device is plugged into

4. add(self: wpilib._wpilib.SendableRegistry, sendable: wpilib._wpilib.Sendable, subsystem: str, name: str) -> None

Adds an object to the registry.

Parameters

- **sendable** – object to add
- **subsystem** – subsystem name
- **name** – component name

addChild(*parent: wpilib._wpilib.Sendable*, *child: wpilib._wpilib.Sendable*) → None
Adds a child object to an object. Adds the child object to the registry if it's not already present.

Parameters

- **parent** – parent object
- **child** – child object

addLW(**args*, ***kwargs*)
Overloaded function.

1. addLW(self: wpilib._wpilib.SendableRegistry, sendable: wpilib._wpilib.Sendable, name: str) -> None

Adds an object to the registry and LiveWindow.

Parameters

- **sendable** – object to add
- **name** – component name

2. addLW(self: wpilib._wpilib.SendableRegistry, sendable: wpilib._wpilib.Sendable, moduleType: str, channel: int) -> None

Adds an object to the registry and LiveWindow.

Parameters

- **sendable** – object to add
- **moduleType** – A string that defines the module name in the label for the value
- **channel** – The channel number the device is plugged into

3. addLW(self: wpilib._wpilib.SendableRegistry, sendable: wpilib._wpilib.Sendable, moduleType: str, moduleNumber: int, channel: int) -> None

Adds an object to the registry and LiveWindow.

Parameters

- **sendable** – object to add

- **moduleType** – A string that defines the module name in the label for the value
- **moduleNumber** – The number of the particular module type
- **channel** – The channel number the device is plugged into

4. addLW(self: wpilib._wpilib.SendableRegistry, sendable: wpilib._wpilib.Sendable, subsystem: str, name: str) -> None

Adds an object to the registry and LiveWindow.

Parameters

- **sendable** – object to add
- **subsystem** – subsystem name
- **name** – component name

contains(*sendable: wpilib._wpilib.Sendable*) → bool
Determines if an object is in the registry.

Parameters **sendable** – object to check

Returns True if in registry, false if not.

disableLiveWindow(*sendable: wpilib._wpilib.Sendable*) → None
Disables LiveWindow for an object.

Parameters **sendable** – object

enableLiveWindow(*sendable: wpilib._wpilib.Sendable*) → None
Enables LiveWindow for an object.

Parameters **sendable** – object

static getInstance() → wpilib._wpilib.SendableRegistry
Gets an instance of the SendableRegistry class.

This is a singleton to guarantee that there is only a single instance regardless of how many times GetInstance is called.

getName(*sendable: wpilib._wpilib.Sendable*) → str
Gets the name of an object.

Parameters **sendable** – object

Returns Name (empty if object is not in registry)

getSendable(*uid: int*) → wpilib._wpilib.Sendable
Get sendable object for a given unique id.

Parameters **uid** – unique id

Returns sendable object (may be null)

getSubsystem(*sendable: wpilib._wpilib.Sendable*) → str
Gets the subsystem name of an object.

Parameters **sendable** – object

Returns Subsystem name (empty if object is not in registry)

getUniqueId(*sendable: wpilib._wpilib.Sendable*) → int
Get unique id for an object. Since objects can move, use this instead of storing Sendable* directly if ownership is in question.

Parameters **sendable** – object

Returns unique id

publish (*sendableUid: int*, *table: _pyntcore._ntcore.NetworkTable*) → None
Publishes an object in the registry to a network table.

Parameters

- **sendableUid** – sendable unique id
- **table** – network table

remove (*sendable: wpilib._wpilib.Sendable*) → bool
Removes an object from the registry.

Parameters **sendable** – object to remove

Returns true if the object was removed; false if it was not present

setName (**args*, ***kwargs*)
Overloaded function.

1. setName(self: wpilib._wpilib.SendableRegistry, sendable: wpilib._wpilib.Sendable, name: str) -> None

Sets the name of an object.

Parameters

- **sendable** – object
- **name** – name

2. setName(self: wpilib._wpilib.SendableRegistry, sendable: wpilib._wpilib.Sendable, moduleType: str, channel: int) -> None

Sets the name of an object with a channel number.

Parameters

- **sendable** – object
- **moduleType** – A string that defines the module name in the label for the value
- **channel** – The channel number the device is plugged into

3. setName(self: wpilib._wpilib.SendableRegistry, sendable: wpilib._wpilib.Sendable, moduleType: str, moduleNumber: int, channel: int) -> None

Sets the name of an object with a module and channel number.

Parameters

- **sendable** – object
- **moduleType** – A string that defines the module name in the label for the value
- **moduleNumber** – The number of the particular module type
- **channel** – The channel number the device is plugged into

4. setName(self: wpilib._wpilib.SendableRegistry, sendable: wpilib._wpilib.Sendable, subsystem: str, name: str) -> None

Sets both the subsystem name and device name of an object.

Parameters

- **sendable** – object
- **subsystem** – subsystem name
- **name** – device name

setSubsystem(*sendable: wpilib._wpilib.Sendable*, *subsystem: str*) → None
Sets the subsystem name of an object.

Parameters

- **sendable** – object
- **subsystem** – subsystem name

update(*sendableUid: int*) → None
Updates network table information from an object.

Parameters **sendableUid** – sendable unique id

1.1.71 SensorUtil

class wpilib.**SensorUtil**
Bases: pybind11_builtins.pybind11_object

Stores most recent status information as well as containing utility functions for checking channels and error processing.

static checkAnalogInputChannel(*channel: int*) → bool
Check that the analog input number is value.

Verify that the analog input number is one of the legal channel numbers. Channel numbers are 0-based.

Returns Analog channel is valid

static checkAnalogOutputChannel(*channel: int*) → bool
Check that the analog output number is valid.

Verify that the analog output number is one of the legal channel numbers. Channel numbers are 0-based.

Returns Analog channel is valid

static checkDigitalChannel(*channel: int*) → bool
Check that the digital channel number is valid.

Verify that the channel number is one of the legal channel numbers. Channel numbers are 0-based.

Returns Digital channel is valid

static checkPDPChannel(*channel: int*) → bool
Verify that the power distribution channel number is within limits.

Returns PDP channel is valid

static checkPDPModule(*module: int*) → bool
Verify that the PDP module number is within limits. module numbers are 0-based

Returns PDP module is valid

static checkPWMChannel(*channel: int*) → bool
Check that the digital channel number is valid.

Verify that the channel number is one of the legal channel numbers. Channel numbers are 0-based.

Returns PWM channel is valid

static checkRelayChannel(*channel: int*) → bool
Check that the relay channel number is valid.

Verify that the channel number is one of the legal channel numbers. Channel numbers are 0-based.

Returns Relay channel is valid

static checkSolenoidChannel(*channel: int*) → bool
Verify that the solenoid channel number is within limits.

Returns Solenoid channel is valid

static checkSolenoidModule(*moduleNumber: int*) → bool
Check that the solenoid module number is valid. module numbers are 0-based

Returns Solenoid module is valid and present

static getDefaultSolenoidModule() → int
Get the number of the default solenoid module.

Returns The number of the default solenoid module.

kAnalogInputs = 8

kDigitalChannels = 31

kPDPChannels = 16

kPwmChannels = 20

kRelayChannels = 4

kSolenoidChannels = 8

kSolenoidModules = 63

1.1.72 SerialPort

class wpilib.**SerialPort**(**args*, ***kwargs*)
Bases: *wpilib.ErrorBase*

Driver for the RS-232 serial port on the roboRIO.

The current implementation uses the VISA formatted I/O mode. This means that all traffic goes through the fomatted buffers. This allows the intermingled use of Printf(), Scanf(), and the raw buffer accessors Read() and Write().

More information can be found in the NI-VISA User Manual here: http://www.ni.com/pdf/manuals/370423a.pdf and the NI-VISA Programmer's Reference Manual here: http://www.ni.com/pdf/manuals/370132c.pdf

Overloaded function.

1. __init__(self: wpilib._wpilib.SerialPort, baudRate: int, port: wpilib._wpilib.SerialPort.Port = Port.kOnboard, dataBits: int = 8, parity: wpilib._wpilib.SerialPort.Parity = Parity.kParity_None, stopBits: wpilib._wpilib.SerialPort.StopBits = StopBits.kStopBits_One) -> None

Create an instance of a Serial Port class.

Parameters

- **baudRate** – The baud rate to configure the serial port.
- **port** – The physical port to use
- **dataBits** – The number of data bits per transfer. Valid values are between 5 and 8 bits.
- **parity** – Select the type of parity checking to use.
- **stopBits** – The number of stop bits to use as defined by the enum StopBits.

2. __init__(self: wpilib._wpilib.SerialPort, baudRate: int, portName: str, port: wpilib._wpilib.SerialPort.Port = Port.kOnboard, dataBits: int = 8, parity: wpilib._wpilib.SerialPort.Parity = Parity.kParity_None, stopBits: wpilib._wpilib.SerialPort.StopBits = StopBits.kStopBits_One) -> None

Create an instance of a Serial Port class.

Prefer to use the constructor that doesn't take a port name, but in some cases the automatic detection might not work correctly.

Parameters

- **baudRate** – The baud rate to configure the serial port.
- **port** – The physical port to use
- **portName** – The direct port name to use
- **dataBits** – The number of data bits per transfer. Valid values are between 5 and 8 bits.
- **parity** – Select the type of parity checking to use.
- **stopBits** – The number of stop bits to use as defined by the enum StopBits.

class FlowControl(*arg0: int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kFlowControl_None

kFlowControl_XonXoff

kFlowControl_RtsCts

kFlowControl_DtrDsr

kFlowControl_DtrDsr = FlowControl.kFlowControl_DtrDsr

kFlowControl_None = FlowControl.kFlowControl_None

kFlowControl_RtsCts = FlowControl.kFlowControl_RtsCts

kFlowControl_XonXoff = FlowControl.kFlowControl_XonXoff

name

(self: handle) -> str

class Parity(*arg0: int*) → None

Bases: pybind11_builtins.pybind11_object

Members:

kParity_None

kParity_Odd

kParity_Even

kParity_Mark

kParity_Space

kParity_Even = Parity.kParity_Even

kParity_Mark = Parity.kParity_Mark

kParity_None = Parity.kParity_None

kParity_Odd = Parity.kParity_Odd

kParity_Space = Parity.kParity_Space

name
(self: handle) -> str

class Port(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kOnboard

kMXP

kUSB

kUSB1

kUSB2

kMXP = Port.kMXP

kOnboard = Port.kOnboard

kUSB = Port.kUSB

kUSB1 = Port.kUSB

kUSB2 = Port.kUSB2

name
(self: handle) -> str

class StopBits(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kStopBits_One

kStopBits_OnePointFive

kStopBits_Two

kStopBits_One = StopBits.kStopBits_One

kStopBits_OnePointFive = StopBits.kStopBits_OnePointFive

kStopBits_Two = StopBits.kStopBits_Two

name
(self: handle) -> str

class WriteBufferMode(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kFlushOnAccess

kFlushWhenFull

kFlushOnAccess = WriteBufferMode.kFlushOnAccess

kFlushWhenFull = WriteBufferMode.kFlushWhenFull

name
(self: handle) -> str

disableTermination() → None
Disable termination behavior.

enableTermination(*terminator: str = 'n'*) → None
Enable termination and specify the termination character.

Termination is currently only implemented for receive. When the the terminator is recieved, the Read() or Scanf() will return fewer bytes than requested, stopping after the terminator.

Parameters **terminator** – The character to use for termination.

flush() → None
Force the output buffer to be written to the port.

This is used when SetWriteBufferMode() is set to kFlushWhenFull to force a flush before the buffer is full.

getBytesReceived() → int
Get the number of bytes currently available to read from the serial port.

Returns The number of bytes available to read

read(*buffer: buffer*) → int
Read raw bytes out of the buffer.

Parameters

- **buffer** – Pointer to the buffer to store the bytes in.
- **count** – The maximum number of bytes to read.

Returns The number of bytes actually read into the buffer.

reset() → None
Reset the serial port driver to a known state.

Empty the transmit and receive buffers in the device and formatted I/O.

setFlowControl(*flowControl: wpilib._wpilib.SerialPort.FlowControl*) → None
Set the type of flow control to enable on this port.

By default, flow control is disabled.

setReadBufferSize(*size: int*) → None
Specify the size of the input buffer.

Specify the amount of data that can be stored before data from the device is returned to Read or Scanf. If you want data that is recieved to be returned immediately, set this to 1.

It the buffer is not filled before the read timeout expires, all data that has been received so far will be returned.

Parameters **size** – The read buffer size.

setTimeout(*timeout: float*) → None
Configure the timeout of the serial port.

This defines the timeout for transactions with the hardware. It will affect reads and very large writes.

Parameters **timeout** – The number of seconds to to wait for I/O.

setWriteBufferMode(*mode: wpilib._wpilib.SerialPort.WriteBufferMode*) → None
Specify the flushing behavior of the output buffer.

When set to kFlushOnAccess, data is synchronously written to the serial port after each call to either Printf() or Write().

When set to kFlushWhenFull, data will only be written to the serial port when the buffer is full or when Flush() is called.

Parameters **mode** – The write buffer mode.

setWriteBufferSize(*size: int*) → None
Specify the size of the output buffer.

Specify the amount of data that can be stored before being transmitted to the device.

Parameters **size** – The write buffer size.

write(*buffer: buffer*) → int
Write raw bytes to the buffer.

Parameters

- **buffer** – Pointer to the buffer to read the bytes from.
- **count** – The maximum number of bytes to write.

Returns The number of bytes actually written into the port.

1.1.73 Servo

class wpilib.**Servo**(*channel: int*) → None
Bases: *wpilib.PWM*

Standard hobby style servo.

The range parameters default to the appropriate values for the Hitec HS-322HD servo provided in the FIRST Kit of Parts in 2008.

Parameters **channel** – The PWM channel to which the servo is attached. 0-9 are on-board, 10-19 are on the MXP port

get() → float
Get the servo position.

Servo values range from 0.0 to 1.0 corresponding to the range of full left to full right.

Returns Position from 0.0 to 1.0.

getAngle() → float
Get the servo angle.

Assume that the servo angle is linear with respect to the PWM value (big assumption, need to test).

Returns The angle in degrees to which the servo is set.

getMaxAngle() → float
Get the maximum angle of the servo.

Returns The maximum angle of the servo in degrees.

getMinAngle() → float
Get the minimum angle of the servo.

Returns The minimum angle of the servo in degrees.

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

set(*value: float*) → None
Set the servo position.

Servo values range from 0.0 to 1.0 corresponding to the range of full left to full right.

Parameters **value** – Position from 0.0 to 1.0.

setAngle(*angle: float*) → None
Set the servo angle.

Assume that the servo angle is linear with respect to the PWM value (big assumption, need to test).

Servo angles that are out of the supported range of the servo simply "saturate" in that direction. In other words, if the servo has a range of (X degrees to Y degrees) than angles of less than X result in an angle of X being set and angles of more than Y degrees result in an angle of Y being set.

Parameters **degrees** – The angle in degrees to set the servo.

setOffline() → None
Set the servo to offline.

Set the servo raw value to 0 (undriven)

1.1.74 SlewRateLimiter

class wpilib.**SlewRateLimiter**(*rateLimit: units_per_second*, *initialValue: float = 0.0*) → None
Bases: pybind11_builtins.pybind11_object

A class that limits the rate of change of an input value. Useful for implementing voltage, setpoint, and/or output ramps. A slew-rate limit is most appropriate when the quantity being controlled is a velocity or a voltage; when controlling a position, consider using a TrapezoidProfile instead.

@see TrapezoidProfile

Creates a new SlewRateLimiter with the given rate limit and initial value.

Parameters

- **rateLimit** – The rate-of-change limit.
- **initialValue** – The initial value of the input.

calculate(*input: float*) → float
Filters the input to limit its slew rate.

Parameters **input** – The input value whose slew rate is to be limited.

Returns The filtered value, which will not change faster than the slew rate.

reset(*value: float*) → None
Resets the slew rate limiter to the specified value; ignores the rate limit when doing so.

Parameters **value** – The value to reset to.

1.1.75 SmartDashboard

class wpilib.**SmartDashboard**
Bases: pybind11_builtins.pybind11_object

static clearFlags(*key: str*, *flags: int*) → None
Clears flags on the specified key in this table. The key can not be null.

Parameters

- **key** – the key name
- **flags** – the flags to clear (bitmask)

static clearPersistent(*key: str*) → None
Stop making a key's value persistent through program restarts. The key cannot be null.

Parameters **key** – the key name

static containsKey(*key: str*) → bool
Determines whether the given key is in this table.

Parameters **key** – the key to search for

Returns true if the table as a value assigned to the given key

static delete(*key: str*) → None
Deletes the specified key in this table.

Parameters **key** – the key name

static getBoolean(*keyName: str*, *defaultValue: object*) → object
Returns the value at the specified key.

If the key is not found, returns the default value.

Parameters **keyName** – the key

Returns the value

static getBooleanArray(*key: str*, *defaultValue: object*) → object
Returns the boolean array the key maps to.

If the key does not exist or is of different type, it will return the default value.

Parameters

- **key** – The key to look up.
- **defaultValue** – The value to be returned if no value is found.

Returns

the value associated with the given key or the given default value if there is no value associated with the key

@note This makes a copy of the array. If the overhead of this is a concern, use GetValue() instead.

@note The returned array is std::vector<int> instead of std::vector<bool> because std::vector<bool> is special-cased in C++. 0 is false, any non-zero value is true.

static getData(*keyName: str*) → wpilib._wpilib.Sendable
Returns the value at the specified key.

Parameters **keyName** – the key

Returns the value

static getEntry(*key: str*) → _pyntcore._ntcore.NetworkTableEntry
Returns an NT Entry mapping to the specified key

This is useful if an entry is used often, or is read and then modified.

Parameters **key** – the key

Returns the entry for the key

static getFlags(*key: str*) → int
Returns the flags for the specified key.

Parameters **key** – the key name

Returns the flags, or 0 if the key is not defined

static getKeys(*types: int = 0*) → List[str]

Parameters **types** – bitmask of types; 0 is treated as a "don't care".

Returns keys currently in the table

static getNumber(*keyName: str*, *defaultValue: object*) → object
Returns the value at the specified key.

If the key is not found, returns the default value.

Parameters **keyName** – the key

Returns the value

static getNumberArray(*key: str*, *defaultValue: object*) → object
Returns the number array the key maps to.

If the key does not exist or is of different type, it will return the default value.

Parameters

- **key** – The key to look up.
- **defaultValue** – The value to be returned if no value is found.

Returns

the value associated with the given key or the given default value if there is no value associated with the key

@note This makes a copy of the array. If the overhead of this is a concern, use GetValue() instead.

static getRaw(*key: str*, *defaultValue: object*) → object
Returns the raw value (byte array) the key maps to.

If the key does not exist or is of different type, it will return the default value.

Parameters

- **key** – The key to look up.
- **defaultValue** – The value to be returned if no value is found.

Returns

the value associated with the given key or the given default value if there is no value associated with the key

@note This makes a copy of the raw contents. If the overhead of this is a concern, use GetValue() instead.

static getString(*keyName: str*, *defaultValue: object*) → object
Returns the value at the specified key.

If the key is not found, returns the default value.

Parameters **keyName** – the key

Returns the value

static getStringArray(*key: str*, *defaultValue: object*) → object
Returns the string array the key maps to.

If the key does not exist or is of different type, it will return the default value.

Parameters

- **key** – The key to look up.
- **defaultValue** – The value to be returned if no value is found.

Returns

the value associated with the given key or the given default value if there is no value associated with the key

@note This makes a copy of the array. If the overhead of this is a concern, use GetValue() instead.

static getValue(*keyName: str*) → _pyntcore._ntcore.Value
Retrieves the complex value (such as an array) in this table into the complex data object.

Parameters

- **keyName** – the key
- **value** – the object to retrieve the value into

static init() → None

static isPersistent(*key: str*) → bool
Returns whether the value is persistent through program restarts. The key cannot be null.

Parameters **key** – the key name

static postListenerTask(*task: Callable[[], None]*) → None
Posts a task from a listener to the ListenerExecutor, so that it can be run synchronously from the main loop on the next call to {@link SmartDashboard#updateValues()}.

Parameters **task** – The task to run synchronously from the main thread.

static putBoolean(*keyName: str*, *value: bool*) → bool
Maps the specified key to the specified value in this table.

The value can be retrieved by calling the get method with a key that is equal to the original key.

Parameters

- **keyName** – the key
- **value** – the value

Returns False if the table key already exists with a different type

static putBooleanArray(*key: str, value: List[int]*) → bool
Put a boolean array in the table.

Parameters

- **key** – the key to be assigned to
- **value** – the value that will be assigned

Returns

False if the table key already exists with a different type

@note The array must be of int's rather than of bool's because std::vector<bool> is special-cased in C++. 0 is false, any non-zero value is true.

static putData(**args*, ***kwargs*)

Overloaded function.

1. putData(key: str, data: wpilib._wpilib.Sendable) -> None

Maps the specified key to the specified value in this table.

The value can be retrieved by calling the get method with a key that is equal to the original key.

In order for the value to appear in the dashboard, it must be registered with SendableRegistry. WPILib components do this automatically.

Parameters

- **keyName** – the key
- **value** – the value

2. putData(value: wpilib._wpilib.Sendable) -> None

Maps the specified key (where the key is the name of the Sendable) to the specified value in this table.

The value can be retrieved by calling the get method with a key that is equal to the original key.

In order for the value to appear in the dashboard, it must be registered with SendableRegistry. WPILib components do this automatically.

Parameters **value** – the value

static putNumber(*keyName: str*, *value: float*) → bool

Maps the specified key to the specified value in this table.

The value can be retrieved by calling the get method with a key that is equal to the original key.

Parameters

- **keyName** – the key
- **value** – the value

Returns False if the table key already exists with a different type

static putNumberArray(*key: str, value: List[float]*) → bool

Put a number array in the table.

Parameters

- **key** – The key to be assigned to.
- **value** – The value that will be assigned.

Returns False if the table key already exists with a different type

static putRaw(*key: str*, *value: str*) → bool

Put a raw value (byte array) in the table.

Parameters

- **key** – The key to be assigned to.
- **value** – The value that will be assigned.

Returns False if the table key already exists with a different type

static putString(*keyName: str*, *value: str*) → bool
Maps the specified key to the specified value in this table.

The value can be retrieved by calling the get method with a key that is equal to the original key.

Parameters

- **keyName** – the key
- **value** – the value

Returns False if the table key already exists with a different type

static putStringArray(*key: str, value: List[str]*) → bool
Put a string array in the table.

Parameters

- **key** – The key to be assigned to.
- **value** – The value that will be assigned.

Returns False if the table key already exists with a different type

static putValue(*keyName: str*, *value: _pyntcore._ntcore.Value*) → bool
Maps the specified key to the specified complex value (such as an array) in this table.

The value can be retrieved by calling the RetrieveValue method with a key that is equal to the original key.

Parameters

- **keyName** – the key
- **value** – the value

Returns False if the table key already exists with a different type

static setDefaultBoolean(*key: str*, *defaultValue: bool*) → bool
Gets the current value in the table, setting it if it does not exist.

Parameters

- **key** – the key
- **defaultValue** – the default value to set if key doesn't exist.

Returns False if the table key exists with a different type

static setDefaultBooleanArray(*key: str, defaultValue: List[int]*) → bool
Gets the current value in the table, setting it if it does not exist.

Parameters

- **key** – the key
- **defaultValue** – the default value to set if key doesn't exist.

Returns False if the table key exists with a different type

static setDefaultNumber(*key: str*, *defaultValue: float*) → bool
Gets the current value in the table, setting it if it does not exist.

Parameters

- **key** – The key.
- **defaultValue** – The default value to set if key doesn't exist.

Returns False if the table key exists with a different type

static setDefaultNumberArray(*key: str, defaultValue: List[float]*) → bool
Gets the current value in the table, setting it if it does not exist.

Parameters

- **key** – The key.
- **defaultValue** – The default value to set if key doesn't exist.

Returns False if the table key exists with a different type

static setDefaultRaw(*key: str*, *defaultValue: str*) → bool
Gets the current value in the table, setting it if it does not exist.

Parameters

- **key** – The key.
- **defaultValue** – The default value to set if key doesn't exist.

Returns False if the table key exists with a different type

static setDefaultString(*key: str*, *defaultValue: str*) → bool
Gets the current value in the table, setting it if it does not exist.

Parameters

- **key** – the key
- **defaultValue** – the default value to set if key doesn't exist.

Returns False if the table key exists with a different type

static setDefaultStringArray(*key: str, defaultValue: List[str]*) → bool
Gets the current value in the table, setting it if it does not exist.

Parameters

- **key** – The key.
- **defaultValue** – The default value to set if key doesn't exist.

Returns False if the table key exists with a different type

static setDefaultValue(*key: str*, *defaultValue: _pyntcore._ntcore.Value*) → bool
Gets the current value in the table, setting it if it does not exist.

Parameters

- **key** – the key
- **defaultValue** – The default value to set if key doesn't exist.

Returns False if the table key exists with a different type

static setFlags(*key: str*, *flags: int*) → None
Sets flags on the specified key in this table. The key can not be null.

Parameters

- **key** – the key name

- **flags** – the flags to set (bitmask)

static setPersistent(*key: str*) → None
Makes a key's value persistent through program restarts.

Parameters **key** – the key to make persistent

static updateValues() → None
Puts all sendable data to the dashboard.

1.1.76 Solenoid

class wpilib.**Solenoid**(**args*, ***kwargs*)
Bases: *wpilib.SolenoidBase*, *wpilib.Sendable*

Solenoid class for running high voltage Digital Output (PCM).

The Solenoid class is typically used for pneumatics solenoids, but could be used for any device within the current spec of the PCM.

Overloaded function.

1. __init__(self: wpilib._wpilib.Solenoid, channel: int) -> None

Constructor using the default PCM ID (0).

Parameters **channel** – The channel on the PCM to control (0..7).

2. __init__(self: wpilib._wpilib.Solenoid, moduleNumber: int, channel: int) -> None

Constructor.

Parameters

- **moduleNumber** – The CAN ID of the PCM the solenoid is attached to
- **channel** – The channel on the PCM to control (0..7).

get() → bool
Read the current value of the solenoid.

Returns The current value of the solenoid.

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

isBlackListed() → bool
Check if solenoid is blacklisted.

If a solenoid is shorted, it is added to the blacklist and disabled until power cycle, or until faults are cleared.

@see ClearAllPCMStickyFaults()

Returns If solenoid is disabled due to short.

set(*on: bool*) → None
Set the value of a solenoid.

Parameters **on** – Turn the solenoid output off or on.

setPulseDuration(*durationSeconds: float*) → None
Set the pulse duration in the PCM. This is used in conjunction with the startPulse method to allow the PCM to control the timing of a pulse. The timing can be controlled in 0.01 second increments.

Parameters **durationSeconds** – The duration of the pulse, from 0.01 to 2.55 seconds.

@see startPulse()

startPulse() → None
Trigger the PCM to generate a pulse of the duration set in setPulseDuration.

@see setPulseDuration()

1.1.77 SolenoidBase

class wpilib.**SolenoidBase**(*pcmID: int*) → None
Bases: *wpilib.ErrorBase*

SolenoidBase class is the common base class for the Solenoid and DoubleSolenoid classes.

Constructor.

Parameters **moduleNumber** – The CAN PCM ID.

clearAllPCMStickyFaults() → None
Clear ALL sticky faults inside PCM that Compressor is wired to.

If a sticky fault is set, then it will be persistently cleared. Compressor drive maybe momentarily disable while flags are being cleared. Care should be taken to not call this too frequently, otherwise normal compressor functionality may be prevented.

If no sticky faults are set then this call will have no effect.

static clearAllPCMStickyFaultsByModule(*module: int*) → None
Clear ALL sticky faults inside PCM that Compressor is wired to.

If a sticky fault is set, then it will be persistently cleared. Compressor drive maybe momentarily disable while flags are being cleared. Care should be taken to not call this too frequently, otherwise normal compressor functionality may be prevented.

If no sticky faults are set then this call will have no effect.

Parameters **module** – the module to read from

getAll() → int
Read all 8 solenoids as a single byte

Returns The current value of all 8 solenoids on the module.

static getAllByModule(*module: int*) → int
Read all 8 solenoids as a single byte

Parameters **module** – the module to read from

Returns The current value of all 8 solenoids on the module.

getPCMSolenoidBlackList() → int
Reads complete solenoid blacklist for all 8 solenoids as a single byte.

If a solenoid is shorted, it is added to the blacklist and disabled until power cycle, or until faults are cleared. @see ClearAllPCMStickyFaults()

Returns The solenoid blacklist of all 8 solenoids on the module.

static getPCMSolenoidBlackListByModule(*module: int*) → int
Reads complete solenoid blacklist for all 8 solenoids as a single byte.

If a solenoid is shorted, it is added to the blacklist and disabled until power cycle, or until faults are cleared. @see ClearAllPCMStickyFaults()

Parameters **module** – the module to read from

Returns The solenoid blacklist of all 8 solenoids on the module.

getPCMSolenoidVoltageFault() → bool

Returns true if PCM is in fault state : The common highside solenoid voltage rail is too low, most likely a solenoid channel is shorted.

static getPCMSolenoidVoltageFaultByModule(*module: int*) → bool

Parameters **module** – the module to read from

Returns true if PCM is in fault state : The common highside solenoid voltage rail is too low, most likely a solenoid channel is shorted.

getPCMSolenoidVoltageStickyFault() → bool

Returns true if PCM sticky fault is set : The common highside solenoid voltage rail is too low, most likely a solenoid channel is shorted.

static getPCMSolenoidVoltageStickyFaultByModule(*module: int*) → bool

Parameters **module** – the module to read from

Returns true if PCM sticky fault is set : The common highside solenoid voltage rail is too low, most likely a solenoid channel is shorted.

1.1.78 Spark

class wpilib.**Spark**(*channel: int*) → None
Bases: *wpilib.PWMSpeedController*

REV Robotics SPARK Speed Controller.

Note that the SPARK uses the following bounds for PWM values. These values should work reasonably well for most controllers, but if users experience issues such as asymmetric behavior around the deadband or inability to saturate the controller in either direction, calibration is recommended. The calibration procedure can be found in the SPARK User Manual available from REV Robotics.

- 2.003ms = full "forward"
- 1.550ms = the "high end" of the deadband range
- 1.500ms = center of the deadband range (off)
- 1.460ms = the "low end" of the deadband range
- 0.999ms = full "reverse"

Constructor for a SPARK.

Parameters **channel** – The PWM channel that the SPARK is attached to. 0-9 are on-board, 10-19 are on the MXP port

1.1.79 SpeedControllerGroup

class wpilib.**SpeedControllerGroup**(**args*) → None
Bases: *wpilib.Sendable*, *wpilib.interfaces.SpeedController*

disable() → None

get() → float

getInverted() → bool

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

pidWrite(*output: float*) → None

set(*speed: float*) → None

setInverted(*isInverted: bool*) → None

stopMotor() → None

1.1.80 Talon

class wpilib.**Talon**(*channel: int*) → None
Bases: *wpilib.PWMSpeedController*

Cross the Road Electronics (CTRE) Talon and Talon SR Speed Controller.

Note that the Talon uses the following bounds for PWM values. These values should work reasonably well for most controllers, but if users experience issues such as asymmetric behavior around the deadband or inability to saturate the controller in either direction, calibration is recommended. The calibration procedure can be found in the Talon User Manual available from CTRE.

- 2.037ms = full "forward"
- 1.539ms = the "high end" of the deadband range
- 1.513ms = center of the deadband range (off)
- 1.487ms = the "low end" of the deadband range
- 0.989ms = full "reverse"

Constructor for a Talon (original or Talon SR).

Parameters **channel** – The PWM channel number that the Talon is attached to. 0-9 are on-board, 10-19 are on the MXP port

1.1.81 TimedRobot

class wpilib.**TimedRobot**(*period: seconds = 0.02*) → None
Bases: *wpilib.IterativeRobotBase*

TimedRobot implements the IterativeRobotBase robot program framework.

The TimedRobot class is intended to be subclassed by a user creating a robot program.

Periodic() functions from the base class are called on an interval by a Notifier instance.

Constructor for TimedRobot.

Parameters **period** – Period.

endCompetition() → None
Ends the main loop in StartCompetition().

getPeriod() → seconds
Get the time period between calls to Periodic() functions.

kDefaultPeriod = 0.02

startCompetition() → None
Provide an alternate "main loop" via StartCompetition().

1.1.82 Timer

class wpilib.**Timer**() → None
Bases: pybind11_builtins.pybind11_object

A wrapper for the frc::Timer class that returns unit-typed values.

Create a new timer object.

Create a new timer object and reset the time to zero. The timer is initially not running and must be started.

advanceIfElapsed(*period: seconds*) → bool
Check if the period specified has passed and if it has, advance the start time by that period. This is useful to decide if it's time to do periodic work without drifting later by the time it took to get around to checking.

Parameters **period** – The period to check for.

Returns True if the period has passed.

get() → seconds
Get the current time from the timer. If the clock is running it is derived from the current system clock the start time stored in the timer class. If the clock is not running, then return the time when it was last stopped.

Returns Current time value for this timer in seconds

static getFPGATimestamp() → seconds
Return the FPGA system clock time in seconds.

Return the time from the FPGA hardware clock in seconds since the FPGA started. Rolls over after 71 minutes.

Returns Robot running time in seconds.

static getMatchTime() → seconds
Return the approximate match time.

The FMS does not send an official match time to the robots, but does send an approximate match time. The value will count down the time remaining in the current period (auto or teleop).

Warning: This is not an official time (so it cannot be used to dispute ref calls or guarantee that a function will trigger before the match ends).

The Practice Match function of the DS approximates the behavior seen on the field.

Returns Time remaining in current match period (auto or teleop)

hasElapsed(*period: seconds*) → bool
Check if the period specified has passed.

Parameters **seconds** – The period to check.

Returns True if the period has passed.

hasPeriodPassed(*period: seconds*) → bool
Check if the period specified has passed and if it has, advance the start time by that period. This is useful to decide if it's time to do periodic work without drifting later by the time it took to get around to checking.

Parameters **period** – The period to check for.

Returns True if the period has passed.

kRolloverTime = 4294.967296

reset() → None
Reset the timer by setting the time to 0.

Make the timer startTime the current time so new requests will be relative to now.

start() → None
Start the timer running.

Just set the running flag to true indicating that all time requests should be relative to the system clock.

stop() → None
Stop the timer.

This computes the time as of now and clears the running flag, causing all subsequent time requests to be read from the accumulated time rather than looking at the system clock.

1.1.83 Ultrasonic

class wpilib.**Ultrasonic**(**args*, ***kwargs*)
Bases: *wpilib.ErrorBase*, *wpilib.Sendable*, *wpilib.interfaces.PIDSource*

Ultrasonic rangefinder class.

The Ultrasonic rangefinder measures absolute distance based on the round-trip time of a ping generated by the controller. These sensors use two transducers, a speaker and a microphone both tuned to the ultrasonic range. A common ultrasonic sensor, the Daventech SRF04 requires a short pulse to be generated on a digital channel. This causes the chirp to be emitted. A second line becomes high as the ping is transmitted and goes low when the echo is received. The time that the line is high determines the round trip distance (time of flight).

Overloaded function.

1. __init__(self: wpilib._wpilib.Ultrasonic, pingChannel: int, echoChannel: int, units: wpilib._wpilib.Ultrasonic.DistanceUnit = DistanceUnit.kInches) -> None

Create an instance of the Ultrasonic Sensor.

This is designed to support the Daventech SRF04 and Vex ultrasonic sensors.

Parameters

- **pingChannel** – The digital output channel that sends the pulse to initiate the sensor sending the ping.
- **echoChannel** – The digital input channel that receives the echo. The length of time that the echo is high represents the round trip time of the ping, and the distance.
- **units** – The units returned in either kInches or kMilliMeters

2. __init__(self: wpilib._wpilib.Ultrasonic, pingChannel: frc::DigitalOutput, echoChannel: frc::DigitalInput, units: wpilib._wpilib.Ultrasonic.DistanceUnit = DistanceUnit.kInches) -> None

Create an instance of an Ultrasonic Sensor from a DigitalInput for the echo channel and a DigitalOutput for the ping channel.

Parameters

- **pingChannel** – The digital output object that starts the sensor doing a ping. Requires a 10uS pulse to start.
- **echoChannel** – The digital input object that times the return pulse to determine the range.
- **units** – The units returned in either kInches or kMilliMeters

class DistanceUnit(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kInches

kMilliMeters

kInches = DistanceUnit.kInches

kMilliMeters = DistanceUnit.kMilliMeters

name
(self: handle) -> str

getDistanceUnits() → wpilib._wpilib.Ultrasonic.DistanceUnit
Get the current DistanceUnit that is used for the PIDSource base object.

Returns The type of DistanceUnit that is being used.

getRangeInches() → float
Get the range in inches from the ultrasonic sensor.

Returns Range in inches of the target returned from the ultrasonic sensor. If there is no valid value yet, i.e. at least one measurement hasn't completed, then return 0.

getRangeMM() → float
Get the range in millimeters from the ultrasonic sensor.

Returns Range in millimeters of the target returned by the ultrasonic sensor. If there is no valid value yet, i.e. at least one measurement hasn't completed, then return 0.

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

isEnabled() → bool

isRangeValid() → bool
Check if there is a valid range measurement.

The ranges are accumulated in a counter that will increment on each edge of the echo (return) signal. If the count is not at least 2, then the range has not yet been measured, and is invalid.

pidGet() → float
Get the range in the current DistanceUnit for the PIDSource base object.

Returns The range in DistanceUnit

ping() → None
Single ping to ultrasonic sensor.

Send out a single ping to the ultrasonic sensor. This only works if automatic (round robin) mode is disabled. A single ping is sent out, and the counter should count the semi-period when it comes in. The counter is reset to make the current value invalid.

static setAutomaticMode(*enabling: bool*) → None
Turn Automatic mode on/off.

When in Automatic mode, all sensors will fire in round robin, waiting a set time between each sensor.

Parameters **enabling** – Set to true if round robin scheduling should start for all the ultrasonic sensors. This scheduling method assures that the sensors are non-interfering because no two sensors fire at the same time. If another scheduling algorithm is prefered, it can be implemented by pinging the sensors manually and waiting for the results to come back.

setDistanceUnits(*units: wpilib._wpilib.Ultrasonic.DistanceUnit*) → None
Set the current DistanceUnit that should be used for the PIDSource base object.

Parameters **units** – The DistanceUnit that should be used.

setEnabled(*enable: bool*) → None

setPIDSourceType(*pidSource: wpilib.interfaces._interfaces.PIDSourceType*) → None

1.1.84 Victor

class wpilib.**Victor**(*channel: int*) → None
Bases: *wpilib.PWMSpeedController*

Vex Robotics Victor 888 Speed Controller.

The Vex Robotics Victor 884 Speed Controller can also be used with this class but may need to be calibrated per the Victor 884 user manual.

Note that the Victor uses the following bounds for PWM values. These values were determined empirically and optimized for the Victor 888. These values should work reasonably well for Victor 884 controllers as well but if users experience issues such as asymmetric behavior around the deadband or inability to saturate the controller in either direction, calibration is recommended. The calibration procedure can be found in the Victor 884 User Manual available from Vex.

- 2.027ms = full "forward"
- 1.525ms = the "high end" of the deadband range
- 1.507ms = center of the deadband range (off)
- 1.490ms = the "low end" of the deadband range
- 1.026ms = full "reverse"

Constructor for a Victor.

Parameters **channel** – The PWM channel number that the Victor is attached to. 0-9 are on-board, 10-19 are on the MXP port

1.1.85 VictorSP

class wpilib.**VictorSP**(*channel: int*) → None
Bases: *wpilib.PWMSpeedController*

Vex Robotics Victor SP Speed Controller.

Note that the Victor SP uses the following bounds for PWM values. These values should work reasonably well for most controllers, but if users experience issues such as asymmetric behavior around the deadband or inability to saturate the controller in either direction, calibration is recommended. The calibration procedure can be found in the Victor SP User Manual available from Vex.

- 2.004ms = full "forward"
- 1.520ms = the "high end" of the deadband range
- 1.500ms = center of the deadband range (off)
- 1.480ms = the "low end" of the deadband range
- 0.997ms = full "reverse"

Constructor for a Victor SP.

Parameters **channel** – The PWM channel that the VictorSP is attached to. 0-9 are on-board, 10-19 are on the MXP port

1.1.86 Watchdog

class wpilib.**Watchdog**() → None
Bases: pybind11_builtins.pybind11_object

A class that's a wrapper around a watchdog timer.

When the timer expires, a message is printed to the console and an optional user-provided callback is invoked.

The watchdog is initialized disabled, so the user needs to call Enable() before use.

Watchdog constructor.

Parameters

- **timeout** – The watchdog's timeout in seconds with microsecond resolution.
- **callback** – This function is called when the timeout expires.

addEpoch(*epochName: str*) → None
Adds time since last epoch to the list printed by PrintEpochs().

Epochs are a way to partition the time elapsed so that when overruns occur, one can determine which parts of an operation consumed the most time.

Parameters **epochName** – The name to associate with the epoch.

disable() → None
Disables the watchdog timer.

enable() → None
Enables the watchdog timer.

getTime() → float
Returns the time in seconds since the watchdog was last fed.

getTimeout() → float
Returns the watchdog's timeout in seconds.

isExpired() → bool
Returns true if the watchdog timer has expired.

printEpochs() → None
Prints list of epochs added so far and their times.

reset() → None
Resets the watchdog timer.

This also enables the timer if it was previously disabled.

setTimeout(*timeout: seconds*) → None
Sets the watchdog's timeout.

Parameters **timeout** – The watchdog's timeout in seconds with microsecond resolution.

suppressTimeoutMessage(*suppress: bool*) → None
Enable or disable suppression of the generic timeout message.

This may be desirable if the user-provided callback already prints a more specific message.

Parameters **suppress** – Whether to suppress generic timeout message.

1.1.87 XboxController

class wpilib.**XboxController**(*port: int*) → None
Bases: *wpilib.interfaces.GenericHID*

Handle input from Xbox 360 or Xbox One controllers connected to the Driver Station.

This class handles Xbox input that comes from the Driver Station. Each time a value is requested the most recent value is returned. There is a single class instance for each controller and the mapping of ports to hardware buttons depends on the code in the Driver Station.

Construct an instance of an Xbox controller.

The controller index is the USB port on the Driver Station.

Parameters **port** – The port on the Driver Station that the controller is plugged into (0-5).

class Axis(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kLeftX

kRightX

kLeftY

kRightY

kLeftTrigger

kRightTrigger

kLeftTrigger = Axis.kLeftTrigger

kLeftX = Axis.kLeftX

kLeftY = Axis.kLeftY

kRightTrigger = Axis.kRightTrigger

kRightX = Axis.kRightX

kRightY = Axis.kRightY

name
(self: handle) -> str

class Button(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kBumperLeft

kBumperRight

kStickLeft

kStickRight

kA

kB

kX

kY

kBack

kStart

kA = Button.kA

kB = Button.kB

kBack = Button.kBack

kBumperLeft = Button.kBumperLeft

kBumperRight = Button.kBumperRight

kStart = Button.kStart

kStickLeft = Button.kStickLeft

kStickRight = Button.kStickRight

kX = Button.kX

kY = Button.kY

name
(self: handle) -> str

getAButton() → bool
Read the value of the A button on the controller.

Returns The state of the button.

getAButtonPressed() → bool
Whether the A button was pressed since the last check.

Returns Whether the button was pressed since the last check.

getAButtonReleased() → bool
Whether the A button was released since the last check.

Returns Whether the button was released since the last check.

getBButton() → bool
Read the value of the B button on the controller.

Returns The state of the button.

getBButtonPressed() → bool
Whether the B button was pressed since the last check.

Returns Whether the button was pressed since the last check.

getBButtonReleased() → bool
Whether the B button was released since the last check.

Returns Whether the button was released since the last check.

getBackButton() → bool
Whether the Y button was released since the last check.

Returns Whether the button was released since the last check.

getBackButtonPressed() → bool
Whether the back button was pressed since the last check.

Returns Whether the button was pressed since the last check.

getBackButtonReleased() → bool
Whether the back button was released since the last check.

Returns Whether the button was released since the last check.

getBumper(*hand: wpilib.interfaces._interfaces.GenericHID.Hand*) → bool
Read the value of the bumper button on the controller.

Parameters **hand** – Side of controller whose value should be returned.

getBumperPressed(*hand: wpilib.interfaces._interfaces.GenericHID.Hand*) → bool
Whether the bumper was pressed since the last check.

Parameters **hand** – Side of controller whose value should be returned.

Returns Whether the button was pressed since the last check.

getBumperReleased(*hand: wpilib.interfaces._interfaces.GenericHID.Hand*) → bool
Whether the bumper was released since the last check.

Parameters **hand** – Side of controller whose value should be returned.

Returns Whether the button was released since the last check.

getStartButton() → bool
Read the value of the start button on the controller.

Parameters **hand** – Side of controller whose value should be returned.

Returns The state of the button.

getStartButtonPressed() → bool
Whether the start button was pressed since the last check.

Returns Whether the button was pressed since the last check.

getStartButtonReleased() → bool
Whether the start button was released since the last check.

Returns Whether the button was released since the last check.

getStickButton(*hand: wpilib.interfaces._interfaces.GenericHID.Hand*) → bool
Read the value of the stick button on the controller.

Parameters **hand** – Side of controller whose value should be returned.

Returns The state of the button.

getStickButtonPressed(*hand: wpilib.interfaces._interfaces.GenericHID.Hand*) → bool
Whether the stick button was pressed since the last check.

Parameters **hand** – Side of controller whose value should be returned.

Returns Whether the button was pressed since the last check.

getStickButtonReleased(*hand: wpilib.interfaces._interfaces.GenericHID.Hand*) → bool
Whether the stick button was released since the last check.

Parameters **hand** – Side of controller whose value should be returned.

Returns Whether the button was released since the last check.

getTriggerAxis(*hand: wpilib.interfaces._interfaces.GenericHID.Hand*) → float
Get the trigger axis value of the controller.

Parameters **hand** – Side of controller whose value should be returned.

getX(*hand: wpilib.interfaces._interfaces.GenericHID.Hand*) → float
Get the X axis value of the controller.

Parameters **hand** – Side of controller whose value should be returned.

getXButton() → bool
Read the value of the X button on the controller.

Returns The state of the button.

getXButtonPressed() → bool
Whether the X button was pressed since the last check.

Returns Whether the button was pressed since the last check.

getXButtonReleased() → bool
Whether the X button was released since the last check.

Returns Whether the button was released since the last check.

getY(*hand: wpilib.interfaces._interfaces.GenericHID.Hand*) → float
Get the Y axis value of the controller.

Parameters **hand** – Side of controller whose value should be returned.

getYButton() → bool
Read the value of the Y button on the controller.

Returns The state of the button.

getYButtonPressed() → bool
Whether the Y button was pressed since the last check.

Returns Whether the button was pressed since the last check.

getYButtonReleased() → bool
Whether the Y button was released since the last check.

Returns Whether the button was released since the last check.

1.2 wpilib.controller Package

wpilib.controller.ArmFeedforward(*args, **kwargs)	A helper class that computes feedforward outputs for a simple arm (modeled as a motor acting against the force of gravity on a beam suspended at an angle).
wpilib.controller.ElevatorFeedforward(*args, ...)	A helper class that computes feedforward outputs for a simple elevator (modeled as a motor acting against the force of gravity).
wpilib.controller.PIDController(self, Kp, ...)	Implements a PID control loop.
wpilib.controller.ProfiledPIDController(...)	Implements a PID control loop whose setpoint is constrained by a trapezoid profile.
wpilib.controller.RamseteController(*args, ...)	Ramsete is a nonlinear time-varying feedback controller for unicycle models that drives the model to a desired pose along a two-dimensional trajectory.
wpilib.controller.SimpleMotorFeedforward(...)	A helper class that computes feedforward voltages for a simple permanent-magnet DC motor.

Continued on next page

Table 2 – continued from previous page

wpilib.controller.SimpleMotorFeedforwardMeters(...)	A helper class that computes feedforward voltages for a simple permanent-magnet DC motor.

1.2.1 ArmFeedforward

class wpilib.controller.**ArmFeedforward**(**args*, ***kwargs*)

Bases: pybind11_builtins.pybind11_object

A helper class that computes feedforward outputs for a simple arm (modeled as a motor acting against the force of gravity on a beam suspended at an angle).

Overloaded function.

1. __init__(self: wpilib.controller._controller.ArmFeedforward) -> None
2. __init__(self: wpilib.controller._controller.ArmFeedforward, kS: volts, kCos: volts, kV: volt_seconds_per_radian, kA: volt_seconds_squared_per_radian = 0.0) -> None

Creates a new ArmFeedforward with the specified gains.

Parameters

- **kS** – The static gain, in volts.
- **kCos** – The gravity gain, in volts.
- **kV** – The velocity gain, in volt seconds per radian.
- **kA** – The acceleration gain, in volt seconds^2 per radian.

calculate(*angle: radians*, *velocity: radians_per_second*, *acceleration: radians_per_second_squared = 0.0*) → volts

Calculates the feedforward from the gains and setpoints.

Parameters

- **angle** – The angle setpoint, in radians.
- **velocity** – The velocity setpoint, in radians per second.
- **acceleration** – The acceleration setpoint, in radians per second^2.

Returns The computed feedforward, in volts.

kA

kCos

kS

kV

maxAchievableAcceleration(*maxVoltage: volts*, *angle: radians*, *velocity: radians_per_second*) → radians_per_second_squared

Calculates the maximum achievable acceleration given a maximum voltage supply, a position, and a velocity. Useful for ensuring that velocity and acceleration constraints for a trapezoidal profile are simultaneously achievable - enter the velocity constraint, and this will give you a simultaneously-achievable acceleration constraint.

Parameters

- **maxVoltage** – The maximum voltage that can be supplied to the arm.
- **angle** – The angle of the arm

- **velocity** – The velocity of the arm.

Returns The maximum possible acceleration at the given velocity and angle.

maxAchievableVelocity(*maxVoltage: volts*, *angle: radians*, *acceleration: radians_per_second_squared*) → radians_per_second

Calculates the maximum achievable velocity given a maximum voltage supply, a position, and an acceleration. Useful for ensuring that velocity and acceleration constraints for a trapezoidal profile are simultaneously achievable - enter the acceleration constraint, and this will give you a simultaneously-achievable velocity constraint.

Parameters

- **maxVoltage** – The maximum voltage that can be supplied to the arm.
- **angle** – The angle of the arm
- **acceleration** – The acceleration of the arm.

Returns The maximum possible velocity at the given acceleration and angle.

minAchievableAcceleration(*maxVoltage: volts*, *angle: radians*, *velocity: radians_per_second*) → radians_per_second_squared

Calculates the minimum achievable acceleration given a maximum voltage supply, a position, and a velocity. Useful for ensuring that velocity and acceleration constraints for a trapezoidal profile are simultaneously achievable - enter the velocity constraint, and this will give you a simultaneously-achievable acceleration constraint.

Parameters

- **maxVoltage** – The maximum voltage that can be supplied to the arm.
- **angle** – The angle of the arm
- **velocity** – The velocity of the arm.

Returns The minimum possible acceleration at the given velocity and angle.

minAchievableVelocity(*maxVoltage: volts*, *angle: radians*, *acceleration: radians_per_second_squared*) → radians_per_second

Calculates the minimum achievable velocity given a maximum voltage supply, a position, and an acceleration. Useful for ensuring that velocity and acceleration constraints for a trapezoidal profile are simultaneously achievable - enter the acceleration constraint, and this will give you a simultaneously-achievable velocity constraint.

Parameters

- **maxVoltage** – The maximum voltage that can be supplied to the arm.
- **angle** – The angle of the arm
- **acceleration** – The acceleration of the arm.

Returns The minimum possible velocity at the given acceleration and angle.

1.2.2 ElevatorFeedforward

class wpilib.controller.**ElevatorFeedforward**(**args*, ***kwargs*)

Bases: pybind11_builtins.pybind11_object

A helper class that computes feedforward outputs for a simple elevator (modeled as a motor acting against the force of gravity).

Overloaded function.

1. __init__(self: wpilib.controller._controller.ElevatorFeedforward) -> None
2. __init__(self: wpilib.controller._controller.ElevatorFeedforward, kS: volts, kG: volts, kV: volt_seconds, kA: volt_seconds_squared = 0.0) -> None

Creates a new ElevatorFeedforward with the specified gains.

Parameters

- **kS** – The static gain, in volts.
- **kG** – The gravity gain, in volts.
- **kV** – The velocity gain, in volt seconds per distance.
- **kA** – The acceleration gain, in volt seconds^2 per distance.

calculate(*velocity: units_per_second*, *acceleration: units_per_second_squared = 0.0*) → volts

Calculates the feedforward from the gains and setpoints.

Parameters

- **velocity** – The velocity setpoint, in distance per second.
- **acceleration** – The acceleration setpoint, in distance per second^2.

Returns The computed feedforward, in volts.

kA

kG

kS

kV

maxAchievableAcceleration(*maxVoltage: volts*, *velocity: units_per_second*) → units_per_second_squared

Calculates the maximum achievable acceleration given a maximum voltage supply and a velocity. Useful for ensuring that velocity and acceleration constraints for a trapezoidal profile are simultaneously achievable - enter the velocity constraint, and this will give you a simultaneously-achievable acceleration constraint.

Parameters

- **maxVoltage** – The maximum voltage that can be supplied to the elevator.
- **velocity** – The velocity of the elevator.

Returns The maximum possible acceleration at the given velocity.

maxAchievableVelocity(*maxVoltage: volts*, *acceleration: units_per_second_squared*) → units_per_second

Calculates the maximum achievable velocity given a maximum voltage supply and an acceleration. Useful for ensuring that velocity and acceleration constraints for a trapezoidal profile are simultaneously achievable - enter the acceleration constraint, and this will give you a simultaneously-achievable velocity constraint.

Parameters

- **maxVoltage** – The maximum voltage that can be supplied to the elevator.
- **acceleration** – The acceleration of the elevator.

Returns The maximum possible velocity at the given acceleration.

minAchievableAcceleration(*maxVoltage: volts*, *velocity: units_per_second*) → units_per_second_squared

Calculates the minimum achievable acceleration given a maximum voltage supply and a velocity. Useful for ensuring that velocity and acceleration constraints for a trapezoidal profile are simultaneously achievable - enter the velocity constraint, and this will give you a simultaneously-achievable acceleration constraint.

Parameters

- **maxVoltage** – The maximum voltage that can be supplied to the elevator.
- **velocity** – The velocity of the elevator.

Returns The minimum possible acceleration at the given velocity.

minAchievableVelocity(*maxVoltage: volts*, *acceleration: units_per_second_squared*) → units_per_second

Calculates the minimum achievable velocity given a maximum voltage supply and an acceleration. Useful for ensuring that velocity and acceleration constraints for a trapezoidal profile are simultaneously achievable - enter the acceleration constraint, and this will give you a simultaneously-achievable velocity constraint.

Parameters

- **maxVoltage** – The maximum voltage that can be supplied to the elevator.
- **acceleration** – The acceleration of the elevator.

Returns The minimum possible velocity at the given acceleration.

1.2.3 PIDController

class wpilib.controller.**PIDController**(*Kp: float*, *Ki: float*, *Kd: float*, *period: seconds = 0.02*) → None

Bases: *wpilib.Sendable*

Implements a PID control loop.

Allocates a PIDController with the given constants for Kp, Ki, and Kd.

Parameters

- **Kp** – The proportional coefficient.
- **Ki** – The integral coefficient.
- **Kd** – The derivative coefficient.
- **period** – The period between controller updates in seconds. The default is 20 milliseconds.

atSetpoint() → bool

Returns true if the error is within the tolerance of the error.

This will return false until at least one input value has been computed.

calculate(**args*, ***kwargs*)

Overloaded function.

1. calculate(self: wpilib.controller._controller.PIDController, measurement: float) -> float

Returns the next output of the PID controller.

Parameters **measurement** – The current measurement of the process variable.

2. calculate(self: wpilib.controller._controller.PIDController, measurement: float, setpoint: float) -> float

Returns the next output of the PID controller.

Parameters

- **measurement** – The current measurement of the process variable.
- **setpoint** – The new setpoint of the controller.

disableContinuousInput() → None
Disables continuous input.

enableContinuousInput(*minimumInput: float*, *maximumInput: float*) → None
Enables continuous input.

Rather then using the max and min input range as constraints, it considers them to be the same point and automatically calculates the shortest route to the setpoint.

Parameters

- **minimumInput** – The minimum value expected from the input.
- **maximumInput** – The maximum value expected from the input.

getD() → float
Gets the differential coefficient.

Returns differential coefficient

getI() → float
Gets the integral coefficient.

Returns integral coefficient

getP() → float
Gets the proportional coefficient.

Returns proportional coefficient

getPeriod() → seconds
Gets the period of this controller.

Returns The period of the controller.

getPositionError() → float
Returns the difference between the setpoint and the measurement.

getSetpoint() → float
Returns the current setpoint of the PIDController.

Returns The current setpoint.

getVelocityError() → float
Returns the velocity error.

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

reset() → None
Reset the previous error, the integral term, and disable the controller.

setD(*Kd: float*) → None
Sets the differential coefficient of the PID controller gain.

Parameters **Kd** – differential coefficient

setI (*Ki: float*) → None
Sets the integral coefficient of the PID controller gain.

Parameters **Ki** – integral coefficient

setIntegratorRange (*minimumIntegral: float*, *maximumIntegral: float*) → None
Sets the minimum and maximum values for the integrator.

When the cap is reached, the integrator value is added to the controller output rather than the integrator value times the integral gain.

Parameters

- **minimumIntegral** – The minimum value of the integrator.
- **maximumIntegral** – The maximum value of the integrator.

setP (*Kp: float*) → None
Sets the proportional coefficient of the PID controller gain.

Parameters **Kp** – proportional coefficient

setPID (*Kp: float*, *Ki: float*, *Kd: float*) → None
Sets the PID Controller gain parameters.

Sets the proportional, integral, and differential coefficients.

Parameters

- **Kp** – Proportional coefficient
- **Ki** – Integral coefficient
- **Kd** – Differential coefficient

setSetpoint (*setpoint: float*) → None
Sets the setpoint for the PIDController.

Parameters **setpoint** – The desired setpoint.

setTolerance (*positionTolerance: float*, *velocityTolerance: float = inf*) → None
Sets the error which is considered tolerable for use with AtSetpoint().

Parameters

- **positionTolerance** – Position error which is tolerable.
- **velociytTolerance** – Velocity error which is tolerable.

1.2.4 ProfiledPIDController

class wpilib.controller.**ProfiledPIDController** (*Kp: float*, *Ki: float*, *Kd: float*, *constraints: wpilib.trajectory._trajectory.TrapezoidProfile.Constraints*, *period: seconds = 0.02*) → None

Bases: *wpilib.Sendable*

Implements a PID control loop whose setpoint is constrained by a trapezoid profile.

Allocates a ProfiledPIDController with the given constants for Kp, Ki, and Kd. Users should call reset() when they first start running the controller to avoid unwanted behavior.

Parameters

- **Kp** – The proportional coefficient.
- **Ki** – The integral coefficient.

- **Kd** – The derivative coefficient.
- **constraints** – Velocity and acceleration constraints for goal.
- **period** – The period between controller updates in seconds. The default is 20 milliseconds.

atGoal() → bool
Returns true if the error is within the tolerance of the error.

This will return false until at least one input value has been computed.

atSetpoint() → bool
Returns true if the error is within the tolerance of the error.

Currently this just reports on target as the actual value passes through the setpoint. Ideally it should be based on being within the tolerance for some period of time.

This will return false until at least one input value has been computed.

calculate(**args*, ***kwargs*)
Overloaded function.

1. calculate(self: wpilib.controller._controller.ProfiledPIDController, measurement: float) -> float

Returns the next output of the PID controller.

Parameters **measurement** – The current measurement of the process variable.

2. calculate(self: wpilib.controller._controller.ProfiledPIDController, measurement: float, goal: wpilib.trajectory._trajectory.TrapezoidProfile.State) -> float

Returns the next output of the PID controller.

Parameters

- **measurement** – The current measurement of the process variable.
- **goal** – The new goal of the controller.

3. calculate(self: wpilib.controller._controller.ProfiledPIDController, measurement: float, goal: float) -> float

Returns the next output of the PID controller.

Parameters

- **measurement** – The current measurement of the process variable.
- **goal** – The new goal of the controller.

4. calculate(self: wpilib.controller._controller.ProfiledPIDController, measurement: float, goal: float, constraints: wpilib.trajectory._trajectory.TrapezoidProfile.Constraints) -> float

Returns the next output of the PID controller.

Parameters

- **measurement** – The current measurement of the process variable.
- **goal** – The new goal of the controller.
- **constraints** – Velocity and acceleration constraints for goal.

disableContinuousInput() → None
Disables continuous input.

enableContinuousInput(*minimumInput: float*, *maximumInput: float*) → None
Enables continuous input.

Rather then using the max and min input range as constraints, it considers them to be the same point and automatically calculates the shortest route to the setpoint.

Parameters

- **minimumInput** – The minimum value expected from the input.
- **maximumInput** – The maximum value expected from the input.

getD() → float
Gets the differential coefficient.

Returns differential coefficient

getGoal() → wpilib.trajectory._trajectory.TrapezoidProfile.State
Gets the goal for the ProfiledPIDController.

getI() → float
Gets the integral coefficient.

Returns integral coefficient

getP() → float
Gets the proportional coefficient.

Returns proportional coefficient

getPeriod() → seconds
Gets the period of this controller.

Returns The period of the controller.

getPositionError() → float
Returns the difference between the setpoint and the measurement.

Returns The error.

getSetpoint() → wpilib.trajectory._trajectory.TrapezoidProfile.State
Returns the current setpoint of the ProfiledPIDController.

Returns The current setpoint.

getVelocityError() → units_per_second
Returns the change in error per second.

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

reset(**args*, ***kwargs*)
Overloaded function.

1. reset(self: wpilib.controller._controller.ProfiledPIDController, measurement: wpilib.trajectory._trajectory.TrapezoidProfile.State) -> None

Reset the previous error and the integral term.

Parameters **measurement** – The current measured State of the system.

2. reset(self: wpilib.controller._controller.ProfiledPIDController, measuredPosition: float, measuredVelocity: units_per_second) -> None

Reset the previous error and the integral term.

Parameters

- **measuredPosition** – The current measured position of the system.
- **measuredVelocity** – The current measured velocity of the system.

3. reset(self: wpilib.controller._controller.ProfiledPIDController, measuredPosition: float) -> None

Reset the previous error and the integral term.

Parameters **measuredPosition** – The current measured position of the system. The velocity is assumed to be zero.

setConstraints(*constraints: wpilib.trajectory._trajectory.TrapezoidProfile.Constraints*) → None
Set velocity and acceleration constraints for goal.

Parameters **constraints** – Velocity and acceleration constraints for goal.

setD(*Kd: float*) → None
Sets the differential coefficient of the PID controller gain.

Parameters **Kd** – differential coefficient

setGoal(**args*, ***kwargs*)
Overloaded function.

1. setGoal(self: wpilib.controller._controller.ProfiledPIDController, goal: wpilib.trajectory._trajectory.TrapezoidProfile.State) -> None

Sets the goal for the ProfiledPIDController.

Parameters **goal** – The desired unprofiled setpoint.

2. setGoal(self: wpilib.controller._controller.ProfiledPIDController, goal: float) -> None

Sets the goal for the ProfiledPIDController.

Parameters **goal** – The desired unprofiled setpoint.

setI(*Ki: float*) → None
Sets the integral coefficient of the PID controller gain.

Parameters **Ki** – integral coefficient

setIntegratorRange(*minimumIntegral: float*, *maximumIntegral: float*) → None
Sets the minimum and maximum values for the integrator.

When the cap is reached, the integrator value is added to the controller output rather than the integrator value times the integral gain.

Parameters

- **minimumIntegral** – The minimum value of the integrator.
- **maximumIntegral** – The maximum value of the integrator.

setP(*Kp: float*) → None
Sets the proportional coefficient of the PID controller gain.

Parameters **Kp** – proportional coefficient

setPID (*Kp: float*, *Ki: float*, *Kd: float*) → None
Sets the PID Controller gain parameters.

Sets the proportional, integral, and differential coefficients.

Parameters

- **Kp** – Proportional coefficient
- **Ki** – Integral coefficient
- **Kd** – Differential coefficient

setTolerance (*positionTolerance: float*, *velocityTolerance: units_per_second = inf*) → None
Sets the error which is considered tolerable for use with AtSetpoint().

Parameters

- **positionTolerance** – Position error which is tolerable.
- **velocityTolerance** – Velocity error which is tolerable.

1.2.5 RamseteController

class wpilib.controller.**RamseteController** (**args*, ***kwargs*)
Bases: pybind11_builtins.pybind11_object

Ramsete is a nonlinear time-varying feedback controller for unicycle models that drives the model to a desired pose along a two-dimensional trajectory. Why would we need a nonlinear control law in addition to the linear ones we have used so far like PID? If we use the original approach with PID controllers for left and right position and velocity states, the controllers only deal with the local pose. If the robot deviates from the path, there is no way for the controllers to correct and the robot may not reach the desired global pose. This is due to multiple endpoints existing for the robot which have the same encoder path arc lengths.

Instead of using wheel path arc lengths (which are in the robot's local coordinate frame), nonlinear controllers like pure pursuit and Ramsete use global pose. The controller uses this extra information to guide a linear reference tracker like the PID controllers back in by adjusting the references of the PID controllers.

The paper "Control of Wheeled Mobile Robots: An Experimental Overview" describes a nonlinear controller for a wheeled vehicle with unicycle-like kinematics; a global pose consisting of x, y, and theta; and a desired pose consisting of x_d, y_d, and theta_d. We call it Ramsete because that's the acronym for the title of the book it came from in Italian ("Robotica Articolata e Mobile per i SErvizi e le TEcnologie").

See <https://file.tavsys.net/control/controls-engineering-in-frc.pdf> section on Ramsete unicycle controller for a derivation and analysis.

Overloaded function.

1. __init__(self: wpilib.controller._controller.RamseteController, b: float, zeta: float) -> None

Construct a Ramsete unicycle controller.

Parameters

- **b** – Tuning parameter (b > 0) for which larger values make convergence more aggressive like a proportional term.
- **zeta** – Tuning parameter (0 < zeta < 1) for which larger values provide more damping in response.

2. __init__(self: wpilib.controller._controller.RamseteController) -> None

Construct a Ramsete unicycle controller. The default arguments for b and zeta of 2.0 and 0.7 have been well-tested to produce desireable results.

atReference() → bool
Returns true if the pose error is within tolerance of the reference.

calculate(**args*, ***kwargs*)
Overloaded function.

1. calculate(self: wpilib.controller._controller.RamseteController, currentPose: wpilib.geometry._geometry.Pose2d, poseRef: wpilib.geometry._geometry.Pose2d, linearVelocityRef: meters_per_second, angularVelocityRef: radians_per_second) -> wpilib.kinematics._kinematics.ChassisSpeeds

Returns the next output of the Ramsete controller.

The reference pose, linear velocity, and angular velocity should come from a drivetrain trajectory.

Parameters

- **currentPose** – The current pose.
- **poseRef** – The desired pose.
- **linearVelocityRef** – The desired linear velocity.
- **angularVelocityRef** – The desired angular velocity.

2. calculate(self: wpilib.controller._controller.RamseteController, currentPose: wpilib.geometry._geometry.Pose2d, desiredState: wpilib.trajectory._trajectory.Trajectory.State) -> wpilib.kinematics._kinematics.ChassisSpeeds

Returns the next output of the Ramsete controller.

The reference pose, linear velocity, and angular velocity should come from a drivetrain trajectory.

Parameters

- **currentPose** – The current pose.
- **desiredState** – The desired pose, linear velocity, and angular velocity from a trajectory.

setEnabled(*enabled: bool*) → None
Enables and disables the controller for troubleshooting purposes.

Parameters **enabled** – If the controller is enabled or not.

setTolerance(*poseTolerance: wpilib.geometry._geometry.Pose2d*) → None
Sets the pose error which is considered tolerable for use with AtReference().

Parameters **poseTolerance** – Pose error which is tolerable.

1.2.6 SimpleMotorFeedforward

class wpilib.controller.**SimpleMotorFeedforward**(**args*, ***kwargs*)
Bases: pybind11_builtins.pybind11_object

A helper class that computes feedforward voltages for a simple permanent-magnet DC motor.

Overloaded function.

1. __init__(self: wpilib.controller._controller.SimpleMotorFeedforward) -> None

2. __init__(self: wpilib.controller._controller.SimpleMotorFeedforward, kS: volts, kV: volt_seconds, kA: volt_seconds_squared = 0.0) -> None

Creates a new SimpleMotorFeedforward with the specified gains.

Parameters

- **kS** – The static gain, in volts.
- **kV** – The velocity gain, in volt seconds per distance.
- **kA** – The acceleration gain, in volt seconds^2 per distance.

calculate(*velocity: units_per_second*, *acceleration: units_per_second_squared = 0.0*) → volts

Calculates the feedforward from the gains and setpoints.

Parameters

- **velocity** – The velocity setpoint, in distance per second.
- **acceleration** – The acceleration setpoint, in distance per second^2.

Returns The computed feedforward, in volts.

kA

kS

kV

maxAchievableAcceleration(*maxVoltage: volts*, *velocity: units_per_second*) → units_per_second_squared

Calculates the maximum achievable acceleration given a maximum voltage supply and a velocity. Useful for ensuring that velocity and acceleration constraints for a trapezoidal profile are simultaneously achievable - enter the velocity constraint, and this will give you a simultaneously-achievable acceleration constraint.

Parameters

- **maxVoltage** – The maximum voltage that can be supplied to the motor.
- **velocity** – The velocity of the motor.

Returns The maximum possible acceleration at the given velocity.

maxAchievableVelocity(*maxVoltage: volts*, *acceleration: units_per_second_squared*) → units_per_second

Calculates the maximum achievable velocity given a maximum voltage supply and an acceleration. Useful for ensuring that velocity and acceleration constraints for a trapezoidal profile are simultaneously achievable - enter the acceleration constraint, and this will give you a simultaneously-achievable velocity constraint.

Parameters

- **maxVoltage** – The maximum voltage that can be supplied to the motor.
- **acceleration** – The acceleration of the motor.

Returns The maximum possible velocity at the given acceleration.

minAchievableAcceleration(*maxVoltage: volts*, *velocity: units_per_second*) → units_per_second_squared

Calculates the minimum achievable acceleration given a maximum voltage supply and a velocity. Useful for ensuring that velocity and acceleration constraints for a trapezoidal profile are simultaneously achievable - enter the velocity constraint, and this will give you a simultaneously-achievable acceleration constraint.

Parameters

- **maxVoltage** – The maximum voltage that can be supplied to the motor.
- **velocity** – The velocity of the motor.

Returns The minimum possible acceleration at the given velocity.

minAchievableVelocity(*maxVoltage: volts*, *acceleration: units_per_second_squared*) → units_per_second

Calculates the minimum achievable velocity given a maximum voltage supply and an acceleration. Useful for ensuring that velocity and acceleration constraints for a trapezoidal profile are simultaneously achievable - enter the acceleration constraint, and this will give you a simultaneously-achievable velocity constraint.

Parameters

- **maxVoltage** – The maximum voltage that can be supplied to the motor.
- **acceleration** – The acceleration of the motor.

Returns The minimum possible velocity at the given acceleration.

1.2.7 SimpleMotorFeedforwardMeters

class wpilib.controller.**SimpleMotorFeedforwardMeters**(**args*, ***kwargs*)

Bases: pybind11_builtins.pybind11_object

A helper class that computes feedforward voltages for a simple permanent-magnet DC motor.

Overloaded function.

1. __init__(self: wpilib.controller._controller.SimpleMotorFeedforwardMeters) -> None
2. __init__(self: wpilib.controller._controller.SimpleMotorFeedforwardMeters, kS: volts, kV: volt_seconds_per_meter, kA: volt_seconds_squared_per_meter = 0.0) -> None

Creates a new SimpleMotorFeedforward with the specified gains.

Parameters

- **kS** – The static gain, in volts.
- **kV** – The velocity gain, in volt seconds per distance.
- **kA** – The acceleration gain, in volt seconds^2 per distance.

calculate(*velocity: meters_per_second*, *acceleration: meters_per_second_squared = 0.0*) → volts

Calculates the feedforward from the gains and setpoints.

Parameters

- **velocity** – The velocity setpoint, in distance per second.
- **acceleration** – The acceleration setpoint, in distance per second^2.

Returns The computed feedforward, in volts.

kA

kS

kV

maxAchievableAcceleration(*maxVoltage: volts*, *velocity: meters_per_second*) → meters_per_second_squared

Calculates the maximum achievable acceleration given a maximum voltage supply and a velocity. Useful for ensuring that velocity and acceleration constraints for a trapezoidal profile are simultaneously achievable - enter the velocity constraint, and this will give you a simultaneously-achievable acceleration constraint.

Parameters

- **maxVoltage** – The maximum voltage that can be supplied to the motor.
- **velocity** – The velocity of the motor.

Returns The maximum possible acceleration at the given velocity.

maxAchievableVelocity(*maxVoltage: volts*, *acceleration: meters_per_second_squared*) → meters_per_second

Calculates the maximum achievable velocity given a maximum voltage supply and an acceleration. Useful for ensuring that velocity and acceleration constraints for a trapezoidal profile are simultaneously achievable - enter the acceleration constraint, and this will give you a simultaneously-achievable velocity constraint.

Parameters

- **maxVoltage** – The maximum voltage that can be supplied to the motor.
- **acceleration** – The acceleration of the motor.

Returns The maximum possible velocity at the given acceleration.

minAchievableAcceleration(*maxVoltage: volts*, *velocity: meters_per_second*) → meters_per_second_squared

Calculates the minimum achievable acceleration given a maximum voltage supply and a velocity. Useful for ensuring that velocity and acceleration constraints for a trapezoidal profile are simultaneously achievable - enter the velocity constraint, and this will give you a simultaneously-achievable acceleration constraint.

Parameters

- **maxVoltage** – The maximum voltage that can be supplied to the motor.
- **velocity** – The velocity of the motor.

Returns The minimum possible acceleration at the given velocity.

minAchievableVelocity(*maxVoltage: volts*, *acceleration: meters_per_second_squared*) → meters_per_second

Calculates the minimum achievable velocity given a maximum voltage supply and an acceleration. Useful for ensuring that velocity and acceleration constraints for a trapezoidal profile are simultaneously achievable - enter the acceleration constraint, and this will give you a simultaneously-achievable velocity constraint.

Parameters

- **maxVoltage** – The maximum voltage that can be supplied to the motor.
- **acceleration** – The acceleration of the motor.

Returns The minimum possible velocity at the given acceleration.

1.3 wpilib.drive Package

wpilib.drive.DifferentialDrive(self, ...)	A class for driving differential drive/skid-steer drive platforms such as the Kit of Parts drive base, "tank drive", or West Coast Drive.
wpilib.drive.KilloughDrive(*args, **kwargs)	A class for driving Killough drive platforms.
wpilib.drive.MecanumDrive(self, ...)	A class for driving Mecanum drive platforms.
wpilib.drive.RobotDriveBase(self)	Common base class for drive platforms.
wpilib.drive.Vector2d(*args, **kwargs)	This is a 2D vector struct that supports basic vector operations.

1.3.1 DifferentialDrive

class wpilib.drive.**DifferentialDrive**(*leftMotor: wpilib.interfaces._interfaces.SpeedController*, *rightMotor: wpilib.interfaces._interfaces.SpeedController*) → None

Bases: *wpilib.drive.RobotDriveBase*, *wpilib.Sendable*

A class for driving differential drive/skid-steer drive platforms such as the Kit of Parts drive base, "tank drive", or West Coast Drive.

These drive bases typically have drop-center / skid-steer with two or more wheels per side (e.g., 6WD or 8WD). This class takes a SpeedController per side. For four and six motor drivetrains, construct and pass in SpeedControllerGroup instances as follows.

Four motor drivetrain: @code{.cpp} class Robot { public: frc::PWMVictorSPX m_frontLeft{1}; frc::PWMVictorSPX m_rearLeft{2}; frc::SpeedControllerGroup m_left{m_frontLeft, m_rearLeft};

frc::PWMVictorSPX m_frontRight{3}; frc::PWMVictorSPX m_rearRight{4}; frc::SpeedControllerGroup m_right{m_frontRight, m_rearRight};

frc::DifferentialDrive m_drive{m_left, m_right}; }; @endcode

Six motor drivetrain: @code{.cpp} class Robot { public: frc::PWMVictorSPX m_frontLeft{1}; frc::PWMVictorSPX m_midLeft{2}; frc::PWMVictorSPX m_rearLeft{3}; frc::SpeedControllerGroup m_left{m_frontLeft, m_midLeft, m_rearLeft};

frc::PWMVictorSPX m_frontRight{4}; frc::PWMVictorSPX m_midRight{5}; frc::PWMVictorSPX m_rearRight{6}; frc::SpeedControllerGroup m_right{m_frontRight, m_midRight, m_rearRight};

frc::DifferentialDrive m_drive{m_left, m_right}; }; @endcode

A differential drive robot has left and right wheels separated by an arbitrary width.

Drive base diagram: <pre> |_______| | | | | | | |_|___|_| | | </pre>

Each Drive() function provides different inverse kinematic relations for a differential drive robot. Motor outputs for the right side are negated, so motor direction inversion by the user is usually unnecessary.

This library uses the NED axes convention (North-East-Down as external reference in the world frame): http://www.nuclearprojects.com/ins/images/axis_big.png.

The positive X axis points ahead, the positive Y axis points to the right, and the positive Z axis points down. Rotations follow the right-hand rule, so clockwise rotation around the Z axis is positive.

Inputs smaller then 0.02 will be set to 0, and larger values will be scaled so that the full range is still used. This deadband value can be changed with SetDeadband().

RobotDrive porting guide: TankDrive(double, double, bool) is equivalent to RobotDrive#TankDrive(double, double, bool) if a deadband of 0 is used. ArcadeDrive(double, double, bool) is equivalent to RobotDrive#ArcadeDrive(double, double, bool) if a deadband of 0 is used and the the rotation input is inverted

eg ArcadeDrive(y, -rotation, false) CurvatureDrive(double, double, bool) is similar in concept to RobotDrive#Drive(double, double) with the addition of a quick turn mode. However, it is not designed to give exactly the same response.

Construct a DifferentialDrive.

To pass multiple motors per side, use a SpeedControllerGroup. If a motor needs to be inverted, do so before passing it in.

arcadeDrive(*xSpeed: float*, *zRotation: float*, *squareInputs: bool = True*) → None
Arcade drive method for differential drive platform.

Note: Some drivers may prefer inverted rotation controls. This can be done by negating the value passed for rotation.

Parameters

- **xSpeed** – The speed at which the robot should drive along the X axis [-1.0..1.0]. Forward is positive.
- **zRotation** – The rotation rate of the robot around the Z axis [-1.0..1.0]. Clockwise is positive.
- **squareInputs** – If set, decreases the input sensitivity at low speeds.

curvatureDrive(*xSpeed: float*, *zRotation: float*, *isQuickTurn: bool*) → None
Curvature drive method for differential drive platform.

The rotation argument controls the curvature of the robot's path rather than its rate of heading change. This makes the robot more controllable at high speeds. Also handles the robot's quick turn functionality - "quick turn" overrides constant-curvature turning for turn-in-place maneuvers.

Parameters

- **xSpeed** – The robot's speed along the X axis [-1.0..1.0]. Forward is positive.
- **zRotation** – The robot's rotation rate around the Z axis [-1.0..1.0]. Clockwise is positive.
- **isQuickTurn** – If set, overrides constant-curvature turning for turn-in-place maneuvers.

getDescription(*desc: wpi::raw_ostream*) → None

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

isRightSideInverted() → bool
Gets if the power sent to the right side of the drivetrain is multipled by -1.

Returns true if the right side is inverted

kDefaultQuickStopAlpha = 0.1

kDefaultQuickStopThreshold = 0.2

setQuickStopAlpha(*alpha: float*) → None
Sets the low-pass filter gain for QuickStop in curvature drive.

The low-pass filter filters incoming rotation rate commands to smooth out high frequency changes.

Parameters **alpha** – Low-pass filter gain [0.0..2.0]. Smaller values result in slower output changes. Values between 1.0 and 2.0 result in output oscillation. Values below 0.0 and above 2.0 are unstable.

setQuickStopThreshold(*threshold: float*) → None
Sets the QuickStop speed threshold in curvature drive.

QuickStop compensates for the robot's moment of inertia when stopping after a QuickTurn.

While QuickTurn is enabled, the QuickStop accumulator takes on the rotation rate value outputted by the low-pass filter when the robot's speed along the X axis is below the threshold. When QuickTurn is disabled, the accumulator's value is applied against the computed angular power request to slow the robot's rotation.

Parameters **threshold** – X speed below which quick stop accumulator will receive rotation rate values [0..1.0].

setRightSideInverted(*rightSideInverted: bool*) → None
Sets if the power sent to the right side of the drivetrain should be multipled by -1.

Parameters **rightSideInverted** – true if right side power should be multipled by -1

stopMotor() → None

tankDrive(*leftSpeed: float*, *rightSpeed: float*, *squareInputs: bool = True*) → None
Tank drive method for differential drive platform.

Parameters

- **leftSpeed** – The robot left side's speed along the X axis [-1.0..1.0]. Forward is positive.
- **rightSpeed** – The robot right side's speed along the X axis [-1.0..1.0]. Forward is positive.
- **squareInputs** – If set, decreases the input sensitivity at low speeds.

1.3.2 KilloughDrive

class wpilib.drive.**KilloughDrive**(**args*, ***kwargs*)
Bases: *wpilib.drive.RobotDriveBase*, *wpilib.Sendable*

A class for driving Killough drive platforms.

Killough drives are triangular with one omni wheel on each corner.

Drive base diagram: <pre> /_____ * / / * / — </pre>

Each Drive() function provides different inverse kinematic relations for a Killough drive. The default wheel vectors are parallel to their respective opposite sides, but can be overridden. See the constructor for more information.

This library uses the NED axes convention (North-East-Down as external reference in the world frame): http://www.nuclearprojects.com/ins/images/axis_big.png.

The positive X axis points ahead, the positive Y axis points right, and the and the positive Z axis points down. Rotations follow the right-hand rule, so clockwise rotation around the Z axis is positive.

Overloaded function.

1. __init__(self: wpilib.drive._drive.KilloughDrive, leftMotor: wpilib.interfaces._interfaces.SpeedController, rightMotor: wpilib.interfaces._interfaces.SpeedController, backMotor: wpilib.interfaces._interfaces.SpeedController) -> None

Construct a Killough drive with the given motors and default motor angles.

The default motor angles make the wheels on each corner parallel to their respective opposite sides.

If a motor needs to be inverted, do so before passing it in.

Parameters

- **leftMotor** – The motor on the left corner.

- **rightMotor** – The motor on the right corner.
- **backMotor** – The motor on the back corner.

2. __init__(self: wpilib.drive._drive.KilloughDrive, leftMotor: wpilib.interfaces._interfaces.SpeedController, rightMotor: wpilib.interfaces._interfaces.SpeedController, backMotor: wpilib.interfaces._interfaces.SpeedController, leftMotorAngle: float, rightMotorAngle: float, backMotorAngle: float) -> None

Construct a Killough drive with the given motors.

Angles are measured in degrees clockwise from the positive X axis.

Parameters

- **leftMotor** – The motor on the left corner.
- **rightMotor** – The motor on the right corner.
- **backMotor** – The motor on the back corner.
- **leftMotorAngle** – The angle of the left wheel's forward direction of travel.
- **rightMotorAngle** – The angle of the right wheel's forward direction of travel.
- **backMotorAngle** – The angle of the back wheel's forward direction of travel.

driveCartesian(*ySpeed: float*, *xSpeed: float*, *zRotation: float*, *gyroAngle: float = 0.0*) → None
Drive method for Killough platform.

Angles are measured clockwise from the positive X axis. The robot's speed is independent from its angle or rotation rate.

Parameters

- **ySpeed** – The robot's speed along the Y axis [-1.0..1.0]. Right is positive.
- **xSpeed** – The robot's speed along the X axis [-1.0..1.0]. Forward is positive.
- **zRotation** – The robot's rotation rate around the Z axis [-1.0..1.0]. Clockwise is positive.
- **gyroAngle** – The current angle reading from the gyro in degrees around the Z axis. Use this to implement field-oriented controls.

drivePolar(*magnitude: float*, *angle: float*, *zRotation: float*) → None
Drive method for Killough platform.

Angles are measured clockwise from the positive X axis. The robot's speed is independent from its angle or rotation rate.

Parameters

- **magnitude** – The robot's speed at a given angle [-1.0..1.0]. Forward is positive.
- **angle** – The angle around the Z axis at which the robot drives in degrees [-180..180].
- **zRotation** – The robot's rotation rate around the Z axis [-1.0..1.0]. Clockwise is positive.

getDescription(*desc: wpi::raw_ostream*) → None

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

kDefaultBackMotorAngle = 270.0

kDefaultLeftMotorAngle = 60.0

kDefaultRightMotorAngle = 120.0

stopMotor() → None

1.3.3 MecanumDrive

class wpilib.drive.**MecanumDrive**(*frontLeftMotor: wpilib.interfaces._interfaces.SpeedController*, *rearLeftMotor: wpilib.interfaces._interfaces.SpeedController*, *frontRightMotor: wpilib.interfaces._interfaces.SpeedController*, *rearRightMotor: wpilib.interfaces._interfaces.SpeedController*) → None

Bases: *wpilib.drive.RobotDriveBase*, *wpilib.Sendable*

A class for driving Mecanum drive platforms.

Mecanum drives are rectangular with one wheel on each corner. Each wheel has rollers toed in 45 degrees toward the front or back. When looking at the wheels from the top, the roller axles should form an X across the robot.

Drive base diagram: <pre> _____/ \ | | / | | /_|___|_* / * </pre>

Each Drive() function provides different inverse kinematic relations for a Mecanum drive robot. Motor outputs for the right side are negated, so motor direction inversion by the user is usually unnecessary.

This library uses the NED axes convention (North-East-Down as external reference in the world frame): http://www.nuclearprojects.com/ins/images/axis_big.png.

The positive X axis points ahead, the positive Y axis points to the right, and the positive Z axis points down. Rotations follow the right-hand rule, so clockwise rotation around the Z axis is positive.

Inputs smaller then 0.02 will be set to 0, and larger values will be scaled so that the full range is still used. This deadband value can be changed with SetDeadband().

RobotDrive porting guide: In MecanumDrive, the right side speed controllers are automatically inverted, while in RobotDrive, no speed controllers are automatically inverted. DriveCartesian(double, double, double, double) is equivalent to RobotDrive#MecanumDrive_Cartesian(double, double, double, double) if a deadband of 0 is used, and the ySpeed and gyroAngle values are inverted compared to RobotDrive (eg DriveCartesian(xSpeed, -ySpeed, zRotation, -gyroAngle). DrivePolar(double, double, double) is equivalent to RobotDrive#MecanumDrive_Polar(double, double, double) if a deadband of 0 is used.

Construct a MecanumDrive.

If a motor needs to be inverted, do so before passing it in.

driveCartesian(*ySpeed: float*, *xSpeed: float*, *zRotation: float*, *gyroAngle: float = 0.0*) → None

Drive method for Mecanum platform.

Angles are measured clockwise from the positive X axis. The robot's speed is independent from its angle or rotation rate.

Parameters

- **ySpeed** – The robot's speed along the Y axis [-1.0..1.0]. Right is positive.
- **xSpeed** – The robot's speed along the X axis [-1.0..1.0]. Forward is positive.
- **zRotation** – The robot's rotation rate around the Z axis [-1.0..1.0]. Clockwise is positive.
- **gyroAngle** – The current angle reading from the gyro in degrees around the Z axis. Use this to implement field-oriented controls.

drivePolar(*magnitude: float*, *angle: float*, *zRotation: float*) → None
Drive method for Mecanum platform.

Angles are measured clockwise from the positive X axis. The robot's speed is independent from its angle or rotation rate.

Parameters

- **magnitude** – The robot's speed at a given angle [-1.0..1.0]. Forward is positive.
- **angle** – The angle around the Z axis at which the robot drives in degrees [-180..180].
- **zRotation** – The robot's rotation rate around the Z axis [-1.0..1.0]. Clockwise is positive.

getDescription(*desc: wpi::raw_ostream*) → None

initSendable(*builder: wpilib._wpilib.SendableBuilder*) → None

isRightSideInverted() → bool
Gets if the power sent to the right side of the drivetrain is multipled by -1.

Returns true if the right side is inverted

setRightSideInverted(*rightSideInverted: bool*) → None
Sets if the power sent to the right side of the drivetrain should be multipled by -1.

Parameters **rightSideInverted** – true if right side power should be multipled by -1

stopMotor() → None

1.3.4 RobotDriveBase

class wpilib.drive.**RobotDriveBase**() → None
Bases: *wpilib.MotorSafety*

Common base class for drive platforms.

class MotorType(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

The location of a motor on the robot for the purpose of driving.

Members:

kFrontLeft

kFrontRight

kRearLeft

kRearRight

kLeft

kRight

kBack

kBack = MotorType.kRearLeft

kFrontLeft = MotorType.kFrontLeft

kFrontRight = MotorType.kFrontRight

kLeft = MotorType.kFrontLeft

kRearLeft = MotorType.kRearLeft

kRearRight = MotorType.kRearRight

kRight = MotorType.kFrontRight

name
(self: handle) -> str

feedWatchdog() → None
Feed the motor safety object. Resets the timer that will stop the motors if it completes.

@see MotorSafetyHelper::Feed()

getDescription(*desc: wpi::raw_ostream*) → None

setDeadband(*deadband: float*) → None
Sets the deadband applied to the drive inputs (e.g., joystick values).

The default value is 0.02. Inputs smaller than the deadband are set to 0.0 while inputs larger than the deadband are scaled from 0.0 to 1.0. See ApplyDeadband().

Parameters **deadband** – The deadband to set.

setMaxOutput(*maxOutput: float*) → None
Configure the scaling factor for using RobotDrive with motor controllers in a mode other than PercentVbus or to limit the maximum output.

Parameters **maxOutput** – Multiplied with the output percentage computed by the drive functions.

stopMotor() → None

1.3.5 Vector2d

class wpilib.drive.**Vector2d**(**args*, ***kwargs*)
Bases: pybind11_builtins.pybind11_object

This is a 2D vector struct that supports basic vector operations.

Overloaded function.

1. __init__(self: wpilib.drive._drive.Vector2d) -> None
2. __init__(self: wpilib.drive._drive.Vector2d, x: float, y: float) -> None

dot(*vec: wpilib.drive._drive.Vector2d*) → float
Returns dot product of this vector with argument.

Parameters **vec** – Vector with which to perform dot product.

magnitude() → float
Returns magnitude of vector.

rotate(*angle: float*) → None
Rotate a vector in Cartesian space.

Parameters **angle** – angle in degrees by which to rotate vector counter-clockwise.

scalarProject(*vec: wpilib.drive._drive.Vector2d*) → float
Returns scalar projection of this vector onto argument.

Parameters **vec** – Vector onto which to project this vector.

x

y

1.4 wpilib.geometry Package

wpilib.geometry.Pose2d(*args, **kwargs)	Represents a 2d pose containing translational and rotational elements.
wpilib.geometry.Rotation2d(*args, **kwargs)	A rotation in a 2d coordinate frame represented a point on the unit circle (cosine and sine).
wpilib.geometry.Transform2d(*args, **kwargs)	Represents a transformation for a Pose2d.
wpilib.geometry.Translation2d(*args, **kwargs)	Represents a translation in 2d space.
wpilib.geometry.Twist2d(self, dx, dy, dtheta)	A change in distance along arc since the last pose update.

1.4.1 Pose2d

class wpilib.geometry.**Pose2d**(**args*, ***kwargs*)

Bases: pybind11_builtins.pybind11_object

Represents a 2d pose containing translational and rotational elements.

Overloaded function.

1. __init__(self: wpilib.geometry._geometry.Pose2d) -> None

Constructs a pose at the origin facing toward the positive X axis. (Translation2d{0, 0} and Rotation{0})

2. __init__(self: wpilib.geometry._geometry.Pose2d, translation: frc::Translation2d, rotation: frc::Rotation2d) -> None

Constructs a pose with the specified translation and rotation.

Parameters

- **translation** – The translational component of the pose.
- **rotation** – The rotational component of the pose.

3. __init__(self: wpilib.geometry._geometry.Pose2d, x: meters, y: meters, rotation: frc::Rotation2d) -> None

Convenience constructors that takes in x and y values directly instead of having to construct a Translation2d.

Parameters

- **x** – The x component of the translational component of the pose.
- **y** – The y component of the translational component of the pose.
- **rotation** – The rotational component of the pose.

4. __init__(self: wpilib.geometry._geometry.Pose2d, x: meters, y: meters, angle: radians) -> None

exp(*twist: frc::Twist2d*) → wpilib.geometry._geometry.Pose2d

Obtain a new Pose2d from a (constant curvature) velocity.

See <https://file.tavsys.net/control/state-space-guide.pdf> section on nonlinear pose estimation for derivation.

The twist is a change in pose in the robot's coordinate frame since the previous pose update. When the user runs exp() on the previous known field-relative pose with the argument being the twist, the user will receive the new field-relative pose.

"Exp" represents the pose exponential, which is solving a differential equation moving the pose forward in time.

Parameters **twist** – The change in pose in the robot's coordinate frame since the previous pose update. For example, if a non-holonomic robot moves forward 0.01 meters and changes angle by 0.5 degrees since the previous pose update, the twist would be Twist2d{0.01, 0.0, toRadians(0.5)}

Returns The new pose of the robot.

static fromFeet(**args*, ***kwargs*)
Overloaded function.

1. fromFeet(x: feet, y: feet, angle: radians) -> wpilib.geometry._geometry.Pose2d
2. fromFeet(x: feet, y: feet, r: frc::Rotation2d) -> wpilib.geometry._geometry.Pose2d

log(*end: wpilib.geometry._geometry.Pose2d*) → frc::Twist2d
Returns a Twist2d that maps this pose to the end pose. If c is the output of a.Log(b), then a.Exp(c) would yield b.

Parameters **end** – The end pose for the transformation.

Returns The twist that maps this to end.

relativeTo(*other: wpilib.geometry._geometry.Pose2d*) → wpilib.geometry._geometry.Pose2d
Returns the other pose relative to the current pose.

This function can often be used for trajectory tracking or pose stabilization algorithms to get the error between the reference and the current pose.

Parameters **other** – The pose that is the origin of the new coordinate frame that the current pose will be converted into.

Returns The current pose relative to the new origin pose.

rotation() → frc::Rotation2d
Returns the underlying rotation.

Returns Reference to the rotational component of the pose.

transformBy(*other: frc::Transform2d*) → wpilib.geometry._geometry.Pose2d
Transforms the pose by the given transformation and returns the new pose. See + operator for the matrix multiplication performed.

Parameters **other** – The transform to transform the pose by.

Returns The transformed pose.

translation() → frc::Translation2d
Returns the underlying translation.

Returns Reference to the translational component of the pose.

1.4.2 Rotation2d

class wpilib.geometry.**Rotation2d**(**args*, ***kwargs*)
Bases: pybind11_builtins.pybind11_object

A rotation in a 2d coordinate frame represented a point on the unit circle (cosine and sine).

Overloaded function.

1. __init__(self: wpilib.geometry._geometry.Rotation2d) -> None

Constructs a Rotation2d with a default angle of 0 degrees.

2. __init__(self: wpilib.geometry._geometry.Rotation2d, value: radians) -> None

Constructs a Rotation2d with the given radian value.

Parameters **value** – The value of the angle in radians.

3. __init__(self: wpilib.geometry._geometry.Rotation2d, x: float, y: float) -> None

Constructs a Rotation2d with the given x and y (cosine and sine) components. The x and y don't have to be normalized.

Parameters

- **x** – The x component or cosine of the rotation.
- **y** – The y component or sine of the rotation.

cos() → float
Returns the cosine of the rotation.

Returns The cosine of the rotation.

degrees() → degrees
Returns the degree value of the rotation.

Returns The degree value of the rotation.

static fromDegrees(*value: degrees*) → wpilib.geometry._geometry.Rotation2d

radians() → radians
Returns the radian value of the rotation.

Returns The radian value of the rotation.

rotateBy(*other: wpilib.geometry._geometry.Rotation2d*) → wpilib.geometry._geometry.Rotation2d
Adds the new rotation to the current rotation using a rotation matrix.

[cos_new] [other.cos, -other.sin][cos] [sin_new] = [other.sin, other.cos][sin]

value_new = std::atan2(cos_new, sin_new)

Parameters **other** – The rotation to rotate by.

Returns The new rotated Rotation2d.

sin() → float
Returns the sine of the rotation.

Returns The sine of the rotation.

tan() → float
Returns the tangent of the rotation.

Returns The tangent of the rotation.

1.4.3 Transform2d

class wpilib.geometry.**Transform2d**(**args*, ***kwargs*)
Bases: pybind11_builtins.pybind11_object

Represents a transformation for a Pose2d.

Overloaded function.

1. __init__(self: wpilib.geometry._geometry.Transform2d, initial: wpilib.geometry._geometry.Pose2d, final: wpilib.geometry._geometry.Pose2d) -> None

Constructs the transform that maps the initial pose to the final pose.

Parameters

- **initial** – The initial pose for the transformation.
- **final** – The final pose for the transformation.

2. __init__(self: wpilib.geometry._geometry.Transform2d, translation: frc::Translation2d, rotation: wpilib.geometry._geometry.Rotation2d) -> None

Constructs a transform with the given translation and rotation components.

Parameters

- **translation** – Translational component of the transform.
- **rotation** – Rotational component of the transform.

3. __init__(self: wpilib.geometry._geometry.Transform2d) -> None

Constructs the identity transform – maps an initial pose to itself.

4. __init__(self: wpilib.geometry._geometry.Transform2d, x: meters, y: meters, angle: radians) -> None

static fromFeet(*x: feet*, *y: feet*, *angle: radians*) → wpilib.geometry._geometry.Transform2d

rotation() → wpilib.geometry._geometry.Rotation2d
Returns the rotational component of the transformation.

Returns Reference to the rotational component of the transform.

translation() → frc::Translation2d
Returns the translation component of the transformation.

Returns Reference to the translational component of the transform.

1.4.4 Translation2d

class wpilib.geometry.**Translation2d**(**args*, ***kwargs*)
Bases: pybind11_builtins.pybind11_object

Represents a translation in 2d space. This object can be used to represent a point or a vector.

This assumes that you are using conventional mathematical axes. When the robot is placed on the origin, facing toward the X direction, moving forward increases the X, whereas moving to the left increases the Y.

Overloaded function.

1. __init__(self: wpilib.geometry._geometry.Translation2d) -> None

Constructs a Translation2d with X and Y components equal to zero.

2. __init__(self: wpilib.geometry._geometry.Translation2d, x: meters, y: meters) -> None

Constructs a Translation2d with the X and Y components equal to the provided values.

Parameters

- **x** – The x component of the translation.
- **y** – The y component of the translation.

X() → meters
Returns the X component of the translation.

Returns The x component of the translation.

Y() → meters
Returns the Y component of the translation.

Returns The y component of the translation.

distance(*other: wpilib.geometry._geometry.Translation2d*) → meters
Calculates the distance between two translations in 2d space.

This function uses the pythagorean theorem to calculate the distance. distance = std::sqrt((x2 - x1)^2 + (y2 - y1)^2)

Parameters **other** – The translation to compute the distance to.

Returns The distance between the two translations.

distanceFeet(*arg0: wpilib.geometry._geometry.Translation2d*) → feet

static fromFeet(*x: feet*, *y: feet*) → wpilib.geometry._geometry.Translation2d

norm() → meters
Returns the norm, or distance from the origin to the translation.

Returns The norm of the translation.

normFeet() → feet

rotateBy(*other: wpilib.geometry._geometry.Rotation2d*) → wpilib.geometry._geometry.Translation2d
Applies a rotation to the translation in 2d space.

This multiplies the translation vector by a counterclockwise rotation matrix of the given angle.

[x_new] [other.cos, -other.sin][x] [y_new] = [other.sin, other.cos][y]

For example, rotating a Translation2d of {2, 0} by 90 degrees will return a Translation2d of {0, 2}.

Parameters **other** – The rotation to rotate the translation by.

Returns The new rotated translation.

x

x_feet

y

y_feet

1.4.5 Twist2d

class wpilib.geometry.**Twist2d**(*dx: meters = 0*, *dy: meters = 0*, *dtheta: radians = 0*) → None
Bases: pybind11_builtins.pybind11_object

A change in distance along arc since the last pose update. We can use ideas from differential calculus to create new Pose2ds from a Twist2d and vise versa.

A Twist can be used to represent a difference between two poses.

dtheta
Angular "dtheta" component (radians)

dtheta_degrees

dx
Linear "dx" component

dx_feet

dy
Linear "dy" component

dy_feet

static fromFeet(*dx: feet = 0*, *dy: feet = 0*, *dtheta: radians = 0*) → wpilib.geometry._geometry.Twist2d

1.5 wpilib.interfaces Package

wpilib.interfaces.Accelerometer(self)	Interface for 3-axis accelerometers.
wpilib.interfaces.CounterBase(self)	Interface for counting the number of ticks on a digital input channel.
wpilib.interfaces.GenericHID(self, port)	GenericHID Interface.
wpilib.interfaces.Gyro(self)	Interface for yaw rate gyros.
wpilib.interfaces.PIDOutput(self)	PIDOutput interface is a generic output for the PID class.
wpilib.interfaces.PIDSource(self)	PIDSource interface is a generic sensor source for the PID class.
wpilib.interfaces.PIDSourceType(self, arg0)	Members:
wpilib.interfaces.Potentiometer(self)	Interface for potentiometers.
wpilib.interfaces.SpeedController(self)	Interface for speed controlling devices.

1.5.1 Accelerometer

class wpilib.interfaces.**Accelerometer**() → None
Bases: pybind11_builtins.pybind11_object

Interface for 3-axis accelerometers.

class Range(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kRange_2G

kRange_4G

kRange_8G

kRange_16G

kRange_16G = Range.kRange_16G

kRange_2G = Range.kRange_2G

kRange_4G = Range.kRange_4G

kRange_8G = Range.kRange_8G

name
(self: handle) -> str

getX() → float
Common interface for getting the x axis acceleration.

Returns The acceleration along the x axis in g-forces

getY() → float
Common interface for getting the y axis acceleration.

Returns The acceleration along the y axis in g-forces

getZ() → float
Common interface for getting the z axis acceleration.

Returns The acceleration along the z axis in g-forces

setRange(*range: wpilib.interfaces._interfaces.Accelerometer.Range*) → None
Common interface for setting the measuring range of an accelerometer.

Parameters **range** – The maximum acceleration, positive or negative, that the accelerometer will measure. Not all accelerometers support all ranges.

1.5.2 CounterBase

class wpilib.interfaces.**CounterBase**() → None
Bases: pybind11_builtins.pybind11_object

Interface for counting the number of ticks on a digital input channel.

Encoders, Gear tooth sensors, and counters should all subclass this so it can be used to build more advanced classes for control and driving.

All counters will immediately start counting - Reset() them if you need them to be zeroed before use.

class EncodingType(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

k1X

k2X

k4X

k1X = EncodingType.k1X

k2X = EncodingType.k2X

k4X = EncodingType.k4X

name
(self: handle) -> str

get() → int

getDirection() → bool

getPeriod() → float

getStopped() → bool

reset() → None

setMaxPeriod(*maxPeriod: float*) → None

1.5.3 GenericHID

class wpilib.interfaces.**GenericHID**(*port: int*) → None
Bases: pybind11_builtins.pybind11_object

GenericHID Interface.

class HIDType(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kUnknown

kXInputUnknown

kXInputGamepad

kXInputWheel

kXInputArcadeStick

kXInputFlightStick

kXInputDancePad

kXInputGuitar

kXInputGuitar2

kXInputDrumKit

kXInputGuitar3

kXInputArcadePad

kHIDJoystick

kHIDGamepad

kHIDDriving

kHIDFlight

kHID1stPerson

kHID1stPerson = HIDType.kHID1stPerson

kHIDDriving = HIDType.kHIDDriving

kHIDFlight = HIDType.kHIDFlight

kHIDGamepad = HIDType.kHIDGamepad

kHIDJoystick = HIDType.kHIDJoystick

kUnknown = HIDType.kUnknown

kXInputArcadePad = HIDType.kXInputArcadePad

kXInputArcadeStick = HIDType.kXInputArcadeStick

kXInputDancePad = HIDType.kXInputDancePad

kXInputDrumKit = HIDType.kXInputDrumKit

kXInputFlightStick = HIDType.kXInputFlightStick

kXInputGamepad = HIDType.kXInputGamepad

kXInputGuitar = HIDType.kXInputGuitar

kXInputGuitar2 = HIDType.kXInputGuitar2

kXInputGuitar3 = HIDType.kXInputGuitar3

kXInputUnknown = HIDType.kXInputUnknown

kXInputWheel = HIDType.kXInputWheel

name
(self: handle) -> str

class Hand(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kLeftHand

kRightHand

kLeftHand = Hand.kLeftHand

kRightHand = Hand.kRightHand

name
(self: handle) -> str

class RumbleType(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kLeftRumble

kRightRumble

kLeftRumble = RumbleType.kLeftRumble

kRightRumble = RumbleType.kRightRumble

name
(self: handle) -> str

getAxisCount() → int
Get the number of axes for the HID.

Returns the number of axis for the current HID

getAxisType(*axis: int*) → int
Get the axis type of a joystick axis.

Returns the axis type of a joystick axis.

getButtonCount() → int
Get the number of buttons for the HID.

Returns the number of buttons on the current HID

getName() → str
Get the name of the HID.

Returns the name of the HID.

getPOV(*pov: int = 0*) → int
Get the angle in degrees of a POV on the HID.

The POV angles start at 0 in the up direction, and increase clockwise (e.g. right is 90, upper-left is 315).

Parameters **pov** – The index of the POV to read (starting at 0)

Returns the angle of the POV in degrees, or -1 if the POV is not pressed.

getPOVCount() → int
Get the number of POVs for the HID.

Returns the number of POVs for the current HID

getPort() → int
Get the port number of the HID.

Returns The port number of the HID.

getRawAxis(*axis: int*) → float
Get the value of the axis.

Parameters **axis** – The axis to read, starting at 0.

Returns The value of the axis.

getRawButton(*button: int*) → bool
Get the button value (starting at button 1).

The buttons are returned in a single 16 bit value with one bit representing the state of each button. The appropriate button is returned as a boolean value.

Parameters **button** – The button number to be read (starting at 1)

Returns The state of the button.

getRawButtonPressed(*button: int*) → bool
Whether the button was pressed since the last check. Button indexes begin at 1.

Parameters **button** – The button index, beginning at 1.

Returns Whether the button was pressed since the last check.

getRawButtonReleased(*button: int*) → bool
Whether the button was released since the last check. Button indexes begin at 1.

Parameters **button** – The button index, beginning at 1.

Returns Whether the button was released since the last check.

getType() → wpilib.interfaces._interfaces.GenericHID.HIDType
Get the type of the HID.

Returns the type of the HID.

getX(*hand: wpilib.interfaces._interfaces.GenericHID.Hand = Hand.kRightHand*) → float

getY(*hand: wpilib.interfaces._interfaces.GenericHID.Hand = Hand.kRightHand*) → float

setOutput(*outputNumber: int*, *value: bool*) → None
Set a single HID output value for the HID.

Parameters

- **outputNumber** – The index of the output to set (1-32)
- **value** – The value to set the output to

setOutputs(*value: int*) → None
Set all output values for the HID.

Parameters **value** – The 32 bit output value (1 bit for each output)

setRumble(*type: wpilib.interfaces._interfaces.GenericHID.RumbleType*, *value: float*) → None
Set the rumble output for the HID.

The DS currently supports 2 rumble values, left rumble and right rumble.

Parameters

- **type** – Which rumble value to set
- **value** – The normalized value (0 to 1) to set the rumble to

1.5.4 Gyro

class wpilib.interfaces.**Gyro**() → None
Bases: pybind11_builtins.pybind11_object

Interface for yaw rate gyros.

calibrate() → None
Calibrate the gyro by running for a number of samples and computing the center value. Then use the center value as the Accumulator center value for subsequent measurements. It's important to make sure that the robot is not moving while the centering calculations are in progress, this is typically done when the robot is first turned on while it's sitting at rest before the competition starts.

getAngle() → float
Return the actual angle in degrees that the robot is currently facing.

The angle is based on the current accumulator value corrected by the oversampling rate, the gyro type and the A/D calibration values. The angle is continuous, that is it will continue from 360 to 361 degrees. This allows algorithms that wouldn't want to see a discontinuity in the gyro output as it sweeps past from 360 to 0 on the second time around.

The angle is expected to increase as the gyro turns clockwise when looked at from the top. It needs to follow NED axis conventions in order to work properly with dependent control loops.

Returns the current heading of the robot in degrees. This heading is based on integration of the returned rate from the gyro.

getRate() → float
Return the rate of rotation of the gyro.

The rate is based on the most recent reading of the gyro analog value.

The rate is expected to be positive as the gyro turns clockwise when looked at from the top. It needs to follow NED axis conventions in order to work properly with dependent control loops.

Returns the current rate in degrees per second

reset() → None
Reset the gyro. Resets the gyro to a heading of zero. This can be used if there is significant drift in the gyro and it needs to be recalibrated after it has been running.

1.5.5 PIDOutput

class wpilib.interfaces.**PIDOutput**() → None
Bases: pybind11_builtins.pybind11_object

PIDOutput interface is a generic output for the PID class.

PWMs use this class. Users implement this interface to allow for a PIDController to read directly from the inputs.

pidWrite(*output: float*) → None

1.5.6 PIDSource

class wpilib.interfaces.**PIDSource**() → None
Bases: pybind11_builtins.pybind11_object

PIDSource interface is a generic sensor source for the PID class.

All sensors that can be used with the PID class will implement the PIDSource that returns a standard value that will be used in the PID code.

getPIDSourceType() → wpilib.interfaces._interfaces.PIDSourceType

pidGet() → float

setPIDSourceType(*pidSource: wpilib.interfaces._interfaces.PIDSourceType*) → None
Set which parameter you are using as a process control variable.

Parameters **pidSource** – An enum to select the parameter.

1.5.7 PIDSourceType

class wpilib.interfaces.**PIDSourceType**(*arg0: int*) → None
Bases: pybind11_builtins.pybind11_object

Members:

kDisplacement

kRate

kDisplacement = PIDSourceType.kDisplacement

kRate = PIDSourceType.kRate

name
(self: handle) -> str

1.5.8 Potentiometer

class wpilib.interfaces.**Potentiometer**() → None
Bases: *wpilib.interfaces.PIDSource*

Interface for potentiometers.

get() → float
Common interface for getting the current value of a potentiometer.

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

setPIDSourceType(*pidSource: wpilib.interfaces._interfaces.PIDSourceType*) → None

1.5.9 SpeedController

class wpilib.interfaces.**SpeedController**() → None
Bases: *wpilib.interfaces.PIDOutput*

Interface for speed controlling devices.

disable() → None
Common interface for disabling a motor.

get() → 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.

getInverted() → bool
Common interface for returning the inversion state of a speed controller.

Returns isInverted The state of inversion, true is inverted.

set(*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.

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

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

setVoltage(*output: volts*) → None
Sets the voltage output of the SpeedController. Compensates for the current bus voltage to ensure that the desired voltage is output even if the battery voltage is below 12V - highly useful when the voltage outputs are "meaningful" (e.g. they come from a feedforward calculation).

NOTE: This function *must* be called regularly in order for voltage compensation to work properly - unlike the ordinary set function, it is not "set it and forget it."

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

stopMotor() → None
Common interface to stop the motor until Set is called again.

1.6 wpilib.kinematics Package

wpilib.kinematics.ChassisSpeeds(self, vx, ...)	Represents the speed of a robot chassis.
wpilib.kinematics.DifferentialDriveKinematics(...)	Helper class that converts a chassis velocity (dx and dtheta components) to left and right wheel velocities for a differential drive.

Continued on next page

Table 6 – continued from previous page

wpilib.kinematics.DifferentialDriveOdometry(...)	Class for differential drive odometry.
wpilib.kinematics.DifferentialDriveWheelSpeeds(...)	Represents the wheel speeds for a differential drive drivetrain.
wpilib.kinematics.MecanumDriveKinematics(...)	Helper class that converts a chassis velocity (dx, dy, and dtheta components) into individual wheel speeds.
wpilib.kinematics.MecanumDriveOdometry(self, ...)	Class for mecanum drive odometry.
wpilib.kinematics.MecanumDriveWheelSpeeds(...)	Represents the wheel speeds for a mecanum drive drivetrain.
wpilib.kinematics.SwerveDrive3Kinematics(...)	Helper class that converts a chassis velocity (dx, dy, and dtheta components) into individual module states (speed and angle).
wpilib.kinematics.SwerveDrive3Odometry(self, ...)	Class for swerve drive odometry.
wpilib.kinematics.SwerveDrive4Kinematics(...)	Helper class that converts a chassis velocity (dx, dy, and dtheta components) into individual module states (speed and angle).
wpilib.kinematics.SwerveDrive4Odometry(self, ...)	Class for swerve drive odometry.
wpilib.kinematics.SwerveDrive6Kinematics(...)	Helper class that converts a chassis velocity (dx, dy, and dtheta components) into individual module states (speed and angle).
wpilib.kinematics.SwerveDrive6Odometry(self, ...)	Class for swerve drive odometry.
wpilib.kinematics.SwerveModuleState(self, ...)	Represents the state of one swerve module.

1.6.1 ChassisSpeeds

class wpilib.kinematics.**ChassisSpeeds**(*vx: meters_per_second = 0*, *vy: meters_per_second = 0*, *omega: radians_per_second = 0*) → None

Bases: pybind11_builtins.pybind11_object

Represents the speed of a robot chassis. Although this struct contains similar members compared to a Twist2d, they do NOT represent the same thing. Whereas a Twist2d represents a change in pose w.r.t to the robot frame of reference, this ChassisSpeeds struct represents a velocity w.r.t to the robot frame of reference.

A strictly non-holonomic drivetrain, such as a differential drive, should never have a dy component because it can never move sideways. Holonomic drivetrains such as swerve and mecanum will often have all three components.

static fromFeet(*vx: feet_per_second = 0*, *vy: feet_per_second = 0*, *omega: radians_per_second = 0*) → wpilib.kinematics._kinematics.ChassisSpeeds

static fromFieldRelativeSpeeds(*vx: meters_per_second*, *vy: meters_per_second*, *omega: radians_per_second*, *robotAngle: wpilib.geometry._geometry.Rotation2d*) → wpilib.kinematics._kinematics.ChassisSpeeds

Converts a user provided field-relative set of speeds into a robot-relative ChassisSpeeds object.

Parameters

- **vx** – The component of speed in the x direction relative to the field. Positive x is away from your alliance wall.

- **vy** – The component of speed in the y direction relative to the field. Positive y is to your left when standing behind the alliance wall.
- **omega** – The angular rate of the robot.
- **robotAngle** – The angle of the robot as measured by a gyroscope. The robot's angle is considered to be zero when it is facing directly away from your alliance station wall. Remember that this should be CCW positive.

Returns ChassisSpeeds object representing the speeds in the robot's frame of reference.

omega
Represents the angular velocity of the robot frame. (CCW is +)

omega_dps

vx
Represents forward velocity w.r.t the robot frame of reference. (Fwd is +)

vx_fps

vy
Represents strafe velocity w.r.t the robot frame of reference. (Left is +)

vy_fps

1.6.2 DifferentialDriveKinematics

class wpilib.kinematics.**DifferentialDriveKinematics**(*trackWidth: meters*) → None
Bases: pybind11_builtins.pybind11_object

Helper class that converts a chassis velocity (dx and dtheta components) to left and right wheel velocities for a differential drive.

Inverse kinematics converts a desired chassis speed into left and right velocity components whereas forward kinematics converts left and right component velocities into a linear and angular chassis speed.

Constructs a differential drive kinematics object.

Parameters **trackWidth** – The track width of the drivetrain. Theoretically, this is the distance between the left wheels and right wheels. However, the empirical value may be larger than the physical measured value due to scrubbing effects.

toChassisSpeeds(*wheelSpeeds: frc::DifferentialDriveWheelSpeeds*) → wpilib.kinematics._kinematics.ChassisSpeeds
Returns a chassis speed from left and right component velocities using forward kinematics.

Parameters **wheelSpeeds** – The left and right velocities.

Returns The chassis speed.

toWheelSpeeds(*chassisSpeeds: wpilib.kinematics._kinematics.ChassisSpeeds*) → frc::DifferentialDriveWheelSpeeds
Returns left and right component velocities from a chassis speed using inverse kinematics.

Parameters **chassisSpeeds** – The linear and angular (dx and dtheta) components that represent the chassis' speed.

Returns The left and right velocities.

trackWidth

1.6.3 DifferentialDriveOdometry

class wpilib.kinematics.**DifferentialDriveOdometry**(*gyroAngle: wpilib.geometry._geometry.Rotation2d*, *initialPose: wpilib.geometry._geometry.Pose2d = Pose2d(Translation2d(x=0.000000, y=0.000000), Rotation2d(0.000000))*) → None

Bases: pybind11_builtins.pybind11_object

Class for differential drive odometry. Odometry allows you to track the robot's position on the field over the course of a match using readings from 2 encoders and a gyroscope.

Teams can use odometry during the autonomous period for complex tasks like path following. Furthermore, odometry can be used for latency compensation when using computer-vision systems.

It is important that you reset your encoders to zero before using this class. Any subsequent pose resets also require the encoders to be reset to zero.

Constructs a DifferentialDriveOdometry object.

Parameters

- **gyroAngle** – The angle reported by the gyroscope.
- **initialPose** – The starting position of the robot on the field.

getPose() → wpilib.geometry._geometry.Pose2d

Returns the position of the robot on the field.

Returns The pose of the robot.

resetPosition(*pose: wpilib.geometry._geometry.Pose2d*, *gyroAngle: wpilib.geometry._geometry.Rotation2d*) → None

Resets the robot's position on the field.

You NEED to reset your encoders (to zero) when calling this method.

The gyroscope angle does not need to be reset here on the user's robot code. The library automatically takes care of offsetting the gyro angle.

Parameters

- **pose** – The position on the field that your robot is at.
- **gyroAngle** – The angle reported by the gyroscope.

update(*gyroAngle: wpilib.geometry._geometry.Rotation2d*, *leftDistance: meters*, *rightDistance: meters*) → wpilib.geometry._geometry.Pose2d

Updates the robot position on the field using distance measurements from encoders. This method is more numerically accurate than using velocities to integrate the pose and is also advantageous for teams that are using lower CPR encoders.

Parameters

- **gyroAngle** – The angle reported by the gyroscope.
- **leftDistance** – The distance traveled by the left encoder.
- **rightDistance** – The distance traveled by the right encoder.

Returns The new pose of the robot.

1.6.4 DifferentialDriveWheelSpeeds

class wpilib.kinematics.**DifferentialDriveWheelSpeeds**(*left: meters_per_second = 0*, *right: meters_per_second = 0*) → None

Bases: pybind11_builtins.pybind11_object

Represents the wheel speeds for a differential drive drivetrain.

static fromFeet(*left: feet_per_second*, *right: feet_per_second*) → wpilib.kinematics._kinematics.DifferentialDriveWheelSpeeds

left
Speed of the left side of the robot.

left_fps

normalize(*attainableMaxSpeed: meters_per_second*) → None
Normalizes the wheel speeds using some max attainable speed. Sometimes, after inverse kinematics, the requested speed from a/several modules may be above the max attainable speed for the driving motor on that module. To fix this issue, one can "normalize" all the wheel speeds to make sure that all requested module speeds are below the absolute threshold, while maintaining the ratio of speeds between modules.

Parameters **attainableMaxSpeed** – The absolute max speed that a wheel can reach.

right
Speed of the right side of the robot.

right_fps

1.6.5 MecanumDriveKinematics

class wpilib.kinematics.**MecanumDriveKinematics**(*frontLeftWheel: wpilib.geometry._geometry.Translation2d*, *frontRightWheel: wpilib.geometry._geometry.Translation2d*, *rearLeftWheel: wpilib.geometry._geometry.Translation2d*, *rearRightWheel: wpilib.geometry._geometry.Translation2d*) → None

Bases: pybind11_builtins.pybind11_object

Helper class that converts a chassis velocity (dx, dy, and dtheta components) into individual wheel speeds.

The inverse kinematics (converting from a desired chassis velocity to individual wheel speeds) uses the relative locations of the wheels with respect to the center of rotation. The center of rotation for inverse kinematics is also variable. This means that you can set your set your center of rotation in a corner of the robot to perform special evasion manuevers.

Forward kinematics (converting an array of wheel speeds into the overall chassis motion) is performs the exact opposite of what inverse kinematics does. Since this is an overdetermined system (more equations than variables), we use a least-squares approximation.

The inverse kinematics: [wheelSpeeds] = [wheelLocations] * [chassisSpeeds] We take the Moore-Penrose pseudoinverse of [wheelLocations] and then multiply by [wheelSpeeds] to get our chassis speeds.

Forward kinematics is also used for odometry – determining the position of the robot on the field using encoders and a gyro.

Constructs a mecanum drive kinematics object.

Parameters

- **frontLeftWheel** – The location of the front-left wheel relative to the physical center of the robot.
- **frontRightWheel** – The location of the front-right wheel relative to the physical center of the robot.
- **rearLeftWheel** – The location of the rear-left wheel relative to the physical center of the robot.
- **rearRightWheel** – The location of the rear-right wheel relative to the physical center of the robot.

toChassisSpeeds(*wheelSpeeds: frc::MecanumDriveWheelSpeeds*) → wpilib.kinematics._kinematics.ChassisSpeeds

Performs forward kinematics to return the resulting chassis state from the given wheel speeds. This method is often used for odometry – determining the robot's position on the field using data from the real-world speed of each wheel on the robot.

Parameters **wheelSpeeds** – The current mecanum drive wheel speeds.

Returns The resulting chassis speed.

toWheelSpeeds(*chassisSpeeds: wpilib.kinematics._kinematics.ChassisSpeeds*, *centerOfRotation: wpilib.geometry._geometry.Translation2d = Translation2d(x=0.000000, y=0.000000)*) → frc::MecanumDriveWheelSpeeds

Performs inverse kinematics to return the wheel speeds from a desired chassis velocity. This method is often used to convert joystick values into wheel speeds.

This function also supports variable centers of rotation. During normal operations, the center of rotation is usually the same as the physical center of the robot; therefore, the argument is defaulted to that use case. However, if you wish to change the center of rotation for evasive manuevers, vision alignment, or for any other use case, you can do so.

Parameters

- **chassisSpeeds** – The desired chassis speed.
- **centerOfRotation** – The center of rotation. For example, if you set the center of rotation at one corner of the robot and provide a chassis speed that only has a dtheta component, the robot will rotate around that corner.

Returns

The wheel speeds. Use caution because they are not normalized. Sometimes, a user input may cause one of the wheel speeds to go above the attainable max velocity. Use the MecanumDriveWheelSpeeds::Normalize() function to rectify this issue. In addition, you can leverage the power of C++17 to directly assign the wheel speeds to variables:

@code{.cpp} auto [fl, fr, bl, br] = kinematics.ToWheelSpeeds(chassisSpeeds); @endcode

1.6.6 MecanumDriveOdometry

class wpilib.kinematics.**MecanumDriveOdometry**(*kinematics: wpilib.kinematics._kinematics.MecanumDriveKinematics*, *gyroAngle: wpilib.geometry._geometry.Rotation2d*, *initialPose: wpilib.geometry._geometry.Pose2d = Pose2d(Translation2d(x=0.000000, y=0.000000), Rotation2d(0.000000))*) → None

Bases: pybind11_builtins.pybind11_object

Class for mecanum drive odometry. Odometry allows you to track the robot's position on the field over a course of a match using readings from your mecanum wheel encoders.

Teams can use odometry during the autonomous period for complex tasks like path following. Furthermore, odometry can be used for latency compensation when using computer-vision systems.

Constructs a MecanumDriveOdometry object.

Parameters

- **kinematics** – The mecanum drive kinematics for your drivetrain.
- **gyroAngle** – The angle reported by the gyroscope.
- **initialPose** – The starting position of the robot on the field.

getPose() → wpilib.geometry._geometry.Pose2d

Returns the position of the robot on the field.

Returns The pose of the robot.

resetPosition(*pose: wpilib.geometry._geometry.Pose2d*, *gyroAngle: wpilib.geometry._geometry.Rotation2d*) → None

Resets the robot's position on the field.

The gyroscope angle does not need to be reset here on the user's robot code. The library automatically takes care of offsetting the gyro angle.

Parameters

- **pose** – The position on the field that your robot is at.
- **gyroAngle** – The angle reported by the gyroscope.

update(*gyroAngle: wpilib.geometry._geometry.Rotation2d*, *wheelSpeeds: frc::MecanumDriveWheelSpeeds*) → wpilib.geometry._geometry.Pose2d

Updates the robot's position on the field using forward kinematics and integration of the pose over time. This method automatically calculates the current time to calculate period (difference between two timestamps). The period is used to calculate the change in distance from a velocity. This also takes in an angle parameter which is used instead of the angular rate that is calculated from forward kinematics.

Parameters

- **gyroAngle** – The angle reported by the gyroscope.
- **wheelSpeeds** – The current wheel speeds.

Returns The new pose of the robot.

updateWithTime(*currentTime: seconds*, *gyroAngle: wpilib.geometry._geometry.Rotation2d*, *wheelSpeeds: frc::MecanumDriveWheelSpeeds*) → wpilib.geometry._geometry.Pose2d

Updates the robot's position on the field using forward kinematics and integration of the pose over time.

This method takes in the current time as a parameter to calculate period (difference between two timestamps). The period is used to calculate the change in distance from a velocity. This also takes in an angle parameter which is used instead of the angular rate that is calculated from forward kinematics.

Parameters

- **currentTime** – The current time.
- **gyroAngle** – The angle reported by the gyroscope.
- **wheelSpeeds** – The current wheel speeds.

Returns The new pose of the robot.

1.6.7 MecanumDriveWheelSpeeds

class wpilib.kinematics.**MecanumDriveWheelSpeeds**(*frontLeft: meters_per_second = 0*, *frontRight: meters_per_second = 0*, *rearLeft: meters_per_second = 0*, *rearRight: meters_per_second = 0*) → None

Bases: pybind11_builtins.pybind11_object

Represents the wheel speeds for a mecanum drive drivetrain.

static fromFeet(*frontLeft: feet_per_second*, *frontRight: feet_per_second*, *rearLeft: feet_per_second*, *rearRight: feet_per_second*) → wpilib.kinematics._kinematics.MecanumDriveWheelSpeeds

frontLeft
Speed of the front-left wheel.

frontLeft_fps

frontRight
Speed of the front-right wheel.

frontRight_fps

normalize(*attainableMaxSpeed: meters_per_second*) → None
Normalizes the wheel speeds using some max attainable speed. Sometimes, after inverse kinematics, the requested speed from a/several modules may be above the max attainable speed for the driving motor on that module. To fix this issue, one can "normalize" all the wheel speeds to make sure that all requested module speeds are below the absolute threshold, while maintaining the ratio of speeds between modules.

Parameters **attainableMaxSpeed** – The absolute max speed that a wheel can reach.

rearLeft
Speed of the rear-left wheel.

rearLeft_fps

rearRight
Speed of the rear-right wheel.

rearRight_fps

1.6.8 SwerveDrive3Kinematics

class wpilib.kinematics.**SwerveDrive3Kinematics**(*arg0: wpilib.geometry._geometry.Translation2d*, *arg1: wpilib.geometry._geometry.Translation2d*, *arg2: wpilib.geometry._geometry.Translation2d*) → None

Bases: pybind11_builtins.pybind11_object

Helper class that converts a chassis velocity (dx, dy, and dtheta components) into individual module states (speed and angle).

The inverse kinematics (converting from a desired chassis velocity to individual module states) uses the relative locations of the modules with respect to the center of rotation. The center of rotation for inverse kinematics is also variable. This means that you can set your set your center of rotation in a corner of the robot to perform special evasion manuevers.

Forward kinematics (converting an array of module states into the overall chassis motion) is performs the exact opposite of what inverse kinematics does. Since this is an overdetermined system (more equations than variables), we use a least-squares approximation.

The inverse kinematics: [moduleStates] = [moduleLocations] * [chassisSpeeds] We take the Moore-Penrose pseudoinverse of [moduleLocations] and then multiply by [moduleStates] to get our chassis speeds.

Forward kinematics is also used for odometry – determining the position of the robot on the field using encoders and a gyro.

static normalizeWheelSpeeds(*moduleStates: List[frc::SwerveModuleState[3]]*, *attainableMaxSpeed: meters_per_second*) → List[frc::SwerveModuleState[3]]

Normalizes the wheel speeds using some max attainable speed. Sometimes, after inverse kinematics, the requested speed from a/several modules may be above the max attainable speed for the driving motor on that module. To fix this issue, one can "normalize" all the wheel speeds to make sure that all requested module speeds are below the absolute threshold, while maintaining the ratio of speeds between modules.

Parameters

- **moduleStates** – Reference to array of module states. The array will be mutated with the normalized speeds!
- **attainableMaxSpeed** – The absolute max speed that a module can reach.

toChassisSpeeds() → wpilib.kinematics._kinematics.ChassisSpeeds

Performs forward kinematics to return the resulting chassis state from the given module states. This method is often used for odometry – determining the robot's position on the field using data from the real-world speed and angle of each module on the robot.

Parameters **moduleStates** – The state of the modules as an std::array of type SwerveModuleState, NumModules long as measured from respective encoders and gyros. The order of the swerve module states should be same as passed into the constructor of this class.

Returns The resulting chassis speed.

toSwerveModuleStates(*chassisSpeeds: wpilib.kinematics._kinematics.ChassisSpeeds*, *centerOfRotation: wpilib.geometry._geometry.Translation2d = Translation2d(x=0.000000, y=0.000000)*) → List[frc::SwerveModuleState[3]]

Performs inverse kinematics to return the module states from a desired chassis velocity. This method is often used to convert joystick values into module speeds and angles.

This function also supports variable centers of rotation. During normal operations, the center of rotation is usually the same as the physical center of the robot; therefore, the argument is defaulted to that use case. However, if you wish to change the center of rotation for evasive manuevers, vision alignment, or for any other use case, you can do so.

Parameters

- **chassisSpeeds** – The desired chassis speed.
- **centerOfRotation** – The center of rotation. For example, if you set the center of rotation at one corner of the robot and provide a chassis speed that only has a dtheta component, the robot will rotate around that corner.

Returns

An array containing the module states. Use caution because these module states are not normalized. Sometimes, a user input may cause one of the module speeds to go above the attainable max velocity. Use the <NormalizeWheelSpeeds> function to rectify this issue. In addition, you can leverage the power of C++17 to directly assign the module states to variables:

@code{.cpp} auto [fl, fr, bl, br] = kinematics.ToSwerveModuleStates(chassisSpeeds); @endcode

1.6.9 SwerveDrive3Odometry

class wpilib.kinematics.**SwerveDrive3Odometry**(*kinematics: wpilib.kinematics._kinematics.SwerveDrive3Kinematics*, *gyroAngle: wpilib.geometry._geometry.Rotation2d*, *initialPose: wpilib.geometry._geometry.Pose2d = Pose2d(Translation2d(x=0.000000, y=0.000000), Rotation2d(0.000000))*) → None

Bases: pybind11_builtins.pybind11_object

Class for swerve drive odometry. Odometry allows you to track the robot's position on the field over a course of a match using readings from your swerve drive encoders and swerve azimuth encoders.

Teams can use odometry during the autonomous period for complex tasks like path following. Furthermore, odometry can be used for latency compensation when using computer-vision systems.

Constructs a SwerveDriveOdometry object.

Parameters

- **kinematics** – The swerve drive kinematics for your drivetrain.
- **gyroAngle** – The angle reported by the gyroscope.
- **initialPose** – The starting position of the robot on the field.

getPose() → wpilib.geometry._geometry.Pose2d

Returns the position of the robot on the field.

Returns The pose of the robot.

resetPosition(*pose: wpilib.geometry._geometry.Pose2d*, *gyroAngle: wpilib.geometry._geometry.Rotation2d*) → None

Resets the robot's position on the field.

The gyroscope angle does not need to be reset here on the user's robot code. The library automatically takes care of offsetting the gyro angle.

Parameters

- **pose** – The position on the field that your robot is at.

- **gyroAngle** – The angle reported by the gyroscope.

update(*arg0: wpilib.geometry._geometry.Rotation2d*, *arg1: frc::SwerveModuleState*, *arg2: frc::SwerveModuleState*, *arg3: frc::SwerveModuleState*) → wpilib.geometry._geometry.Pose2d

Updates the robot's position on the field using forward kinematics and integration of the pose over time. This method automatically calculates the current time to calculate period (difference between two timestamps). The period is used to calculate the change in distance from a velocity. This also takes in an angle parameter which is used instead of the angular rate that is calculated from forward kinematics.

Parameters

- **gyroAngle** – The angle reported by the gyroscope.
- **moduleStates** – The current state of all swerve modules. Please provide the states in the same order in which you instantiated your SwerveDriveKinematics.

Returns The new pose of the robot.

updateWithTime(*arg0: seconds*, *arg1: wpilib.geometry._geometry.Rotation2d*, *arg2: frc::SwerveModuleState*, *arg3: frc::SwerveModuleState*, *arg4: frc::SwerveModuleState*) → wpilib.geometry._geometry.Pose2d

Updates the robot's position on the field using forward kinematics and integration of the pose over time. This method takes in the current time as a parameter to calculate period (difference between two timestamps). The period is used to calculate the change in distance from a velocity. This also takes in an angle parameter which is used instead of the angular rate that is calculated from forward kinematics.

Parameters

- **currentTime** – The current time.
- **gyroAngle** – The angle reported by the gyroscope.
- **moduleStates** – The current state of all swerve modules. Please provide the states in the same order in which you instantiated your SwerveDriveKinematics.

Returns The new pose of the robot.

1.6.10 SwerveDrive4Kinematics

class wpilib.kinematics.**SwerveDrive4Kinematics**(*arg0: wpilib.geometry._geometry.Translation2d*, *arg1: wpilib.geometry._geometry.Translation2d*, *arg2: wpilib.geometry._geometry.Translation2d*, *arg3: wpilib.geometry._geometry.Translation2d*) → None

Bases: pybind11_builtins.pybind11_object

Helper class that converts a chassis velocity (dx, dy, and dtheta components) into individual module states (speed and angle).

The inverse kinematics (converting from a desired chassis velocity to individual module states) uses the relative locations of the modules with respect to the center of rotation. The center of rotation for inverse kinematics is also variable. This means that you can set your set your center of rotation in a corner of the robot to perform special evasion manuevers.

Forward kinematics (converting an array of module states into the overall chassis motion) is performs the exact opposite of what inverse kinematics does. Since this is an overdetermined system (more equations than variables), we use a least-squares approximation.

The inverse kinematics: [moduleStates] = [moduleLocations] * [chassisSpeeds] We take the Moore-Penrose pseudoinverse of [moduleLocations] and then multiply by [moduleStates] to get our chassis speeds.

Forward kinematics is also used for odometry – determining the position of the robot on the field using encoders and a gyro.

static normalizeWheelSpeeds(*moduleStates: List[frc::SwerveModuleState[4]], attainableMaxSpeed: meters_per_second*) → List[frc::SwerveModuleState[4]]

Normalizes the wheel speeds using some max attainable speed. Sometimes, after inverse kinematics, the requested speed from a/several modules may be above the max attainable speed for the driving motor on that module. To fix this issue, one can "normalize" all the wheel speeds to make sure that all requested module speeds are below the absolute threshold, while maintaining the ratio of speeds between modules.

Parameters

- **moduleStates** – Reference to array of module states. The array will be mutated with the normalized speeds!
- **attainableMaxSpeed** – The absolute max speed that a module can reach.

toChassisSpeeds() → wpilib.kinematics._kinematics.ChassisSpeeds

Performs forward kinematics to return the resulting chassis state from the given module states. This method is often used for odometry – determining the robot's position on the field using data from the real-world speed and angle of each module on the robot.

Parameters **moduleStates** – The state of the modules as an std::array of type SwerveModuleState, NumModules long as measured from respective encoders and gyros. The order of the swerve module states should be same as passed into the constructor of this class.

Returns The resulting chassis speed.

toSwerveModuleStates(*chassisSpeeds: wpilib.kinematics._kinematics.ChassisSpeeds, centerOfRotation: wpilib.geometry._geometry.Translation2d = Translation2d(x=0.000000, y=0.000000)*) → List[frc::SwerveModuleState[4]]

Performs inverse kinematics to return the module states from a desired chassis velocity. This method is often used to convert joystick values into module speeds and angles.

This function also supports variable centers of rotation. During normal operations, the center of rotation is usually the same as the physical center of the robot; therefore, the argument is defaulted to that use case. However, if you wish to change the center of rotation for evasive manuevers, vision alignment, or for any other use case, you can do so.

Parameters

- **chassisSpeeds** – The desired chassis speed.
- **centerOfRotation** – The center of rotation. For example, if you set the center of rotation at one corner of the robot and provide a chassis speed that only has a dtheta component, the robot will rotate around that corner.

Returns

An array containing the module states. Use caution because these module states are not normalized. Sometimes, a user input may cause one of the module speeds to go above the attainable max velocity. Use the <NormalizeWheelSpeeds> function to rectify this issue. In addition, you can leverage the power of C++17 to directly assign the module states to variables:

@code{.cpp} auto [fl, fr, bl, br] = kinematics.ToSwerveModuleStates(chassisSpeeds); @endcode

1.6.11 SwerveDrive4Odometry

class wpilib.kinematics.**SwerveDrive4Odometry**(*kinematics: wpilib.kinematics._kinematics.SwerveDrive4Kinematics*, *gyroAngle: wpilib.geometry._geometry.Rotation2d*, *initialPose: wpilib.geometry._geometry.Pose2d = Pose2d(Translation2d(x=0.000000, y=0.000000), Rotation2d(0.000000))*) → None

Bases: pybind11_builtins.pybind11_object

Class for swerve drive odometry. Odometry allows you to track the robot's position on the field over a course of a match using readings from your swerve drive encoders and swerve azimuth encoders.

Teams can use odometry during the autonomous period for complex tasks like path following. Furthermore, odometry can be used for latency compensation when using computer-vision systems.

Constructs a SwerveDriveOdometry object.

Parameters

- **kinematics** – The swerve drive kinematics for your drivetrain.
- **gyroAngle** – The angle reported by the gyroscope.
- **initialPose** – The starting position of the robot on the field.

getPose() → wpilib.geometry._geometry.Pose2d

Returns the position of the robot on the field.

Returns The pose of the robot.

resetPosition(*pose: wpilib.geometry._geometry.Pose2d*, *gyroAngle: wpilib.geometry._geometry.Rotation2d*) → None

Resets the robot's position on the field.

The gyroscope angle does not need to be reset here on the user's robot code. The library automatically takes care of offsetting the gyro angle.

Parameters

- **pose** – The position on the field that your robot is at.
- **gyroAngle** – The angle reported by the gyroscope.

update(*arg0: wpilib.geometry._geometry.Rotation2d*, *arg1: frc::SwerveModuleState*, *arg2: frc::SwerveModuleState*, *arg3: frc::SwerveModuleState*, *arg4: frc::SwerveModuleState*) → wpilib.geometry._geometry.Pose2d

Updates the robot's position on the field using forward kinematics and integration of the pose over time. This method automatically calculates the current time to calculate period (difference between two timestamps). The period is used to calculate the change in distance from a velocity. This also takes in an angle parameter which is used instead of the angular rate that is calculated from forward kinematics.

Parameters

- **gyroAngle** – The angle reported by the gyroscope.
- **moduleStates** – The current state of all swerve modules. Please provide the states in the same order in which you instantiated your SwerveDriveKinematics.

Returns The new pose of the robot.

updateWithTime(*arg0: seconds*, *arg1: wpilib.geometry._geometry.Rotation2d*, *arg2: frc::SwerveModuleState*, *arg3: frc::SwerveModuleState*, *arg4: frc::SwerveModuleState*, *arg5: frc::SwerveModuleState*) → wpilib.geometry._geometry.Pose2d

Updates the robot's position on the field using forward kinematics and integration of the pose over time. This method takes in the current time as a parameter to calculate period (difference between two timestamps). The period is used to calculate the change in distance from a velocity. This also takes in an angle parameter which is used instead of the angular rate that is calculated from forward kinematics.

Parameters

- **currentTime** – The current time.
- **gyroAngle** – The angle reported by the gyroscope.
- **moduleStates** – The current state of all swerve modules. Please provide the states in the same order in which you instantiated your SwerveDriveKinematics.

Returns The new pose of the robot.

1.6.12 SwerveDrive6Kinematics

class wpilib.kinematics.**SwerveDrive6Kinematics**(*arg0: wpilib.geometry._geometry.Translation2d*, *arg1: wpilib.geometry._geometry.Translation2d*, *arg2: wpilib.geometry._geometry.Translation2d*, *arg3: wpilib.geometry._geometry.Translation2d*, *arg4: wpilib.geometry._geometry.Translation2d*, *arg5: wpilib.geometry._geometry.Translation2d*) → None

Bases: pybind11_builtins.pybind11_object

Helper class that converts a chassis velocity (dx, dy, and dtheta components) into individual module states (speed and angle).

The inverse kinematics (converting from a desired chassis velocity to individual module states) uses the relative locations of the modules with respect to the center of rotation. The center of rotation for inverse kinematics is also variable. This means that you can set your set your center of rotation in a corner of the robot to perform special evasion manuevers.

Forward kinematics (converting an array of module states into the overall chassis motion) is performs the exact opposite of what inverse kinematics does. Since this is an overdetermined system (more equations than variables), we use a least-squares approximation.

The inverse kinematics: [moduleStates] = [moduleLocations] * [chassisSpeeds] We take the Moore-Penrose pseudoinverse of [moduleLocations] and then multiply by [moduleStates] to get our chassis speeds.

Forward kinematics is also used for odometry – determining the position of the robot on the field using encoders and a gyro.

static normalizeWheelSpeeds(*moduleStates: List[frc::SwerveModuleState[6]]*, *attainableMaxSpeed: meters_per_second*) → List[frc::SwerveModuleState[6]]

Normalizes the wheel speeds using some max attainable speed. Sometimes, after inverse kinematics, the requested speed from a/several modules may be above the max attainable speed for the driving motor on that module. To fix this issue, one can "normalize" all the wheel speeds to make sure that all requested module speeds are below the absolute threshold, while maintaining the ratio of speeds between modules.

Parameters

- **moduleStates** – Reference to array of module states. The array will be mutated with the normalized speeds!
- **attainableMaxSpeed** – The absolute max speed that a module can reach.

toChassisSpeeds() → wpilib.kinematics._kinematics.ChassisSpeeds

Performs forward kinematics to return the resulting chassis state from the given module states. This method is often used for odometry – determining the robot's position on the field using data from the real-world speed and angle of each module on the robot.

Parameters **moduleStates** – The state of the modules as an std::array of type SwerveModuleState, NumModules long as measured from respective encoders and gyros. The order of the swerve module states should be same as passed into the constructor of this class.

Returns The resulting chassis speed.

toSwerveModuleStates(*chassisSpeeds: wpilib.kinematics._kinematics.ChassisSpeeds*, *centerOfRotation: wpilib.geometry._geometry.Translation2d = Translation2d(x=0.000000, y=0.000000)*) → List[frc::SwerveModuleState[6]]

Performs inverse kinematics to return the module states from a desired chassis velocity. This method is often used to convert joystick values into module speeds and angles.

This function also supports variable centers of rotation. During normal operations, the center of rotation is usually the same as the physical center of the robot; therefore, the argument is defaulted to that use case. However, if you wish to change the center of rotation for evasive manuevers, vision alignment, or for any other use case, you can do so.

Parameters

- **chassisSpeeds** – The desired chassis speed.
- **centerOfRotation** – The center of rotation. For example, if you set the center of rotation at one corner of the robot and provide a chassis speed that only has a dtheta component, the robot will rotate around that corner.

Returns

An array containing the module states. Use caution because these module states are not normalized. Sometimes, a user input may cause one of the module speeds to go above the attainable max velocity. Use the <NormalizeWheelSpeeds> function to rectify this issue. In addition, you can leverage the power of C++17 to directly assign the module states to variables:

@code{.cpp} auto [fl, fr, bl, br] = kinematics.ToSwerveModuleStates(chassisSpeeds); @endcode

1.6.13 SwerveDrive6Odometry

class wpilib.kinematics.**SwerveDrive6Odometry**(*kinematics: wpilib.kinematics._kinematics.SwerveDrive6Kinematics*, *gyroAngle: wpilib.geometry._geometry.Rotation2d*, *initialPose: wpilib.geometry._geometry.Pose2d = Pose2d(Translation2d(x=0.000000, y=0.000000), Rotation2d(0.000000))*) → None

Bases: pybind11_builtins.pybind11_object

Class for swerve drive odometry. Odometry allows you to track the robot's position on the field over a course of a match using readings from your swerve drive encoders and swerve azimuth encoders.

Teams can use odometry during the autonomous period for complex tasks like path following. Furthermore, odometry can be used for latency compensation when using computer-vision systems.

Constructs a SwerveDriveOdometry object.

Parameters

- **kinematics** – The swerve drive kinematics for your drivetrain.
- **gyroAngle** – The angle reported by the gyroscope.
- **initialPose** – The starting position of the robot on the field.

getPose() → wpilib.geometry._geometry.Pose2d
Returns the position of the robot on the field.

Returns The pose of the robot.

resetPosition(*pose: wpilib.geometry._geometry.Pose2d*, *gyroAngle: wpilib.geometry._geometry.Rotation2d*) → None
Resets the robot's position on the field.

The gyroscope angle does not need to be reset here on the user's robot code. The library automatically takes care of offsetting the gyro angle.

Parameters

- **pose** – The position on the field that your robot is at.
- **gyroAngle** – The angle reported by the gyroscope.

update(*arg0: wpilib.geometry._geometry.Rotation2d*, *arg1: frc::SwerveModuleState*, *arg2: frc::SwerveModuleState*, *arg3: frc::SwerveModuleState*, *arg4: frc::SwerveModuleState*, *arg5: frc::SwerveModuleState*, *arg6: frc::SwerveModuleState*) → wpilib.geometry._geometry.Pose2d
Updates the robot's position on the field using forward kinematics and integration of the pose over time. This method automatically calculates the current time to calculate period (difference between two timestamps). The period is used to calculate the change in distance from a velocity. This also takes in an angle parameter which is used instead of the angular rate that is calculated from forward kinematics.

Parameters

- **gyroAngle** – The angle reported by the gyroscope.
- **moduleStates** – The current state of all swerve modules. Please provide the states in the same order in which you instantiated your SwerveDriveKinematics.

Returns The new pose of the robot.

updateWithTime(*arg0: seconds*, *arg1: wpilib.geometry._geometry.Rotation2d*, *arg2: frc::SwerveModuleState*, *arg3: frc::SwerveModuleState*, *arg4: frc::SwerveModuleState*, *arg5: frc::SwerveModuleState*, *arg6: frc::SwerveModuleState*, *arg7: frc::SwerveModuleState*) → wpilib.geometry._geometry.Pose2d
Updates the robot's position on the field using forward kinematics and integration of the pose over time. This method takes in the current time as a parameter to calculate period (difference between two timestamps). The period is used to calculate the change in distance from a velocity. This also takes in an angle parameter which is used instead of the angular rate that is calculated from forward kinematics.

Parameters

- **currentTime** – The current time.

- **gyroAngle** – The angle reported by the gyroscope.
- **moduleStates** – The current state of all swerve modules. Please provide the states in the same order in which you instantiated your SwerveDriveKinematics.

Returns The new pose of the robot.

1.6.14 SwerveModuleState

class wpilib.kinematics.**SwerveModuleState**(*speed: meters_per_second = 0*, *angle: wpilib.geometry._geometry.Rotation2d = Rotation2d(0.000000)*) → None

Bases: pybind11_builtins.pybind11_object

Represents the state of one swerve module.

angle
Angle of the module.

speed
Speed of the wheel of the module.

speed_fps

1.7 wpilib.simulation Package

wpilib.simulation.Field2d(self)	2D representation of game field (for simulation).

1.7.1 Field2d

class wpilib.simulation.**Field2d**() → None

Bases: pybind11_builtins.pybind11_object

2D representation of game field (for simulation).

In non-simulation mode this simply stores and returns the robot pose.

The robot pose is the actual location shown on the simulation view. This may or may not match the robot's internal odometry. For example, if the robot is shown at a particular starting location, the pose in this class would represent the actual location on the field, but the robot's internal state might have a 0,0,0 pose (unless it's initialized to something different).

As the user is able to edit the pose, code performing updates should get the robot pose, transform it as appropriate (e.g. based on simulated wheel velocity), and set the new pose.

getRobotPose() → wpilib.geometry._geometry.Pose2d
Get the robot pose.

Returns 2D pose

setRobotPose(**args*, ***kwargs*)
Overloaded function.

1. setRobotPose(self: wpilib.simulation._simulation.Field2d, pose: wpilib.geometry._geometry.Pose2d) -> None

Set the robot pose from a Pose object.

Parameters **pose** – 2D pose

2. setRobotPose(self: wpilib.simulation._simulation.Field2d, x: meters, y: meters, rotation: wpilib.geometry._geometry.Rotation2d) -> None

Set the robot pose from x, y, and rotation.

Parameters

- **x** – X location
- **y** – Y location
- **rotation** – rotation

1.8 wpilib.spline Package

wpilib.spline.CubicHermiteSpline(self, ...)	Represents a hermite spline of degree 3.
wpilib.spline.QuinticHermiteSpline(self, ...)	Represents a hermite spline of degree 5.
wpilib.spline.Spline3(self)	Represents a two-dimensional parametric spline that interpolates between two points.
wpilib.spline.Spline5(self)	Represents a two-dimensional parametric spline that interpolates between two points.
wpilib.spline.SplineHelper(self)	Helper class that is used to generate cubic and quintic splines from user provided waypoints.
wpilib.spline.SplineParameterizer	Class used to parameterize a spline by its arc length.

1.8.1 CubicHermiteSpline

class wpilib.spline.**CubicHermiteSpline**() → None

Bases: *wpilib.spline.Spline3*

Represents a hermite spline of degree 3.

Constructs a cubic hermite spline with the specified control vectors. Each control vector contains info about the location of the point and its first derivative.

Parameters

- **xInitialControlVector** – The control vector for the initial point in the x dimension.
- **xFinalControlVector** – The control vector for the final point in the x dimension.
- **yInitialControlVector** – The control vector for the initial point in the y dimension.
- **yFinalControlVector** – The control vector for the final point in the y dimension.

1.8.2 QuinticHermiteSpline

class wpilib.spline.**QuinticHermiteSpline**() → None

Bases: *wpilib.spline.Spline5*

Represents a hermite spline of degree 5.

Constructs a quintic hermite spline with the specified control vectors. Each control vector contains into about the location of the point, its first derivative, and its second derivative.

Parameters

- **xInitialControlVector** – The control vector for the initial point in the x dimension.
- **xFinalControlVector** – The control vector for the final point in the x dimension.
- **yInitialControlVector** – The control vector for the initial point in the y dimension.
- **yFinalControlVector** – The control vector for the final point in the y dimension.

1.8.3 Spline3

class wpilib.spline.**Spline3**() → None
Bases: pybind11_builtins.pybind11_object

Represents a two-dimensional parametric spline that interpolates between two points.

@tparam Degree The degree of the spline.

class ControlVector(**args*, ***kwargs*)
Bases: pybind11_builtins.pybind11_object

Represents a control vector for a spline.

Each element in each array represents the value of the derivative at the index. For example, the value of x[2] is the second derivative in the x dimension.

Overloaded function.

1. __init__(self: wpilib.spline._spline.Spline3.ControlVector) -> None
2. __init__(self: wpilib.spline._spline.Spline3.ControlVector, x: List[float[2]], y: List[float[2]]) -> None

x

y

getPoint(*t: float*) → Tuple[wpilib.geometry._geometry.Pose2d, radians_per_meter]
Gets the pose and curvature at some point t on the spline.

Parameters **t** – The point t

Returns The pose and curvature at that point.

1.8.4 Spline5

class wpilib.spline.**Spline5**() → None
Bases: pybind11_builtins.pybind11_object

Represents a two-dimensional parametric spline that interpolates between two points.

@tparam Degree The degree of the spline.

class ControlVector(**args*, ***kwargs*)
Bases: pybind11_builtins.pybind11_object

Represents a control vector for a spline.

Each element in each array represents the value of the derivative at the index. For example, the value of x[2] is the second derivative in the x dimension.

Overloaded function.

1. __init__(self: wpilib.spline._spline.Spline5.ControlVector) -> None
2. __init__(self: wpilib.spline._spline.Spline5.ControlVector, x: List[float[3]], y: List[float[3]]) -> None

x

y

getPoint(*t: float*) → Tuple[wpilib.geometry._geometry.Pose2d, radians_per_meter]
Gets the pose and curvature at some point t on the spline.

Parameters **t** – The point t

Returns The pose and curvature at that point.

1.8.5 SplineHelper

class wpilib.spline.**SplineHelper**() → None
Bases: pybind11_builtins.pybind11_object

Helper class that is used to generate cubic and quintic splines from user provided waypoints.

static cubicControlVectorsFromWaypoints(*start: wpilib.geometry._geometry.Pose2d, interiorWaypoints: List[wpilib.geometry._geometry.Translation2d], end: wpilib.geometry._geometry.Pose2d*) → List[wpilib.spline._spline.Spline3.ControlVector[2]]
Returns 2 cubic control vectors from a set of exterior waypoints and interior translations.

Parameters

- **start** – The starting pose.
- **interiorWaypoints** – The interior waypoints.
- **end** – The ending pose.

Returns 2 cubic control vectors.

static cubicSplinesFromControlVectors(*start: wpilib.spline._spline.Spline3.ControlVector, waypoints: List[wpilib.geometry._geometry.Translation2d], end: wpilib.spline._spline.Spline3.ControlVector*) → List[frc::CubicHermiteSpline]
Returns a set of cubic splines corresponding to the provided control vectors. The user is free to set the direction of the start and end point. The directions for the middle waypoints are determined automatically to ensure continuous curvature throughout the path.

The derivation for the algorithm used can be found here: <https://www.uio.no/studier/emner/matnat/ifi/nedlagte-emner/INF-MAT4350/h08/undervisningsmateriale/chap7alecture.pdf>

Parameters

- **start** – The starting control vector.
- **waypoints** – The middle waypoints. This can be left blank if you only wish to create a path with two waypoints.
- **end** – The ending control vector.

Returns A vector of cubic hermite splines that interpolate through the provided waypoints.

static quinticControlVectorsFromWaypoints(*waypoints: List[wpilib.geometry._geometry.Pose2d]*) → List[wpilib.spline._spline.Spline5.ControlVector]

Returns quintic control vectors from a set of waypoints.

Parameters **waypoints** – The waypoints

Returns List of control vectors

static quinticSplinesFromControlVectors(*controlVectors: List[wpilib.spline._spline.Spline5.ControlVector]*) → List[frc::QuinticHermiteSpline]

Returns a set of quintic splines corresponding to the provided control vectors. The user is free to set the direction of all waypoints. Continuous curvature is guaranteed throughout the path.

Parameters **controlVectors** – The control vectors.

Returns A vector of quintic hermite splines that interpolate through the provided waypoints.

1.8.6 SplineParameterizer

class wpilib.spline.**SplineParameterizer**

Bases: pybind11_builtins.pybind11_object

Class used to parameterize a spline by its arc length.

static parameterize(**args*, ***kwargs*)

Overloaded function.

1. parameterize(spline: wpilib.spline._spline.Spline3, t0: float = 0.0, t1: float = 1.0) -> List[Tuple[wpilib.geometry._geometry.Pose2d, radians_per_meter]]

Parameterizes the spline. This method breaks up the spline into various arcs until their dx, dy, and dtheta are within specific tolerances.

Parameters

- **spline** – The spline to parameterize.
- **t0** – Starting internal spline parameter. It is recommended to leave this as default.
- **t1** – Ending internal spline parameter. It is recommended to leave this as default.

Returns A vector of poses and curvatures that represents various points on the spline.

2. parameterize(spline: wpilib.spline._spline.Spline5, t0: float = 0.0, t1: float = 1.0) -> List[Tuple[wpilib.geometry._geometry.Pose2d, radians_per_meter]]

Parameterizes the spline. This method breaks up the spline into various arcs until their dx, dy, and dtheta are within specific tolerances.

Parameters

- **spline** – The spline to parameterize.
- **t0** – Starting internal spline parameter. It is recommended to leave this as default.
- **t1** – Ending internal spline parameter. It is recommended to leave this as default.

Returns A vector of poses and curvatures that represents various points on the spline.

1.9 wpilib.trajectory Package

wpilib.trajectory.Trajectory(*args, **kwargs)	Represents a time-parameterized trajectory.
wpilib.trajectory.TrajectoryConfig(self, ...)	Represents the configuration for generating a trajectory.
wpilib.trajectory.TrajectoryGenerator(self)	Helper class used to generate trajectories with various constraints.
wpilib.trajectory.TrajectoryParameterizer(self)	Class used to parameterize a trajectory by time.
wpilib.trajectory.TrajectoryUtil	
wpilib.trajectory.TrapezoidProfile(self, ...)	A trapezoid-shaped velocity profile.

1.9.1 Trajectory

class wpilib.trajectory.**Trajectory**(**args*, ***kwargs*)

Bases: pybind11_builtins.pybind11_object

Represents a time-parameterized trajectory. The trajectory contains of various States that represent the pose, curvature, time elapsed, velocity, and acceleration at that point.

Overloaded function.

1. __init__(self: wpilib.trajectory._trajectory.Trajectory) -> None
2. __init__(self: wpilib.trajectory._trajectory.Trajectory, states: List[wpilib.trajectory._trajectory.Trajectory.State]) -> None

Constructs a trajectory from a vector of states.

class State(*t: seconds = 0.0*, *velocity: meters_per_second = 0.0*, *acceleration: meters_per_second_squared = 0.0*, *pose: wpilib.geometry._geometry.Pose2d = Pose2d(Translation2d(x=0.000000, y=0.000000), Rotation2d(0.000000))*, *curvature: radians_per_meter = 0.0*) → None

Bases: pybind11_builtins.pybind11_object

Represents one point on the trajectory.

acceleration

curvature

interpolate(*endValue: wpilib.trajectory._trajectory.Trajectory.State*, *i: float*) → wpilib.trajectory._trajectory.Trajectory.State

Interpolates between two States.

Parameters

- **endValue** – The end value for the interpolation.
- **i** – The interpolant (fraction).

Returns The interpolated state.

pose

t

velocity

initialPose() → wpilib.geometry._geometry.Pose2d

Returns the initial pose of the trajectory.

Returns The initial pose of the trajectory.

relativeTo(*pose: wpilib.geometry._geometry.Pose2d*) → wpilib.trajectory._trajectory.Trajectory
Transforms all poses in the trajectory so that they are relative to the given pose. This is useful for converting a field-relative trajectory into a robot-relative trajectory.

Parameters **pose** – The pose that is the origin of the coordinate frame that the current trajectory will be transformed into.

Returns The transformed trajectory.

sample(*t: seconds*) → wpilib.trajectory._trajectory.Trajectory.State
Sample the trajectory at a point in time.

Parameters **t** – The point in time since the beginning of the trajectory to sample.

Returns The state at that point in time.

states() → List[wpilib.trajectory._trajectory.Trajectory.State]
Return the states of the trajectory.

Returns The states of the trajectory.

totalTime() → seconds
Returns the overall duration of the trajectory.

Returns The duration of the trajectory.

transformBy(*transform: wpilib.geometry._geometry.Transform2d*) → wpilib.trajectory._trajectory.Trajectory
Transforms all poses in the trajectory by the given transform. This is useful for converting a robot-relative trajectory into a field-relative trajectory. This works with respect to the first pose in the trajectory.

Parameters **transform** – The transform to transform the trajectory by.

Returns The transformed trajectory.

1.9.2 TrajectoryConfig

class wpilib.trajectory.**TrajectoryConfig**(*maxVelocity: meters_per_second*, *maxAcceleration: meters_per_second_squared*) → None
Bases: pybind11_builtins.pybind11_object

Represents the configuration for generating a trajectory. This class stores the start velocity, end velocity, max velocity, max acceleration, custom constraints, and the reversed flag.

The class must be constructed with a max velocity and max acceleration. The other parameters (start velocity, end velocity, constraints, reversed) have been defaulted to reasonable values (0, 0, {}, false). These values can be changed via the SetXXX methods.

Constructs a config object.

Parameters

- **maxVelocity** – The max velocity of the trajectory.
- **maxAcceleration** – The max acceleration of the trajectory.

addConstraint(**args*, ***kwargs*)
Overloaded function.

1. addConstraint(self: wpilib.trajectory._trajectory.TrajectoryConfig, constraint: wpilib.trajectory.constraint._constraint.CentripetalAccelerationConstraint) -> None

Adds a user-defined constraint to the trajectory.

Parameters **constraint** – The user-defined constraint.

2. addConstraint(self: wpilib.trajectory._trajectory.TrajectoryConfig, constraint: wpilib.trajectory.constraint._constraint.DifferentialDriveKinematicsConstraint) -> None

Adds a user-defined constraint to the trajectory.

Parameters **constraint** – The user-defined constraint.

3. addConstraint(self: wpilib.trajectory._trajectory.TrajectoryConfig, constraint: wpilib.trajectory.constraint._constraint.DifferentialDriveVoltageConstraint) -> None

Adds a user-defined constraint to the trajectory.

Parameters **constraint** – The user-defined constraint.

4. addConstraint(self: wpilib.trajectory._trajectory.TrajectoryConfig, constraint: wpilib.trajectory.constraint._constraint.MecanumDriveKinematicsConstraint) -> None

Adds a user-defined constraint to the trajectory.

Parameters **constraint** – The user-defined constraint.

endVelocity() → meters_per_second
Returns the ending velocity of the trajectory.

Returns The ending velocity of the trajectory.

isReversed() → bool
Returns whether the trajectory is reversed or not.

Returns whether the trajectory is reversed or not.

maxAcceleration() → meters_per_second_squared
Returns the maximum acceleration of the trajectory.

Returns The maximum acceleration of the trajectory.

maxVelocity() → meters_per_second
Returns the maximum velocity of the trajectory.

Returns The maximum velocity of the trajectory.

setEndVelocity(*endVelocity: meters_per_second*) → None
Sets the end velocity of the trajectory.

Parameters **endVelocity** – The end velocity of the trajectory.

setKinematics(**args*, ***kwargs*)
Overloaded function.

1. setKinematics(self: wpilib.trajectory._trajectory.TrajectoryConfig, kinematics: wpilib.kinematics._kinematics.DifferentialDriveKinematics) -> None

Adds a differential drive kinematics constraint to ensure that no wheel velocity of a differential drive goes above the max velocity.

Parameters **kinematics** – The differential drive kinematics.

2. setKinematics(self: wpilib.trajectory._trajectory.TrajectoryConfig, kinematics: wpilib.kinematics._kinematics.MecanumDriveKinematics) -> None

Adds a mecanum drive kinematics constraint to ensure that no wheel velocity of a mecanum drive goes above the max velocity.

Parameters **kinematics** – The mecanum drive kinematics.

setReversed(*reversed: bool*) → None
Sets the reversed flag of the trajectory.

Parameters **reversed** – Whether the trajectory should be reversed or not.

setStartVelocity(*startVelocity: meters_per_second*) → None
Sets the start velocity of the trajectory.

Parameters **startVelocity** – The start velocity of the trajectory.

startVelocity() → meters_per_second
Returns the starting velocity of the trajectory.

Returns The starting velocity of the trajectory.

1.9.3 TrajectoryGenerator

class wpilib.trajectory.**TrajectoryGenerator**() → None
Bases: pybind11_builtins.pybind11_object

Helper class used to generate trajectories with various constraints.

static generateTrajectory(**args*, ***kwargs*)
Overloaded function.

1. generateTrajectory(initial: frc::Spline<3>::ControlVector, interiorWaypoints: List[wpilib.geometry._geometry.Translation2d], end: frc::Spline<3>::ControlVector, config: wpilib.trajectory._trajectory.TrajectoryConfig) -> wpilib.trajectory._trajectory.Trajectory

Generates a trajectory from the given control vectors and config. This method uses clamped cubic splines – a method in which the exterior control vectors and interior waypoints are provided. The headings are automatically determined at the interior points to ensure continuous curvature.

Parameters

- **initial** – The initial control vector.
- **interiorWaypoints** – The interior waypoints.
- **end** – The ending control vector.
- **config** – The configuration for the trajectory.

Returns The generated trajectory.

2. generateTrajectory(start: wpilib.geometry._geometry.Pose2d, interiorWaypoints: List[wpilib.geometry._geometry.Translation2d], end: wpilib.geometry._geometry.Pose2d, config: wpilib.trajectory._trajectory.TrajectoryConfig) -> wpilib.trajectory._trajectory.Trajectory

Generates a trajectory from the given waypoints and config. This method uses clamped cubic splines – a method in which the initial pose, final pose, and interior waypoints are provided. The headings are automatically determined at the interior points to ensure continuous curvature.

Parameters

- **start** – The starting pose.
- **interiorWaypoints** – The interior waypoints.

- **end** – The ending pose.
- **config** – The configuration for the trajectory.

Returns The generated trajectory.

3. generateTrajectory(controlVectors: List[frc::Spline<5>::ControlVector], config: wpilib.trajectory._trajectory.TrajectoryConfig) -> wpilib.trajectory._trajectory.Trajectory

Generates a trajectory from the given quintic control vectors and config. This method uses quintic hermite splines – therefore, all points must be represented by control vectors. Continuous curvature is guaranteed in this method.

Parameters

- **controlVectors** – List of quintic control vectors.
- **config** – The configuration for the trajectory.

Returns The generated trajectory.

4. generateTrajectory(waypoints: List[wpilib.geometry._geometry.Pose2d], config: wpilib.trajectory._trajectory.TrajectoryConfig) -> wpilib.trajectory._trajectory.Trajectory

Generates a trajectory from the given waypoints and config. This method uses quintic hermite splines – therefore, all points must be represented by Pose2d objects. Continuous curvature is guaranteed in this method.

Parameters

- **waypoints** – List of waypoints..
- **config** – The configuration for the trajectory.

Returns The generated trajectory.

static setErrorHandler(*func: Callable[[str], None]*) → None
Set error reporting function. By default, it is output to stderr.

Parameters **func** – Error reporting function.

static splinePointsFromSplines(**args*, ***kwargs*)
Overloaded function.

1. splinePointsFromSplines(splines: List[frc::CubicHermiteSpline]) -> List[Tuple[wpilib.geometry._geometry.Pose2d, radians_per_meter]]

Generate spline points from a vector of splines by parameterizing the splines.

Parameters **splines** – The splines to parameterize.

Returns The spline points for use in time parameterization of a trajectory.

2. splinePointsFromSplines(splines: List[frc::QuinticHermiteSpline]) -> List[Tuple[wpilib.geometry._geometry.Pose2d, radians_per_meter]]

Generate spline points from a vector of splines by parameterizing the splines.

Parameters **splines** – The splines to parameterize.

Returns The spline points for use in time parameterization of a trajectory.

1.9.4 TrajectoryParameterizer

class wpilib.trajectory.**TrajectoryParameterizer**() → None
Bases: pybind11_builtins.pybind11_object

Class used to parameterize a trajectory by time.

static timeParameterizeTrajectory(*points: List[Tuple[wpilib.geometry._geometry.Pose2d, radians_per_meter]]*, *constraints: List[wpilib.trajectory.constraint._constraint.TrajectoryConstraint]*, *startVelocity: meters_per_second*, *endVelocity: meters_per_second*, *maxVelocity: meters_per_second*, *maxAcceleration: meters_per_second_squared*, *reversed: bool*) → None

Parameterize the trajectory by time. This is where the velocity profile is generated.

The derivation of the algorithm used can be found here: <http://www2.informatik.uni-freiburg.de/~lau/students/Sprunk2008.pdf>

Parameters

- **points** – Reference to the spline points.
- **constraints** – A vector of various velocity and acceleration constraints.
- **startVelocity** – The start velocity for the trajectory.
- **endVelocity** – The end velocity for the trajectory.
- **maxVelocity** – The max velocity for the trajectory.
- **maxAcceleration** – The max acceleration for the trajectory.
- **reversed** – Whether the robot should move backwards. Note that the robot will still move from a -> b -> ... -> z as defined in the waypoints.

Returns The trajectory.

1.9.5 TrajectoryUtil

class wpilib.trajectory.**TrajectoryUtil**
Bases: pybind11_builtins.pybind11_object

static deserializeTrajectory(*json_str: str*) → wpilib.trajectory._trajectory.Trajectory
Serializes a Trajectory to PathWeaver-style JSON.

Parameters **trajectory** – the trajectory to export

Returns the string containing the serialized JSON

static fromPathweaverJson(*path: str*) → wpilib.trajectory._trajectory.Trajectory
Imports a Trajectory from a PathWeaver-style JSON file.

Parameters **path** – The path of the json file to import from.

Returns The trajectory represented by the file.

static serializeTrajectory(*trajectory: wpilib.trajectory._trajectory.Trajectory*) → str
Deserializes a Trajectory from PathWeaver-style JSON.

Parameters **json** – the string containing the serialized JSON

Returns the trajectory represented by the JSON

static toPathweaverJson(*trajectory: wpilib.trajectory._trajectory.Trajectory*, *path: str*) → None

Exports a Trajectory to a PathWeaver-style JSON file.

Parameters

- **trajectory** – the trajectory to export
- **path** – the path of the file to export to

Returns The interpolated state.

1.9.6 TrapezoidProfile

class wpilib.trajectory.**TrapezoidProfile**(*constraints: wpilib.trajectory._trajectory.TrapezoidProfile.Constraints*, *goal: wpilib.trajectory._trajectory.TrapezoidProfile.State*, *initial: wpilib.trajectory._trajectory.TrapezoidProfile.State = <wpilib.trajectory._trajectory.TrapezoidProfile.State object at 0x7f744d609420>*) → None

Bases: pybind11_builtins.pybind11_object

A trapezoid-shaped velocity profile.

While this class can be used for a profiled movement from start to finish, the intended usage is to filter a reference's dynamics based on trapezoidal velocity constraints. To compute the reference obeying this constraint, do the following.

Initialization: @code{.cpp} TrapezoidalMotionProfile::Constraints constraints{kMaxV, kMaxA}; double previousProfiledReference = initialReference; @endcode

Run on update: @code{.cpp} TrapezoidalMotionProfile profile{constraints, unprofiledReference, previousProfiledReference}; previousProfiledReference = profile.Calculate(timeSincePreviousUpdate); @endcode

where *unprofiledReference* is free to change between calls. Note that when the unprofiled reference is within the constraints, *Calculate()* returns the unprofiled reference unchanged.

Otherwise, a timer can be started to provide monotonic values for *Calculate()* and to determine when the profile has completed via *IsFinished()*.

Construct a TrapezoidProfile.

Parameters

- **constraints** – The constraints on the profile, like maximum velocity.
- **goal** – The desired state when the profile is complete.
- **initial** – The initial state (usually the current state).

class Constraints(**args*, ***kwargs*)

Bases: pybind11_builtins.pybind11_object

Overloaded function.

1. __init__(self: wpilib.trajectory._trajectory.TrapezoidProfile.Constraints) -> None
2. __init__(self: wpilib.trajectory._trajectory.TrapezoidProfile.Constraints, maxVelocity: units_per_second, maxAcceleration: units_per_second_squared) -> None

class State(*position: float = 0*, *velocity: units_per_second = 0*) → None

Bases: pybind11_builtins.pybind11_object

position

velocity

calculate(*t: seconds*) → wpilib.trajectory._trajectory.TrapezoidProfile.State
Calculate the correct position and velocity for the profile at a time t where the beginning of the profile was at time t = 0.

Parameters **t** – The time since the beginning of the profile.

isFinished(*t: seconds*) → bool
Returns true if the profile has reached the goal.

The profile has reached the goal if the time since the profile started has exceeded the profile's total time.

Parameters **t** – The time since the beginning of the profile.

timeLeftUntil(*target: float*) → seconds
Returns the time left until a target distance in the profile is reached.

Parameters **target** – The target distance.

totalTime() → seconds
Returns the total time the profile takes to reach the goal.

1.10 wpilib.trajectory.constraint Package

wpilib.trajectory.constraint.CentripetalAccelerationConstraint(...)	A constraint on the maximum absolute centripetal acceleration allowed when traversing a trajectory.
wpilib.trajectory.constraint.DifferentialDriveKinematicsConstraint(...)	
wpilib.trajectory.constraint.DifferentialDriveVoltageConstraint(...)	A class that enforces constraints on differential drive voltage expenditure based on the motor dynamics and the drive kinematics.
wpilib.trajectory.constraint.MecanumDriveKinematicsConstraint(...)	
wpilib.trajectory.constraint.SwerveDrive3KinematicsConstraint(...)	
wpilib.trajectory.constraint.SwerveDrive4KinematicsConstraint(...)	
wpilib.trajectory.constraint.SwerveDrive6KinematicsConstraint(...)	
wpilib.trajectory.constraint.TrajectoryConstraint(self)	An interface for defining user-defined velocity and acceleration constraints while generating trajectories.

1.10.1 CentripetalAccelerationConstraint

class wpilib.trajectory.constraint.**CentripetalAccelerationConstraint**(*maxCentripetalAcceleration: meters_per_second_squared*) → None

Bases: *wpilib.trajectory.constraint.TrajectoryConstraint*

A constraint on the maximum absolute centripetal acceleration allowed when traversing a trajectory. The centripetal acceleration of a robot is defined as the velocity squared divided by the radius of curvature.

Effectively, limiting the maximum centripetal acceleration will cause the robot to slow down around tight turns, making it easier to track trajectories with sharp turns.

maxVelocity(*pose: wpilib.geometry._geometry.Pose2d*, *curvature: radians_per_meter*, *velocity: meters_per_second*) → meters_per_second

minMaxAcceleration(*pose: wpilib.geometry._geometry.Pose2d*, *curvature: radians_per_meter*, *speed: meters_per_second*) → wpilib.trajectory.constraint._constraint.TrajectoryConstraint.MinMax

1.10.2 DifferentialDriveKinematicsConstraint

class wpilib.trajectory.constraint.**DifferentialDriveKinematicsConstraint**(*kinematics: wpilib.kinematics._kinema*
maxSpeed: meters_per_second) → None

Bases: *wpilib.trajectory.constraint.TrajectoryConstraint*

maxVelocity(*pose: wpilib.geometry._geometry.Pose2d*, *curvature: radians_per_meter*, *velocity: meters_per_second*) → meters_per_second

minMaxAcceleration(*pose: wpilib.geometry._geometry.Pose2d*, *curvature: radians_per_meter*, *speed: meters_per_second*) → wpilib.trajectory.constraint._constraint.TrajectoryConstraint.MinMax

1.10.3 DifferentialDriveVoltageConstraint

class wpilib.trajectory.constraint.**DifferentialDriveVoltageConstraint**(*feedforward: frc::SimpleMotorFeedforward<1l>, units::base_unit<std::ratio<1l>, std::ratio<0l, 1l>, std::ratio<0l, 1l>, std::ratio<0l, 1l>, std::ratio<0l, 1l>, std::ratio<0l, 1l>, std::ratio<0l, 1l>, std::ratio<0l, 1l>, std::ratio<0l, 1l> >, std::ratio<0l, 1l>, std::ratio<0l, 1l> > >, kinematics: wpilib.kinematics._kinematics.*, *maxVoltage: volts*) → None

Bases: *wpilib.trajectory.constraint.TrajectoryConstraint*

A class that enforces constraints on differential drive voltage expenditure based on the motor dynamics and the drive kinematics. Ensures that the acceleration of any wheel of the robot while following the trajectory is never higher than what can be achieved with the given maximum voltage.

Creates a new DifferentialDriveVoltageConstraint.

Parameters

- **feedforward** – A feedforward component describing the behavior of the drive.
- **kinematics** – A kinematics component describing the drive geometry.
- **maxVoltage** – The maximum voltage available to the motors while following the path. Should be somewhat less than the nominal battery voltage (12V) to account for "voltage sag" due to current draw.

maxVelocity(*pose: wpilib.geometry._geometry.Pose2d*, *curvature: radians_per_meter*, *velocity: meters_per_second*) → meters_per_second

minMaxAcceleration(*pose: wpilib.geometry._geometry.Pose2d*, *curvature: radians_per_meter*, *speed: meters_per_second*) → wpilib.trajectory.constraint._constraint.TrajectoryConstraint.MinMax

1.10.4 MecanumDriveKinematicsConstraint

class wpilib.trajectory.constraint.**MecanumDriveKinematicsConstraint**(*kinematics: wpilib.kinematics._kinematics.Me*
maxSpeed: meters_per_second)
→ None

Bases: *wpilib.trajectory.constraint.TrajectoryConstraint*

maxVelocity(*pose: wpilib.geometry._geometry.Pose2d*, *curvature: radians_per_meter*, *velocity: meters_per_second*) → meters_per_second

minMaxAcceleration(*pose: wpilib.geometry._geometry.Pose2d*, *curvature: radians_per_meter*, *speed: meters_per_second*) → wpilib.trajectory.constraint._constraint.TrajectoryConstraint.MinMax

1.10.5 SwerveDrive3KinematicsConstraint

class wpilib.trajectory.constraint.**SwerveDrive3KinematicsConstraint**(*kinematics: wpilib.kinematics._kinematics.Sw*
maxSpeed: meters_per_second)
→ None

Bases: *wpilib.trajectory.constraint.TrajectoryConstraint*

maxVelocity(*pose: wpilib.geometry._geometry.Pose2d*, *curvature: radians_per_meter*, *velocity: meters_per_second*) → meters_per_second

minMaxAcceleration(*pose: wpilib.geometry._geometry.Pose2d*, *curvature: radians_per_meter*, *speed: meters_per_second*) → wpilib.trajectory.constraint._constraint.TrajectoryConstraint.MinMax

1.10.6 SwerveDrive4KinematicsConstraint

class wpilib.trajectory.constraint.**SwerveDrive4KinematicsConstraint**(*kinematics: wpilib.kinematics._kinematics.Sw*
maxSpeed: meters_per_second)
→ None

Bases: *wpilib.trajectory.constraint.TrajectoryConstraint*

maxVelocity(*pose: wpilib.geometry._geometry.Pose2d*, *curvature: radians_per_meter*, *velocity: meters_per_second*) → meters_per_second

minMaxAcceleration(*pose: wpilib.geometry._geometry.Pose2d*, *curvature: radians_per_meter*, *speed: meters_per_second*) → wpilib.trajectory.constraint._constraint.TrajectoryConstraint.MinMax

1.10.7 SwerveDrive6KinematicsConstraint

class wpilib.trajectory.constraint.**SwerveDrive6KinematicsConstraint**(*kinematics: wpilib.kinematics._kinematics.Sw*
maxSpeed: meters_per_second) → None

Bases: *wpilib.trajectory.constraint.TrajectoryConstraint*

maxVelocity(*pose: wpilib.geometry._geometry.Pose2d*, *curvature: radians_per_meter*, *velocity: meters_per_second*) → meters_per_second

minMaxAcceleration(*pose: wpilib.geometry._geometry.Pose2d*, *curvature: radians_per_meter*, *speed: meters_per_second*) → wpilib.trajectory.constraint._constraint.TrajectoryConstraint.MinMax

1.10.8 TrajectoryConstraint

class wpilib.trajectory.constraint.**TrajectoryConstraint**() → None

Bases: pybind11_builtins.pybind11_object

An interface for defining user-defined velocity and acceleration constraints while generating trajectories.

class MinMax() → None

Bases: pybind11_builtins.pybind11_object

Represents a minimum and maximum acceleration.

maxAcceleration

minAcceleration

maxVelocity(*pose: wpilib.geometry._geometry.Pose2d*, *curvature: radians_per_meter*, *velocity: meters_per_second*) → meters_per_second

Returns the max velocity given the current pose and curvature.

Parameters

- **pose** – The pose at the current point in the trajectory.
- **curvature** – The curvature at the current point in the trajectory.
- **velocity** – The velocity at the current point in the trajectory before constraints are applied.

Returns The absolute maximum velocity.

minMaxAcceleration(*pose: wpilib.geometry._geometry.Pose2d*, *curvature: radians_per_meter*, *speed: meters_per_second*) → wpilib.trajectory.constraint._constraint.TrajectoryConstraint.MinMax

Returns the minimum and maximum allowable acceleration for the trajectory given pose, curvature, and speed.

Parameters

- **pose** – The pose at the current point in the trajectory.
- **curvature** – The curvature at the current point in the trajectory.
- **speed** – The speed at the current point in the trajectory.

Returns The min and max acceleration bounds.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

A

acceleration (*wpilib.trajectory.Trajectory.State attribute*), 179
Accelerometer (*class in wpilib.interfaces*), 151
Accelerometer.Range (*class in wpilib.interfaces*), 151
add() (*wpilib.DigitalGlitchFilter method*), 39
add() (*wpilib.SendableRegistry method*), 95
addBooleanArrayProperty() (*wpilib.SendableBuilder method*), 91
addBooleanArrayProperty() (*wpilib.SendableBuilderImpl method*), 93
addBooleanProperty() (*wpilib.SendableBuilder method*), 91
addBooleanProperty() (*wpilib.SendableBuilderImpl method*), 93
addChild() (*wpilib.SendableRegistry method*), 96
addConstraint() (*wpilib.trajectory.TrajectoryConfig method*), 180
addDoubleArrayProperty() (*wpilib.SendableBuilder method*), 91
addDoubleArrayProperty() (*wpilib.SendableBuilderImpl method*), 93
addDoubleProperty() (*wpilib.SendableBuilder method*), 91
addDoubleProperty() (*wpilib.SendableBuilderImpl method*), 93
addEpoch() (*wpilib.Watchdog method*), 120
addLW() (*wpilib.SendableRegistry method*), 96
addOption() (*wpilib.SendableChooser method*), 95
addRawProperty() (*wpilib.SendableBuilder method*), 91
addRawProperty() (*wpilib.SendableBuilderImpl method*), 93
AddressableLED (*class in wpilib*), 11
AddressableLED.LEDData (*class in wpilib*), 11
addressOnly() (*wpilib.I2C method*), 58
addSmallBooleanArrayProperty() (*wpilib.SendableBuilder method*), 91
addSmallBooleanArrayProperty() (*wpilib.SendableBuilderImpl method*), 93
addSmallDoubleArrayProperty() (*wpilib.SendableBuilder method*), 92
addSmallDoubleArrayProperty() (*wpilib.SendableBuilderImpl method*), 94
addSmallRawProperty() (*wpilib.SendableBuilder method*), 92
addSmallRawProperty() (*wpilib.SendableBuilderImpl method*), 94
addSmallStringArrayProperty() (*wpilib.SendableBuilder method*), 92
addSmallStringArrayProperty() (*wpilib.SendableBuilderImpl method*), 94
addSmallStringProperty() (*wpilib.SendableBuilder method*), 92
addSmallStringProperty() (*wpilib.SendableBuilderImpl method*), 94
addStringArrayProperty() (*wpilib.SendableBuilder method*), 92
addStringArrayProperty() (*wpilib.SendableBuilderImpl method*), 94
addStringProperty() (*wpilib.SendableBuilder method*), 92
addStringProperty() (*wpilib.SendableBuilderImpl method*), 94
addValueProperty() (*wpilib.SendableBuilder method*), 93
addValueProperty() (*wpilib.SendableBuilderImpl method*), 94
advanceIfElapsed() (*wpilib.Timer method*), 116
ADXL345_I2C (*class in wpilib*), 7
ADXL345_I2C.AllAxes (*class in wpilib*), 7
ADXL345_I2C.Axes (*class in wpilib*), 7
ADXL345_SPI (*class in wpilib*), 8
ADXL345_SPI.AllAxes (*class in wpilib*), 8
ADXL345_SPI.Axes (*class in wpilib*), 9
ADXL362 (*class in wpilib*), 9
ADXL362.AllAxes (*class in wpilib*), 10
ADXL362.Axes (*class in wpilib*), 10

ADXRS450_Gyro (*class in wpilib*), 10
AnalogAccelerometer (*class in wpilib*), 13
AnalogEncoder (*class in wpilib*), 14
AnalogGyro (*class in wpilib*), 14
AnalogInput (*class in wpilib*), 17
AnalogOutput (*class in wpilib*), 20
AnalogPotentiometer (*class in wpilib*), 20
AnalogTrigger (*class in wpilib*), 21
AnalogTriggerOutput (*class in wpilib*), 23
AnalogTriggerType (*class in wpilib*), 24
angle (*wpilib.kinematics.SwerveModuleState attribute*), 174
arcadeDrive() (*wpilib.drive.DifferentialDrive method*), 140
ArmFeedforward (*class in wpilib.controller*), 125
atGoal() (*wpilib.controller.ProfiledPIDController method*), 131
atReference() (*wpilib.controller.RamseteController method*), 135
atSetpoint() (*wpilib.controller.PIDController method*), 128
atSetpoint() (*wpilib.controller.ProfiledPIDController method*), 131
autonomousInit() (*wpilib.IterativeRobotBase method*), 62
autonomousPeriodic() (*wpilib.IterativeRobotBase method*), 62

B

blue (*wpilib.Color attribute*), 28
blue (*wpilib.Color8Bit attribute*), 32
BuiltInAccelerometer (*class in wpilib*), 24
busOffCount (*wpilib.CANStatus attribute*), 27

C

calculate() (*wpilib.controller.ArmFeedforward method*), 125
calculate() (*wpilib.controller.ElevatorFeedforward method*), 127
calculate() (*wpilib.controller.PIDController method*), 128
calculate() (*wpilib.controller.ProfiledPIDController method*), 131
calculate() (*wpilib.controller.RamseteController method*), 135
calculate() (*wpilib.controller.SimpleMotorFeedforward method*), 136
calculate() (*wpilib.controller.SimpleMotorFeedforwardMeters method*), 137
calculate() (*wpilib.LinearFilter method*), 67
calculate() (*wpilib.MedianFilter method*), 69
calculate() (*wpilib.SlewRateLimiter method*), 105
calculate() (*wpilib.trajectory.TrapezoidProfile method*), 186
calibrate() (*wpilib.ADXRS450_Gyro method*), 11
calibrate() (*wpilib.AnalogGyro method*), 15
calibrate() (*wpilib.interfaces.Gyro method*), 156
CameraServer (*class in wpilib*), 27
CAN (*class in wpilib*), 25
cancelInterrupts() (*wpilib.InterruptableSensorBase method*), 60
CANData (*class in wpilib*), 26
CANStatus (*class in wpilib*), 27
CentripetalAccelerationConstraint (*class in wpilib.trajectory.constraint*), 186
ChassisSpeeds (*class in wpilib.kinematics*), 159
check() (*wpilib.MotorSafety method*), 69
checkAnalogInputChannel() (*wpilib.SensorUtil static method*), 99
checkAnalogOutputChannel() (*wpilib.SensorUtil static method*), 99
checkDigitalChannel() (*wpilib.SensorUtil static method*), 99
checkMotors() (*wpilib.MotorSafety static method*), 69
checkPDPChannel() (*wpilib.SensorUtil static method*), 99
checkPDPModule() (*wpilib.SensorUtil static method*), 99
checkPWMChannel() (*wpilib.SensorUtil static method*), 99
checkRelayChannel() (*wpilib.SensorUtil static method*), 100
checkSolenoidChannel() (*wpilib.SensorUtil static method*), 100
checkSolenoidModule() (*wpilib.SensorUtil static method*), 100
clear() (*wpilib.Error method*), 55
clearAllPCMStickyFaults() (*wpilib.Compressor method*), 33
clearAllPCMStickyFaults() (*wpilib.SolenoidBase method*), 113
clearAllPCMStickyFaultsByModule() (*wpilib.SolenoidBase static method*), 113
clearDownSource() (*wpilib.Counter method*), 35
clearError() (*wpilib.ErrorBase method*), 56
clearFlags() (*wpilib.SmartDashboard static method*), 106
clearGlobalErrors() (*wpilib.ErrorBase method*), 56
clearPersistent() (*wpilib.SmartDashboard static method*), 106
clearProperties() (*wpilib.SendableBuilderImpl method*), 94
clearStickyFaults() (*wpilib.PowerDistributionPanel method*), 77

clearUpSource() (*wpilib.Counter method*), 35
cloneError() (*wpilib.ErrorBase method*), 56
Color (*class in wpilib*), 27
Color8Bit (*class in wpilib*), 32
Compressor (*class in wpilib*), 32
configureAutoStall() (*wpilib.SPI method*), 87
contains() (*wpilib.SendableRegistry method*), 97
containsKey() (*wpilib.Preferences method*), 78
containsKey() (*wpilib.SmartDashboard static method*), 106
cos() (*wpilib.geometry.Rotation2d method*), 148
Counter (*class in wpilib*), 34
Counter.Mode (*class in wpilib*), 35
CounterBase (*class in wpilib.interfaces*), 152
CounterBase.EncodingType (*class in wpilib.interfaces*), 152
createOutput() (*wpilib.AnalogTrigger method*), 21
cubicControlVectorsFromWaypoints() (*wpilib.spline.SplineHelper static method*), 177
CubicHermiteSpline (*class in wpilib.spline*), 175
cubicSplinesFromControlVectors() (*wpilib.spline.SplineHelper static method*), 177
curvature (*wpilib.trajectory.Trajectory.State attribute*), 179
curvatureDrive() (*wpilib.drive.DifferentialDrive method*), 140

D

data (*wpilib.CANData attribute*), 26
degrees() (*wpilib.geometry.Rotation2d method*), 148
delete() (*wpilib.SmartDashboard static method*), 106
deserializeTrajectory() (*wpilib.trajectory.TrajectoryUtil static method*), 184
DifferentialDrive (*class in wpilib.drive*), 139
DifferentialDriveKinematics (*class in wpilib.kinematics*), 160
DifferentialDriveKinematicsConstraint (*class in wpilib.trajectory.constraint*), 187
DifferentialDriveOdometry (*class in wpilib.kinematics*), 161
DifferentialDriveVoltageConstraint (*class in wpilib.trajectory.constraint*), 188
DifferentialDriveWheelSpeeds (*class in wpilib.kinematics*), 162
DigitalGlitchFilter (*class in wpilib*), 39
DigitalInput (*class in wpilib*), 40
DigitalOutput (*class in wpilib*), 41
DigitalSource (*class in wpilib*), 42
disable() (*wpilib.interfaces.SpeedController method*), 158
disable() (*wpilib.NidecBrushless method*), 70
disable() (*wpilib.PWMSpeedController method*), 75
disable() (*wpilib.SpeedControllerGroup method*), 114
disable() (*wpilib.Watchdog method*), 120
disableAllTelemetry() (*wpilib.LiveWindow method*), 68
disableContinuousInput() (*wpilib.controller.PIDController method*), 129
disableContinuousInput() (*wpilib.controller.ProfiledPIDController method*), 131
disabled (*wpilib.LiveWindow attribute*), 68
disabledInit() (*wpilib.IterativeRobotBase method*), 62
disabledPeriodic() (*wpilib.IterativeRobotBase method*), 62
disableInterrupts() (*wpilib.InterruptableSensorBase method*), 60
disableLiveWindow() (*wpilib.SendableRegistry method*), 97
disablePWM() (*wpilib.DigitalOutput method*), 41
disableTelemetry() (*wpilib.LiveWindow method*), 68
disableTermination() (*wpilib.SerialPort method*), 103
distance() (*wpilib.geometry.Translation2d method*), 150
distanceFeet() (*wpilib.geometry.Translation2d method*), 150
DMC60 (*class in wpilib*), 38
dot() (*wpilib.drive.Vector2d method*), 145
DoubleSolenoid (*class in wpilib*), 43
DoubleSolenoid.Value (*class in wpilib*), 43
driveCartesian() (*wpilib.drive.KilloughDrive method*), 142
driveCartesian() (*wpilib.drive.MecanumDrive method*), 143
drivePolar() (*wpilib.drive.KilloughDrive method*), 142
drivePolar() (*wpilib.drive.MecanumDrive method*), 143
DriverStation (*class in wpilib*), 44
DriverStation.Alliance (*class in wpilib*), 44
DriverStation.MatchType (*class in wpilib*), 44
ds (*wpilib.RobotBase attribute*), 82
dtheta (*wpilib.geometry.Twist2d attribute*), 151
dtheta_degrees (*wpilib.geometry.Twist2d attribute*), 151
DutyCycle (*class in wpilib*), 49
DutyCycleEncoder (*class in wpilib*), 50
dx (*wpilib.geometry.Twist2d attribute*), 151
dx_feet (*wpilib.geometry.Twist2d attribute*), 151
dy (*wpilib.geometry.Twist2d attribute*), 151

dy_feet (*wpilib.geometry.Twist2d attribute*), 151

E

ElevatorFeedforward (*class in wpilib.controller*), 126
enable() (*wpilib.NidecBrushless method*), 70
enable() (*wpilib.Watchdog method*), 120
enableContinuousInput() (*wpilib.controller.PIDController method*), 129
enableContinuousInput() (*wpilib.controller.ProfiledPIDController method*), 132
enabled (*wpilib.LiveWindow attribute*), 68
enabled() (*wpilib.Compressor method*), 33
enableDeadbandElimination() (*wpilib.PWM method*), 72
enableInterrupts() (*wpilib.InterruptableSensorBase method*), 60
enableLiveWindow() (*wpilib.SendableRegistry method*), 97
enablePWM() (*wpilib.DigitalOutput method*), 41
enableTelemetry() (*wpilib.LiveWindow method*), 68
enableTermination() (*wpilib.SerialPort method*), 103
Encoder (*class in wpilib*), 52
Encoder.IndexingType (*class in wpilib*), 52
endCompetition() (*wpilib.IterativeRobot method*), 62
endCompetition() (*wpilib.RobotBase method*), 82
endCompetition() (*wpilib.TimedRobot method*), 115
endVelocity() (*wpilib.trajectory.TrajectoryConfig method*), 181
Error (*class in wpilib*), 55
ErrorBase (*class in wpilib*), 56
exp() (*wpilib.geometry.Pose2d method*), 146

F

feed() (*wpilib.MotorSafety method*), 69
feedWatchdog() (*wpilib.drive.RobotDriveBase method*), 145
Field2d (*class in wpilib.simulation*), 174
flush() (*wpilib.SerialPort method*), 103
forceAutoRead() (*wpilib.SPI method*), 87
freeAccumulator() (*wpilib.SPI method*), 87
freeAuto() (*wpilib.SPI method*), 87
fromDegrees() (*wpilib.geometry.Rotation2d static method*), 148
fromFeet() (*wpilib.geometry.Pose2d static method*), 147
fromFeet() (*wpilib.geometry.Transform2d static method*), 149
fromFeet() (*wpilib.geometry.Translation2d static method*), 150
fromFeet() (*wpilib.geometry.Twist2d static method*), 151
fromFeet() (*wpilib.kinematics.ChassisSpeeds static method*), 159
fromFeet() (*wpilib.kinematics.DifferentialDriveWheelSpeeds static method*), 162
fromFeet() (*wpilib.kinematics.MecanumDriveWheelSpeeds static method*), 165
fromFieldRelativeSpeeds() (*wpilib.kinematics.ChassisSpeeds static method*), 159
fromPathweaverJson() (*wpilib.trajectory.TrajectoryUtil static method*), 184
frontLeft (*wpilib.kinematics.MecanumDriveWheelSpeeds attribute*), 165
frontLeft_fps (*wpilib.kinematics.MecanumDriveWheelSpeeds attribute*), 165
frontRight (*wpilib.kinematics.MecanumDriveWheelSpeeds attribute*), 165
frontRight_fps (*wpilib.kinematics.MecanumDriveWheelSpeeds attribute*), 165

G

generateTrajectory() (*wpilib.trajectory.TrajectoryGenerator static method*), 182
GenericHID (*class in wpilib.interfaces*), 153
GenericHID.Hand (*class in wpilib.interfaces*), 154
GenericHID.HIDType (*class in wpilib.interfaces*), 153
GenericHID.RumbleType (*class in wpilib.interfaces*), 154
get() (*wpilib.AnalogEncoder method*), 14
get() (*wpilib.AnalogPotentiometer method*), 21
get() (*wpilib.AnalogTriggerOutput method*), 23
get() (*wpilib.Counter method*), 36
get() (*wpilib.DigitalInput method*), 40
get() (*wpilib.DigitalOutput method*), 41
get() (*wpilib.DoubleSolenoid method*), 43
get() (*wpilib.DutyCycleEncoder method*), 51
get() (*wpilib.Encoder method*), 53
get() (*wpilib.interfaces.CounterBase method*), 153
get() (*wpilib.interfaces.Potentiometer method*), 157
get() (*wpilib.interfaces.SpeedController method*), 158
get() (*wpilib.NidecBrushless method*), 70
get() (*wpilib.PWMSpeedController method*), 75
get() (*wpilib.Relay method*), 82
get() (*wpilib.Servo method*), 104
get() (*wpilib.Solenoid method*), 112

get() (*wpilib.SpeedControllerGroup method*), 114
get() (*wpilib.Timer method*), 116
getAButton() (*wpilib.XboxController method*), 122
getAButtonPressed() (*wpilib.XboxController method*), 122
getAButtonReleased() (*wpilib.XboxController method*), 122
getAcceleration() (*wpilib.ADXL345_I2C method*), 8
getAcceleration() (*wpilib.ADXL345_SPI method*), 9
getAcceleration() (*wpilib.ADXL362 method*), 10
getAcceleration() (*wpilib.AnalogAccelerometer method*), 13
getAccelerations() (*wpilib.ADXL345_I2C method*), 8
getAccelerations() (*wpilib.ADXL345_SPI method*), 9
getAccelerations() (*wpilib.ADXL362 method*), 10
getAccumulatorAverage() (*wpilib.SPI method*), 87
getAccumulatorCount() (*wpilib.AnalogInput method*), 17
getAccumulatorCount() (*wpilib.SPI method*), 87
getAccumulatorIntegratedAverage() (*wpilib.SPI method*), 87
getAccumulatorIntegratedValue() (*wpilib.SPI method*), 87
getAccumulatorLastValue() (*wpilib.SPI method*), 87
getAccumulatorOutput() (*wpilib.AnalogInput method*), 17
getAccumulatorOutput() (*wpilib.SPI method*), 87
getAccumulatorValue() (*wpilib.AnalogInput method*), 17
getAccumulatorValue() (*wpilib.SPI method*), 88
getAll() (*wpilib.SolenoidBase method*), 113
getAllByModule() (*wpilib.SolenoidBase static method*), 113
getAlliance() (*wpilib.DriverStation method*), 45
getAnalogTriggerTypeForRouting() (*wpilib.AnalogTriggerOutput method*), 23
getAnalogTriggerTypeForRouting() (*wpilib.DigitalInput method*), 40
getAnalogTriggerTypeForRouting() (*wpilib.DigitalOutput method*), 41
getAnalogTriggerTypeForRouting() (*wpilib.DigitalSource method*), 42
getAnalogTriggerTypeForRouting() (*wpilib.InterruptableSensorBase method*), 60
getAngle() (*wpilib.ADXRS450_Gyro method*), 11
getAngle() (*wpilib.AnalogGyro method*), 15
getAngle() (*wpilib.interfaces.Gyro method*), 156
getAngle() (*wpilib.Servo method*), 104
getAutoDroppedCount() (*wpilib.SPI method*), 88
getAverageBits() (*wpilib.AnalogInput method*), 17
getAverageValue() (*wpilib.AnalogInput method*), 17
getAverageVoltage() (*wpilib.AnalogInput method*), 17
getAxisCount() (*wpilib.interfaces.GenericHID method*), 154
getAxisType() (*wpilib.interfaces.GenericHID method*), 154
getBackButton() (*wpilib.XboxController method*), 122
getBackButtonPressed() (*wpilib.XboxController method*), 122
getBackButtonReleased() (*wpilib.XboxController method*), 122
getBatteryVoltage() (*wpilib.DriverStation method*), 45
getBButton() (*wpilib.XboxController method*), 122
getBButtonPressed() (*wpilib.XboxController method*), 122
getBButtonReleased() (*wpilib.XboxController method*), 122
getBoolean() (*wpilib.Preferences method*), 78
getBoolean() (*wpilib.SmartDashboard static method*), 106
getBooleanArray() (*wpilib.SmartDashboard static method*), 106
getBumper() (*wpilib.XboxController method*), 123
getBumperPressed() (*wpilib.XboxController method*), 123
getBumperReleased() (*wpilib.XboxController method*), 123
getButtonCount() (*wpilib.interfaces.GenericHID method*), 155
getBytesReceived() (*wpilib.SerialPort method*), 103
getCANStatus() (*wpilib.RobotController static method*), 84
getCenter() (*wpilib.AnalogGyro method*), 16
getChannel() (*wpilib.AnalogInput method*), 18
getChannel() (*wpilib.AnalogOutput method*), 20
getChannel() (*wpilib.AnalogTriggerOutput method*), 23
getChannel() (*wpilib.DigitalInput method*), 40
getChannel() (*wpilib.DigitalOutput method*), 41
getChannel() (*wpilib.DigitalSource method*), 42
getChannel() (*wpilib.NidecBrushless method*), 71
getChannel() (*wpilib.PWM method*), 72
getChannel() (*wpilib.Relay method*), 82
getClosedLoopControl() (*wpilib.Compressor method*), 33
getCode() (*wpilib.Error method*), 55

getCompressorCurrent() (*wpilib.Compressor method*), 33
getCompressorCurrentTooHighFault() (*wpilib.Compressor method*), 33
getCompressorCurrentTooHighStickyFault() (*wpilib.Compressor method*), 33
getCompressorNotConnectedFault() (*wpilib.Compressor method*), 33
getCompressorNotConnectedStickyFault() (*wpilib.Compressor method*), 33
getCompressorShortedFault() (*wpilib.Compressor method*), 33
getCompressorShortedStickyFault() (*wpilib.Compressor method*), 34
getControlState() (*wpilib.DriverStation method*), 45
getControlState() (*wpilib.RobotBase method*), 82
getCurrent() (*wpilib.PowerDistributionPanel method*), 77
getCurrent3V3() (*wpilib.RobotController static method*), 84
getCurrent5V() (*wpilib.RobotController static method*), 84
getCurrent6V() (*wpilib.RobotController static method*), 84
getCurrentThreadPriority() (*in module wpilib*), 6
getD() (*wpilib.controller.PIDController method*), 129
getD() (*wpilib.controller.ProfiledPIDController method*), 132
getData() (*wpilib.SmartDashboard static method*), 106
getDefaultSolenoidModule() (*wpilib.SensorUtil static method*), 100
getDescription() (*wpilib.drive.DifferentialDrive method*), 140
getDescription() (*wpilib.drive.KilloughDrive method*), 142
getDescription() (*wpilib.drive.MecanumDrive method*), 144
getDescription() (*wpilib.drive.RobotDriveBase method*), 145
getDescription() (*wpilib.MotorSafety method*), 70
getDescription() (*wpilib.NidecBrushless method*), 71
getDescription() (*wpilib.PWM method*), 72
getDescription() (*wpilib.Relay method*), 82
getDirection() (*wpilib.Counter method*), 36
getDirection() (*wpilib.Encoder method*), 53
getDirection() (*wpilib.interfaces.CounterBase method*), 153
getDirectionDegrees() (*wpilib.Joystick method*), 64
getDirectionRadians() (*wpilib.Joystick method*), 65
getDistance() (*wpilib.AnalogEncoder method*), 14
getDistance() (*wpilib.DutyCycleEncoder method*), 51
getDistance() (*wpilib.Encoder method*), 53
getDistancePerPulse() (*wpilib.Encoder method*), 53
getDistancePerRotation() (*wpilib.AnalogEncoder method*), 14
getDistancePerRotation() (*wpilib.DutyCycleEncoder method*), 51
getDistanceUnits() (*wpilib.Ultrasonic method*), 118
getDouble() (*wpilib.Preferences method*), 78
getEnabled3V3() (*wpilib.RobotController static method*), 84
getEnabled5V() (*wpilib.RobotController static method*), 84
getEnabled6V() (*wpilib.RobotController static method*), 84
getEncodingScale() (*wpilib.Encoder method*), 53
getEntry() (*wpilib.SendableBuilder method*), 93
getEntry() (*wpilib.SendableBuilderImpl method*), 94
getEntry() (*wpilib.SmartDashboard static method*), 107
getError() (*wpilib.ErrorBase method*), 56
getEventName() (*wpilib.DriverStation method*), 45
getExpiration() (*wpilib.MotorSafety method*), 70
getFaultCount3V3() (*wpilib.RobotController static method*), 84
getFaultCount5V() (*wpilib.RobotController static method*), 85
getFaultCount6V() (*wpilib.RobotController static method*), 85
getFilename() (*wpilib.Error method*), 55
getFlags() (*wpilib.SmartDashboard static method*), 107
getFloat() (*wpilib.Preferences method*), 78
getFPGAIndex() (*wpilib.Counter method*), 36
getFPGAIndex() (*wpilib.DutyCycle method*), 50
getFPGAIndex() (*wpilib.Encoder method*), 53
getFPGARevision() (*wpilib.RobotController static method*), 84
getFPGATime() (*wpilib.RobotController static method*), 84
getFPGATimestamp() (*wpilib.Timer static method*), 116
getFPGAVersion() (*wpilib.RobotController static method*), 84
getFrequency() (*wpilib.DutyCycle method*), 50
getFrequency() (*wpilib.DutyCycleEncoder method*), 51
getFunction() (*wpilib.Error method*), 55
getGameSpecificMessage()

(*wpilib.DriverStation method*), 45
getGlobalError() (*wpilib.ErrorBase static method*), 56
getGlobalErrors() (*wpilib.ErrorBase static method*), 56
getGoal() (*wpilib.controller.ProfiledPIDController method*), 132
getI() (*wpilib.controller.PIDController method*), 129
getI() (*wpilib.controller.ProfiledPIDController method*), 132
getIndex() (*wpilib.AnalogTrigger method*), 22
getInputCurrent() (*wpilib.RobotController static method*), 85
getInputVoltage() (*wpilib.RobotController static method*), 85
getInstance() (*wpilib.DriverStation static method*), 45
getInstance() (*wpilib.LiveWindow static method*), 68
getInstance() (*wpilib.Preferences static method*), 79
getInstance() (*wpilib.SendableRegistry static method*), 97
getInt() (*wpilib.Preferences method*), 79
getInverted() (*wpilib.interfaces.SpeedController method*), 158
getInverted() (*wpilib.NidecBrushless method*), 71
getInverted() (*wpilib.PWMSpeedController method*), 75
getInverted() (*wpilib.SpeedControllerGroup method*), 114
getInWindow() (*wpilib.AnalogTrigger method*), 22
getJoystickAxisType() (*wpilib.DriverStation method*), 45
getJoystickIsXbox() (*wpilib.DriverStation method*), 45
getJoystickName() (*wpilib.DriverStation method*), 45
getJoystickType() (*wpilib.DriverStation method*), 45
getKeys() (*wpilib.Preferences method*), 79
getKeys() (*wpilib.SmartDashboard static method*), 107
getLineNumber() (*wpilib.Error method*), 55
getLocation() (*wpilib.DriverStation method*), 46
getLong() (*wpilib.Preferences method*), 79
getLSBWeight() (*wpilib.AnalogInput method*), 18
getMagnitude() (*wpilib.Joystick method*), 65
getMatchNumber() (*wpilib.DriverStation method*), 46
getMatchTime() (*wpilib.DriverStation method*), 46
getMatchTime() (*wpilib.Timer static method*), 116
getMatchType() (*wpilib.DriverStation method*), 46
getMaxAngle() (*wpilib.Servo method*), 104
getMessage() (*wpilib.Error method*), 55
getMinAngle() (*wpilib.Servo method*), 104
getName() (*wpilib.interfaces.GenericHID method*), 155
getName() (*wpilib.SendableRegistry method*), 97
getNumber() (*wpilib.SmartDashboard static method*), 107
getNumberArray() (*wpilib.SmartDashboard static method*), 107
getOffset() (*wpilib.AnalogGyro method*), 16
getOffset() (*wpilib.AnalogInput method*), 18
getOriginatingObject() (*wpilib.Error method*), 55
getOutput() (*wpilib.DutyCycle method*), 50
getOutputRaw() (*wpilib.DutyCycle method*), 50
getOutputScaleFactor() (*wpilib.DutyCycle method*), 50
getOversampleBits() (*wpilib.AnalogInput method*), 18
getP() (*wpilib.controller.PIDController method*), 129
getP() (*wpilib.controller.ProfiledPIDController method*), 132
getPCMSolenoidBlackList() (*wpilib.SolenoidBase method*), 113
getPCMSolenoidBlackListByModule() (*wpilib.SolenoidBase static method*), 113
getPCMSolenoidVoltageFault() (*wpilib.SolenoidBase method*), 114
getPCMSolenoidVoltageFaultByModule() (*wpilib.SolenoidBase static method*), 114
getPCMSolenoidVoltageStickyFault() (*wpilib.SolenoidBase method*), 114
getPCMSolenoidVoltageStickyFaultByModule() (*wpilib.SolenoidBase static method*), 114
getPeriod() (*wpilib.controller.PIDController method*), 129
getPeriod() (*wpilib.controller.ProfiledPIDController method*), 132
getPeriod() (*wpilib.Counter method*), 36
getPeriod() (*wpilib.Encoder method*), 53
getPeriod() (*wpilib.interfaces.CounterBase method*), 153
getPeriod() (*wpilib.TimedRobot method*), 115
getPeriodCycles() (*wpilib.DigitalGlitchFilter method*), 39
getPeriodNanoSeconds() (*wpilib.DigitalGlitchFilter method*), 39
getPIDSourceType() (*wpilib.interfaces.PIDSource method*), 157
getPoint() (*wpilib.spline.Spline3 method*), 176
getPoint() (*wpilib.spline.Spline5 method*), 177
getPort() (*wpilib.interfaces.GenericHID method*), 155
getPortHandleForRouting()

(*wpilib.AnalogTriggerOutput method*), 23
getPortHandleForRouting() (*wpilib.DigitalInput method*), 40
getPortHandleForRouting() (*wpilib.DigitalOutput method*), 41
getPortHandleForRouting() (*wpilib.DigitalSource method*), 42
getPortHandleForRouting() (*wpilib.InterruptableSensorBase method*), 60
getPose() (*wpilib.kinematics.DifferentialDriveOdometry method*), 161
getPose() (*wpilib.kinematics.MecanumDriveOdometry method*), 164
getPose() (*wpilib.kinematics.SwerveDrive3Odometry method*), 167
getPose() (*wpilib.kinematics.SwerveDrive4Odometry method*), 170
getPose() (*wpilib.kinematics.SwerveDrive6Odometry method*), 173
getPosition() (*wpilib.PWM method*), 72
getPositionError() (*wpilib.controller.PIDController method*), 129
getPositionError() (*wpilib.controller.ProfiledPIDController method*), 132
getPositionOffset() (*wpilib.AnalogEncoder method*), 14
getPOV() (*wpilib.interfaces.GenericHID method*), 155
getPOVCount() (*wpilib.interfaces.GenericHID method*), 155
getPressureSwitchValue() (*wpilib.Compressor method*), 34
getRangeInches() (*wpilib.Ultrasonic method*), 118
getRangeMM() (*wpilib.Ultrasonic method*), 118
getRate() (*wpilib.ADXRS450_Gyro method*), 11
getRate() (*wpilib.AnalogGyro method*), 16
getRate() (*wpilib.Encoder method*), 53
getRate() (*wpilib.interfaces.Gyro method*), 156
getRaw() (*wpilib.Encoder method*), 53
getRaw() (*wpilib.PWM method*), 73
getRaw() (*wpilib.SmartDashboard static method*), 107
getRawAxis() (*wpilib.interfaces.GenericHID method*), 155
getRawBounds() (*wpilib.PWM method*), 73
getRawButton() (*wpilib.interfaces.GenericHID method*), 155
getRawButtonPressed() (*wpilib.interfaces.GenericHID method*), 155
getRawButtonReleased() (*wpilib.interfaces.GenericHID method*), 155
getReplayNumber() (*wpilib.DriverStation method*), 46
getRobotPose() (*wpilib.simulation.Field2d method*), 174
getSampleRate() (*wpilib.AnalogInput static method*), 18
getSamplesToAverage() (*wpilib.Counter method*), 36
getSamplesToAverage() (*wpilib.Encoder method*), 53
getSelected() (*wpilib.SendableChooser method*), 95
getSendable() (*wpilib.SendableRegistry method*), 97
getSetpoint() (*wpilib.controller.PIDController method*), 129
getSetpoint() (*wpilib.controller.ProfiledPIDController method*), 132
getSourceChannel() (*wpilib.DutyCycle method*), 50
getSpeed() (*wpilib.PWM method*), 73
getStartButton() (*wpilib.XboxController method*), 123
getStartButtonPressed() (*wpilib.XboxController method*), 123
getStartButtonReleased() (*wpilib.XboxController method*), 123
getStickAxis() (*wpilib.DriverStation method*), 46
getStickAxisCount() (*wpilib.DriverStation method*), 46
getStickButton() (*wpilib.DriverStation method*), 46
getStickButton() (*wpilib.XboxController method*), 123
getStickButtonCount() (*wpilib.DriverStation method*), 47
getStickButtonPressed() (*wpilib.DriverStation method*), 47
getStickButtonPressed() (*wpilib.XboxController method*), 123
getStickButtonReleased() (*wpilib.DriverStation method*), 47
getStickButtonReleased() (*wpilib.XboxController method*), 123
getStickButtons() (*wpilib.DriverStation method*), 47
getStickPOV() (*wpilib.DriverStation method*), 47
getStickPOVCount() (*wpilib.DriverStation method*), 47
getStopped() (*wpilib.Counter method*), 36
getStopped() (*wpilib.Encoder method*), 54
getStopped() (*wpilib.interfaces.CounterBase method*), 153
getString() (*wpilib.Preferences method*), 79
getString() (*wpilib.SmartDashboard static method*),

108
getStringArray() (*wpilib.SmartDashboard static method*), 108
getSubsystem() (*wpilib.SendableRegistry method*), 97
getTable() (*wpilib.SendableBuilderImpl method*), 94
getTemperature() (*wpilib.PowerDistributionPanel method*), 77
getThreadId() (*wpilib.RobotBase static method*), 82
getThreadPriority() (*in module wpilib*), 6
getThrottle() (*wpilib.Joystick method*), 65
getThrottleChannel() (*wpilib.Joystick method*), 65
getTime() (*in module wpilib*), 6
getTime() (*wpilib.Watchdog method*), 120
getTimeout() (*wpilib.Watchdog method*), 120
getTimestamp() (*wpilib.Error method*), 55
getTop() (*wpilib.Joystick method*), 65
getTopPressed() (*wpilib.Joystick method*), 65
getTopReleased() (*wpilib.Joystick method*), 65
getTotalCurrent() (*wpilib.PowerDistributionPanel method*), 77
getTotalEnergy() (*wpilib.PowerDistributionPanel method*), 77
getTotalPower() (*wpilib.PowerDistributionPanel method*), 77
getTrigger() (*wpilib.Joystick method*), 65
getTriggerAxis() (*wpilib.XboxController method*), 123
getTriggerPressed() (*wpilib.Joystick method*), 65
getTriggerReleased() (*wpilib.Joystick method*), 65
getTriggerState() (*wpilib.AnalogTrigger method*), 22
getTwist() (*wpilib.Joystick method*), 65
getTwistChannel() (*wpilib.Joystick method*), 65
getType() (*wpilib.interfaces.GenericHID method*), 155
getUniqueId() (*wpilib.SendableRegistry method*), 97
getUserButton() (*wpilib.RobotController static method*), 85
getValue() (*wpilib.AnalogInput method*), 18
getValue() (*wpilib.SmartDashboard static method*), 108
getVelocityError() (*wpilib.controller.PIDController method*), 129
getVelocityError() (*wpilib.controller.ProfiledPIDController method*), 132
getVoltage() (*wpilib.AnalogInput method*), 18
getVoltage() (*wpilib.AnalogOutput method*), 20
getVoltage() (*wpilib.PowerDistributionPanel method*), 78
getVoltage3V3() (*wpilib.RobotController static method*), 85
getVoltage5V() (*wpilib.RobotController static method*), 85
getVoltage6V() (*wpilib.RobotController static method*), 85
getX() (*wpilib.ADXL345_I2C method*), 8
getX() (*wpilib.ADXL345_SPI method*), 9
getX() (*wpilib.ADXL362 method*), 10
getX() (*wpilib.BuiltInAccelerometer method*), 24
getX() (*wpilib.interfaces.Accelerometer method*), 152
getX() (*wpilib.interfaces.GenericHID method*), 155
getX() (*wpilib.Joystick method*), 65
getX() (*wpilib.XboxController method*), 123
getXButton() (*wpilib.XboxController method*), 124
getXButtonPressed() (*wpilib.XboxController method*), 124
getXButtonReleased() (*wpilib.XboxController method*), 124
getXChannel() (*wpilib.Joystick method*), 66
getY() (*wpilib.ADXL345_I2C method*), 8
getY() (*wpilib.ADXL345_SPI method*), 9
getY() (*wpilib.ADXL362 method*), 10
getY() (*wpilib.BuiltInAccelerometer method*), 24
getY() (*wpilib.interfaces.Accelerometer method*), 152
getY() (*wpilib.interfaces.GenericHID method*), 155
getY() (*wpilib.Joystick method*), 66
getY() (*wpilib.XboxController method*), 124
getYButton() (*wpilib.XboxController method*), 124
getYButtonPressed() (*wpilib.XboxController method*), 124
getYButtonReleased() (*wpilib.XboxController method*), 124
getYChannel() (*wpilib.Joystick method*), 66
getZ() (*wpilib.ADXL345_I2C method*), 8
getZ() (*wpilib.ADXL345_SPI method*), 9
getZ() (*wpilib.ADXL362 method*), 10
getZ() (*wpilib.BuiltInAccelerometer method*), 24
getZ() (*wpilib.interfaces.Accelerometer method*), 152
getZ() (*wpilib.Joystick method*), 66
getZChannel() (*wpilib.Joystick method*), 66
green (*wpilib.Color attribute*), 28
green (*wpilib.Color8Bit attribute*), 32
Gyro (*class in wpilib.interfaces*), 156
GyroBase (*class in wpilib*), 58

H

hasElapsed() (*wpilib.Timer method*), 116
hasPeriodPassed() (*wpilib.Timer method*), 116
hasTable() (*wpilib.SendableBuilderImpl method*), 94
highPass() (*wpilib.LinearFilter static method*), 68

I

I2C (*class in wpilib*), 58
I2C.Port (*class in wpilib*), 58
inAutonomous() (*wpilib.DriverStation method*), 47
inDisabled() (*wpilib.DriverStation method*), 47
init() (*wpilib.SmartDashboard static method*), 108
initAccumulator() (*wpilib.AnalogInput method*), 18
initAccumulator() (*wpilib.SPI method*), 88
initAuto() (*wpilib.SPI method*), 88
initGyro() (*wpilib.AnalogGyro method*), 16
initialPose() (*wpilib.trajectory.Trajectory method*), 179
initSendable() (*wpilib.ADXL345_I2C method*), 8
initSendable() (*wpilib.ADXL345_SPI method*), 9
initSendable() (*wpilib.ADXL362 method*), 10
initSendable() (*wpilib.AnalogAccelerometer method*), 13
initSendable() (*wpilib.AnalogEncoder method*), 14
initSendable() (*wpilib.AnalogInput method*), 18
initSendable() (*wpilib.AnalogOutput method*), 20
initSendable() (*wpilib.AnalogPotentiometer method*), 21
initSendable() (*wpilib.AnalogTrigger method*), 22
initSendable() (*wpilib.AnalogTriggerOutput method*), 24
initSendable() (*wpilib.BuiltInAccelerometer method*), 24
initSendable() (*wpilib.Compressor method*), 34
initSendable() (*wpilib.controller.PIDController method*), 129
initSendable() (*wpilib.controller.ProfiledPIDController method*), 132
initSendable() (*wpilib.Counter method*), 36
initSendable() (*wpilib.DigitalGlitchFilter method*), 39
initSendable() (*wpilib.DigitalInput method*), 41
initSendable() (*wpilib.DigitalOutput method*), 41
initSendable() (*wpilib.DoubleSolenoid method*), 43
initSendable() (*wpilib.drive.DifferentialDrive method*), 140
initSendable() (*wpilib.drive.KilloughDrive method*), 142
initSendable() (*wpilib.drive.MecanumDrive method*), 144
initSendable() (*wpilib.DutyCycleEncoder method*), 51
initSendable() (*wpilib.Encoder method*), 54
initSendable() (*wpilib.GyroBase method*), 58
initSendable() (*wpilib.NidecBrushless method*), 71
initSendable() (*wpilib.PowerDistributionPanel method*), 78
initSendable() (*wpilib.Relay method*), 82
initSendable() (*wpilib.Sendable method*), 90
initSendable() (*wpilib.SendableChooser method*), 95
initSendable() (*wpilib.Servo method*), 105
initSendable() (*wpilib.Solenoid method*), 112
initSendable() (*wpilib.SpeedControllerGroup method*), 115
initSendable() (*wpilib.Ultrasonic method*), 118
inOperatorControl() (*wpilib.DriverStation method*), 47
interpolate() (*wpilib.trajectory.Trajectory.State method*), 179
InterruptableSensorBase (*class in wpilib*), 60
InterruptableSensorBase.WaitResult (*class in wpilib*), 60
inTest() (*wpilib.DriverStation method*), 47
is_alive() (*wpilib.CameraServer class method*), 27
isAccumulatorChannel() (*wpilib.AnalogInput method*), 18
isActuator() (*wpilib.SendableBuilderImpl method*), 94
isAlive() (*wpilib.MotorSafety method*), 70
isAnalogTrigger() (*wpilib.AnalogTriggerOutput method*), 24
isAnalogTrigger() (*wpilib.DigitalInput method*), 41
isAnalogTrigger() (*wpilib.DigitalOutput method*), 41
isAnalogTrigger() (*wpilib.DigitalSource method*), 42
isAutonomous() (*wpilib.DriverStation method*), 48
isAutonomous() (*wpilib.RobotBase method*), 83
isAutonomous() (*wpilib.RobotState static method*), 85
isAutonomousEnabled() (*wpilib.DriverStation method*), 48
isAutonomousEnabled() (*wpilib.RobotBase method*), 83
isBlackListed() (*wpilib.Solenoid method*), 112
isBrownedOut() (*wpilib.RobotController static method*), 85
isConnected() (*wpilib.DutyCycleEncoder method*), 51
isDisabled() (*wpilib.DriverStation method*), 48
isDisabled() (*wpilib.RobotBase method*), 83
isDisabled() (*wpilib.RobotState static method*), 85
isDSAttached() (*wpilib.DriverStation method*), 48
isEnabled() (*wpilib.DriverStation method*), 48
isEnabled() (*wpilib.LiveWindow method*), 69
isEnabled() (*wpilib.RobotBase method*), 83
isEnabled() (*wpilib.RobotState static method*), 85
isEnabled() (*wpilib.Ultrasonic method*), 118
isEStopped() (*wpilib.DriverStation method*), 48
isEStopped() (*wpilib.RobotState static method*), 85

isExpired() (*wpilib.Watchdog method*), 120
isFinished() (*wpilib.trajectory.TrapezoidProfile method*), 186
isFMSAttached() (*wpilib.DriverStation method*), 48
isFwdSolenoidBlackListed() (*wpilib.DoubleSolenoid method*), 43
isNewControlData() (*wpilib.DriverStation method*), 48
isNewDataAvailable() (*wpilib.RobotBase method*), 83
isOperatorControl() (*wpilib.DriverStation method*), 48
isOperatorControl() (*wpilib.RobotBase method*), 83
isOperatorControl() (*wpilib.RobotState static method*), 86
isOperatorControlEnabled() (*wpilib.DriverStation method*), 48
isOperatorControlEnabled() (*wpilib.RobotBase method*), 83
isPersistent() (*wpilib.SmartDashboard static method*), 108
isPulsing() (*wpilib.DigitalOutput method*), 41
isRangeValid() (*wpilib.Ultrasonic method*), 118
isReal() (*wpilib.RobotBase static method*), 83
isReversed() (*wpilib.trajectory.TrajectoryConfig method*), 181
isRevSolenoidBlackListed() (*wpilib.DoubleSolenoid method*), 44
isRightSideInverted() (*wpilib.drive.DifferentialDrive method*), 140
isRightSideInverted() (*wpilib.drive.MecanumDrive method*), 144
isSafetyEnabled() (*wpilib.MotorSafety method*), 70
isSimulation() (*wpilib.RobotBase static method*), 83
isSysActive() (*wpilib.RobotController static method*), 85
isTest() (*wpilib.DriverStation method*), 49
isTest() (*wpilib.RobotBase method*), 83
isTest() (*wpilib.RobotState static method*), 86
IterativeRobot (*class in wpilib*), 61
IterativeRobotBase (*class in wpilib*), 62

J

Jaguar (*class in wpilib*), 63
Joystick (*class in wpilib*), 64
Joystick.AxisType (*class in wpilib*), 64
Joystick.ButtonType (*class in wpilib*), 64

K

k1X (*wpilib.interfaces.CounterBase.EncodingType attribute*), 152
k2X (*wpilib.interfaces.CounterBase.EncodingType attribute*), 152
k4X (*wpilib.interfaces.CounterBase.EncodingType attribute*), 152
kA (*wpilib.controller.ArmFeedforward attribute*), 125
kA (*wpilib.controller.ElevatorFeedforward attribute*), 127
kA (*wpilib.controller.SimpleMotorFeedforward attribute*), 136
kA (*wpilib.controller.SimpleMotorFeedforwardMeters attribute*), 137
kA (*wpilib.XboxController.Button attribute*), 122
kAccumulatorModuleNumber (*wpilib.AnalogInput attribute*), 19
kAccumulatorNumChannels (*wpilib.AnalogInput attribute*), 19
kAliceBlue (*wpilib.Color attribute*), 28
kAnalogInputs (*wpilib.SensorUtil attribute*), 100
kAntiqueWhite (*wpilib.Color attribute*), 28
kAqua (*wpilib.Color attribute*), 28
kAquamarine (*wpilib.Color attribute*), 28
kAverageBits (*wpilib.AnalogGyro attribute*), 16
kAxis_X (*wpilib.ADXL345_I2C.Axes attribute*), 8
kAxis_X (*wpilib.ADXL345_SPI.Axes attribute*), 9
kAxis_X (*wpilib.ADXL362.Axes attribute*), 10
kAxis_Y (*wpilib.ADXL345_I2C.Axes attribute*), 8
kAxis_Y (*wpilib.ADXL345_SPI.Axes attribute*), 9
kAxis_Y (*wpilib.ADXL362.Axes attribute*), 10
kAxis_Z (*wpilib.ADXL345_I2C.Axes attribute*), 8
kAxis_Z (*wpilib.ADXL345_SPI.Axes attribute*), 9
kAxis_Z (*wpilib.ADXL362.Axes attribute*), 10
kAzure (*wpilib.Color attribute*), 28
kB (*wpilib.XboxController.Button attribute*), 122
kBack (*wpilib.drive.RobotDriveBase.MotorType attribute*), 144
kBack (*wpilib.XboxController.Button attribute*), 122
kBeige (*wpilib.Color attribute*), 28
kBisque (*wpilib.Color attribute*), 28
kBlack (*wpilib.Color attribute*), 28
kBlanchedAlmond (*wpilib.Color attribute*), 28
kBlue (*wpilib.Color attribute*), 28
kBlue (*wpilib.DriverStation.Alliance attribute*), 44
kBlueViolet (*wpilib.Color attribute*), 28
kBoth (*wpilib.InterruptableSensorBase.WaitResult attribute*), 60
kBothDirections (*wpilib.Relay.Direction attribute*), 81
kBrown (*wpilib.Color attribute*), 28
kBumperLeft (*wpilib.XboxController.Button attribute*), 122
kBumperRight (*wpilib.XboxController.Button attribute*), 122
kBurlywood (*wpilib.Color attribute*), 28
kCadetBlue (*wpilib.Color attribute*), 28

kCalibrationSampleTime (*wpilib.AnalogGyro attribute*), 16
kChartreuse (*wpilib.Color attribute*), 28
kChocolate (*wpilib.Color attribute*), 28
kCoral (*wpilib.Color attribute*), 28
kCornflowerBlue (*wpilib.Color attribute*), 28
kCornsilk (*wpilib.Color attribute*), 28
kCos (*wpilib.controller.ArmFeedforward attribute*), 125
kCrimson (*wpilib.Color attribute*), 28
kCyan (*wpilib.Color attribute*), 28
kDarkBlue (*wpilib.Color attribute*), 28
kDarkCyan (*wpilib.Color attribute*), 28
kDarkGoldenrod (*wpilib.Color attribute*), 28
kDarkGray (*wpilib.Color attribute*), 28
kDarkGreen (*wpilib.Color attribute*), 28
kDarkKhaki (*wpilib.Color attribute*), 28
kDarkMagenta (*wpilib.Color attribute*), 29
kDarkOliveGreen (*wpilib.Color attribute*), 29
kDarkOrange (*wpilib.Color attribute*), 29
kDarkOrchid (*wpilib.Color attribute*), 29
kDarkRed (*wpilib.Color attribute*), 29
kDarkSalmon (*wpilib.Color attribute*), 29
kDarkSeaGreen (*wpilib.Color attribute*), 29
kDarkSlateBlue (*wpilib.Color attribute*), 29
kDarkSlateGray (*wpilib.Color attribute*), 29
kDarkTurquoise (*wpilib.Color attribute*), 29
kDarkViolet (*wpilib.Color attribute*), 29
kDeepPink (*wpilib.Color attribute*), 29
kDeepSkyBlue (*wpilib.Color attribute*), 29
kDefaultBackMotorAngle (*wpilib.drive.KilloughDrive attribute*), 142
kDefaultLeftMotorAngle (*wpilib.drive.KilloughDrive attribute*), 142
kDefaultPeriod (*wpilib.TimedRobot attribute*), 115
kDefaultQuickStopAlpha (*wpilib.drive.DifferentialDrive attribute*), 140
kDefaultQuickStopThreshold (*wpilib.drive.DifferentialDrive attribute*), 140
kDefaultRightMotorAngle (*wpilib.drive.KilloughDrive attribute*), 142
kDefaultThrottleChannel (*wpilib.Joystick attribute*), 66
kDefaultTwistChannel (*wpilib.Joystick attribute*), 66
kDefaultVoltsPerDegreePerSecond (*wpilib.AnalogGyro attribute*), 16
kDefaultXChannel (*wpilib.Joystick attribute*), 66
kDefaultYChannel (*wpilib.Joystick attribute*), 66
kDefaultZChannel (*wpilib.Joystick attribute*), 66
kDenim (*wpilib.Color attribute*), 29
kDigitalChannels (*wpilib.SensorUtil attribute*), 100
kDimGray (*wpilib.Color attribute*), 29
kDisplacement (*wpilib.interfaces.PIDSourceType attribute*), 157
kDodgerBlue (*wpilib.Color attribute*), 29
kElimination (*wpilib.DriverStation.MatchType attribute*), 44
kExternalDirection (*wpilib.Counter.Mode attribute*), 35
kFallingEdge (*wpilib.InterruptableSensorBase.WaitResult attribute*), 60
kFallingPulse (*wpilib.AnalogTriggerType attribute*), 24
kFirebrick (*wpilib.Color attribute*), 29
kFirstBlue (*wpilib.Color attribute*), 29
kFirstRed (*wpilib.Color attribute*), 29
kFloralWhite (*wpilib.Color attribute*), 29
kFlowControl_DtrDsr (*wpilib.SerialPort.FlowControl attribute*), 101
kFlowControl_None (*wpilib.SerialPort.FlowControl attribute*), 101
kFlowControl_RtsCts (*wpilib.SerialPort.FlowControl attribute*), 101
kFlowControl_XonXoff (*wpilib.SerialPort.FlowControl attribute*), 101
kFlushOnAccess (*wpilib.SerialPort.WriteBufferMode attribute*), 103
kFlushWhenFull (*wpilib.SerialPort.WriteBufferMode attribute*), 103
kForestGreen (*wpilib.Color attribute*), 29
kForward (*wpilib.DoubleSolenoid.Value attribute*), 43
kForward (*wpilib.Relay.Value attribute*), 81
kForwardOnly (*wpilib.Relay.Direction attribute*), 81
kFrontLeft (*wpilib.drive.RobotDriveBase.MotorType attribute*), 144
kFrontRight (*wpilib.drive.RobotDriveBase.MotorType attribute*), 144
kFuchsia (*wpilib.Color attribute*), 29
kG (*wpilib.controller.ElevatorFeedforward attribute*), 127
kGainsboro (*wpilib.Color attribute*), 29
kGhostWhite (*wpilib.Color attribute*), 29
kGold (*wpilib.Color attribute*), 29
kGoldenrod (*wpilib.Color attribute*), 29
kGray (*wpilib.Color attribute*), 29
kGreen (*wpilib.Color attribute*), 29
kGreenYellow (*wpilib.Color attribute*), 29
kHID1stPerson (*wpilib.interfaces.GenericHID.HIDType attribute*), 153
kHIDDriving (*wpilib.interfaces.GenericHID.HIDType attribute*), 153
kHIDFlight (*wpilib.interfaces.GenericHID.HIDType attribute*), 153

kHIDGamepad (*wpilib.interfaces.GenericHID.HIDType attribute*), 153
kHIDJoystick (*wpilib.interfaces.GenericHID.HIDType attribute*), 153
kHoneydew (*wpilib.Color attribute*), 29
kHotPink (*wpilib.Color attribute*), 29
KilloughDrive (*class in wpilib.drive*), 141
kInches (*wpilib.Ultrasonic.DistanceUnit attribute*), 118
kIndianRed (*wpilib.Color attribute*), 29
kIndigo (*wpilib.Color attribute*), 29
kInvalid (*wpilib.DriverStation.Alliance attribute*), 44
kInWindow (*wpilib.AnalogTriggerType attribute*), 24
kIvory (*wpilib.Color attribute*), 29
kJoystickPorts (*wpilib.DriverStation attribute*), 49
kKhaki (*wpilib.Color attribute*), 29
kLavender (*wpilib.Color attribute*), 29
kLavenderBlush (*wpilib.Color attribute*), 30
kLawnGreen (*wpilib.Color attribute*), 30
kLeft (*wpilib.drive.RobotDriveBase.MotorType attribute*), 144
kLeftHand (*wpilib.interfaces.GenericHID.Hand attribute*), 154
kLeftRumble (*wpilib.interfaces.GenericHID.RumbleType attribute*), 154
kLeftTrigger (*wpilib.XboxController.Axis attribute*), 121
kLeftX (*wpilib.XboxController.Axis attribute*), 121
kLeftY (*wpilib.XboxController.Axis attribute*), 121
kLemonChiffon (*wpilib.Color attribute*), 30
kLightBlue (*wpilib.Color attribute*), 30
kLightCoral (*wpilib.Color attribute*), 30
kLightCyan (*wpilib.Color attribute*), 30
kLightGoldenrodYellow (*wpilib.Color attribute*), 30
kLightGray (*wpilib.Color attribute*), 30
kLightGreen (*wpilib.Color attribute*), 30
kLightPink (*wpilib.Color attribute*), 30
kLightSalmon (*wpilib.Color attribute*), 30
kLightSeaGreen (*wpilib.Color attribute*), 30
kLightSkyBlue (*wpilib.Color attribute*), 30
kLightSlateGray (*wpilib.Color attribute*), 30
kLightSteelBlue (*wpilib.Color attribute*), 30
kLightYellow (*wpilib.Color attribute*), 30
kLime (*wpilib.Color attribute*), 30
kLimeGreen (*wpilib.Color attribute*), 30
kLinen (*wpilib.Color attribute*), 30
kMagenta (*wpilib.Color attribute*), 30
kMaroon (*wpilib.Color attribute*), 30
kMediumAquamarine (*wpilib.Color attribute*), 30
kMediumBlue (*wpilib.Color attribute*), 30
kMediumOrchid (*wpilib.Color attribute*), 30
kMediumPurple (*wpilib.Color attribute*), 30
kMediumSeaGreen (*wpilib.Color attribute*), 30
kMediumSlateBlue (*wpilib.Color attribute*), 30
kMediumSpringGreen (*wpilib.Color attribute*), 30
kMediumTurquoise (*wpilib.Color attribute*), 30
kMediumVioletRed (*wpilib.Color attribute*), 30
kMidnightBlue (*wpilib.Color attribute*), 30
kMilliMeters (*wpilib.Ultrasonic.DistanceUnit attribute*), 118
kMintcream (*wpilib.Color attribute*), 30
kMistyRose (*wpilib.Color attribute*), 30
kMoccasin (*wpilib.Color attribute*), 30
kMXP (*wpilib.I2C.Port attribute*), 58
kMXP (*wpilib.SerialPort.Port attribute*), 102
kMXP (*wpilib.SPI.Port attribute*), 86
kNavajoWhite (*wpilib.Color attribute*), 30
kNavy (*wpilib.Color attribute*), 30
kNone (*wpilib.DriverStation.MatchType attribute*), 44
kOff (*wpilib.DoubleSolenoid.Value attribute*), 43
kOff (*wpilib.Relay.Value attribute*), 81
kOldLace (*wpilib.Color attribute*), 31
kOlive (*wpilib.Color attribute*), 31
kOliveDrab (*wpilib.Color attribute*), 31
kOn (*wpilib.Relay.Value attribute*), 81
kOnboard (*wpilib.I2C.Port attribute*), 58
kOnboard (*wpilib.SerialPort.Port attribute*), 102
kOnboardCS0 (*wpilib.SPI.Port attribute*), 86
kOnboardCS1 (*wpilib.SPI.Port attribute*), 86
kOnboardCS2 (*wpilib.SPI.Port attribute*), 87
kOnboardCS3 (*wpilib.SPI.Port attribute*), 87
kOrange (*wpilib.Color attribute*), 31
kOrangeRed (*wpilib.Color attribute*), 31
kOrchid (*wpilib.Color attribute*), 31
kOversampleBits (*wpilib.AnalogGyro attribute*), 16
kPaleGoldenrod (*wpilib.Color attribute*), 31
kPaleGreen (*wpilib.Color attribute*), 31
kPaleTurquoise (*wpilib.Color attribute*), 31
kPaleVioletRed (*wpilib.Color attribute*), 31
kPapayaWhip (*wpilib.Color attribute*), 31
kParity_Even (*wpilib.SerialPort.Parity attribute*), 102
kParity_Mark (*wpilib.SerialPort.Parity attribute*), 102
kParity_None (*wpilib.SerialPort.Parity attribute*), 102
kParity_Odd (*wpilib.SerialPort.Parity attribute*), 102
kParity_Space (*wpilib.SerialPort.Parity attribute*), 102
kPDPChannels (*wpilib.SensorUtil attribute*), 100
kPeachPuff (*wpilib.Color attribute*), 31
kPeriodMultiplier_1X (*wpilib.PWM.PeriodMultiplier attribute*), 72
kPeriodMultiplier_2X (*wpilib.PWM.PeriodMultiplier attribute*), 72

kPeriodMultiplier_4X (*wpilib.PWM.PeriodMultiplier attribute*), 72
kPeru (*wpilib.Color attribute*), 31
kPink (*wpilib.Color attribute*), 31
kPlum (*wpilib.Color attribute*), 31
kPowderBlue (*wpilib.Color attribute*), 31
kPractice (*wpilib.DriverStation.MatchType attribute*), 44
kPulseLength (*wpilib.Counter.Mode attribute*), 35
kPurple (*wpilib.Color attribute*), 31
kPwmChannels (*wpilib.SensorUtil attribute*), 100
kQualification (*wpilib.DriverStation.MatchType attribute*), 45
kRange_16G (*wpilib.interfaces.Accelerometer.Range attribute*), 152
kRange_2G (*wpilib.interfaces.Accelerometer.Range attribute*), 152
kRange_4G (*wpilib.interfaces.Accelerometer.Range attribute*), 152
kRange_8G (*wpilib.interfaces.Accelerometer.Range attribute*), 152
kRate (*wpilib.interfaces.PIDSourceType attribute*), 157
kRearLeft (*wpilib.drive.RobotDriveBase.MotorType attribute*), 144
kRearRight (*wpilib.drive.RobotDriveBase.MotorType attribute*), 145
kRed (*wpilib.Color attribute*), 31
kRed (*wpilib.DriverStation.Alliance attribute*), 44
kRelayChannels (*wpilib.SensorUtil attribute*), 100
kResetOnFallingEdge (*wpilib.Encoder.IndexingType attribute*), 52
kResetOnRisingEdge (*wpilib.Encoder.IndexingType attribute*), 52
kResetWhileHigh (*wpilib.Encoder.IndexingType attribute*), 52
kResetWhileLow (*wpilib.Encoder.IndexingType attribute*), 53
kReverse (*wpilib.DoubleSolenoid.Value attribute*), 43
kReverse (*wpilib.Relay.Value attribute*), 81
kReverseOnly (*wpilib.Relay.Direction attribute*), 81
kRight (*wpilib.drive.RobotDriveBase.MotorType attribute*), 145
kRightHand (*wpilib.interfaces.GenericHID.Hand attribute*), 154
kRightRumble (*wpilib.interfaces.GenericHID.RumbleType attribute*), 154
kRightTrigger (*wpilib.XboxController.Axis attribute*), 121
kRightX (*wpilib.XboxController.Axis attribute*), 121
kRightY (*wpilib.XboxController.Axis attribute*), 121
kRisingEdge (*wpilib.InterruptableSensorBase.WaitResult attribute*), 60
kRisingPulse (*wpilib.AnalogTriggerType attribute*), 24
kRolloverTime (*wpilib.Timer attribute*), 116
kRosyBrown (*wpilib.Color attribute*), 31
kRoyalBlue (*wpilib.Color attribute*), 31
kS (*wpilib.controller.ArmFeedforward attribute*), 125
kS (*wpilib.controller.ElevatorFeedforward attribute*), 127
kS (*wpilib.controller.SimpleMotorFeedforward attribute*), 136
kS (*wpilib.controller.SimpleMotorFeedforwardMeters attribute*), 137
kSaddleBrown (*wpilib.Color attribute*), 31
kSalmon (*wpilib.Color attribute*), 31
kSamplesPerSecond (*wpilib.AnalogGyro attribute*), 16
kSandyBrown (*wpilib.Color attribute*), 31
kSeaGreen (*wpilib.Color attribute*), 31
kSeashell (*wpilib.Color attribute*), 31
kSemiperiod (*wpilib.Counter.Mode attribute*), 35
kSienna (*wpilib.Color attribute*), 31
kSilver (*wpilib.Color attribute*), 31
kSkyBlue (*wpilib.Color attribute*), 31
kSlateBlue (*wpilib.Color attribute*), 31
kSlateGray (*wpilib.Color attribute*), 31
kSnow (*wpilib.Color attribute*), 31
kSolenoidChannels (*wpilib.SensorUtil attribute*), 100
kSolenoidModules (*wpilib.SensorUtil attribute*), 100
kSpringGreen (*wpilib.Color attribute*), 31
kStart (*wpilib.XboxController.Button attribute*), 122
kState (*wpilib.AnalogTriggerType attribute*), 24
kSteelBlue (*wpilib.Color attribute*), 31
kStickLeft (*wpilib.XboxController.Button attribute*), 122
kStickRight (*wpilib.XboxController.Button attribute*), 122
kStopBits_One (*wpilib.SerialPort.StopBits attribute*), 102
kStopBits_OnePointFive (*wpilib.SerialPort.StopBits attribute*), 102
kStopBits_Two (*wpilib.SerialPort.StopBits attribute*), 102
kTan (*wpilib.Color attribute*), 31
kTeal (*wpilib.Color attribute*), 31
kTeamDeviceType (*wpilib.CAN attribute*), 25
kTeamManufacturer (*wpilib.CAN attribute*), 25
kThistle (*wpilib.Color attribute*), 31
kThrottleAxis (*wpilib.Joystick.AxisType attribute*), 64
kTimeout (*wpilib.InterruptableSensorBase.WaitResult attribute*), 60
kTomato (*wpilib.Color attribute*), 32
kTopButton (*wpilib.Joystick.ButtonType attribute*), 64

kTriggerButton (*wpilib.Joystick.ButtonType attribute*), 64
kTurquoise (*wpilib.Color attribute*), 32
kTwistAxis (*wpilib.Joystick.AxisType attribute*), 64
kTwoPulse (*wpilib.Counter.Mode attribute*), 35
kUnknown (*wpilib.interfaces.GenericHID.HIDType attribute*), 154
kUSB (*wpilib.SerialPort.Port attribute*), 102
kUSB1 (*wpilib.SerialPort.Port attribute*), 102
kUSB2 (*wpilib.SerialPort.Port attribute*), 102
kV (*wpilib.controller.ArmFeedforward attribute*), 125
kV (*wpilib.controller.ElevatorFeedforward attribute*), 127
kV (*wpilib.controller.SimpleMotorFeedforward attribute*), 136
kV (*wpilib.controller.SimpleMotorFeedforwardMeters attribute*), 137
kViolet (*wpilib.Color attribute*), 32
kWheat (*wpilib.Color attribute*), 32
kWhite (*wpilib.Color attribute*), 32
kWhiteSmoke (*wpilib.Color attribute*), 32
kX (*wpilib.XboxController.Button attribute*), 122
kXAxis (*wpilib.Joystick.AxisType attribute*), 64
kXInputArcadePad (*wpilib.interfaces.GenericHID.HIDType attribute*), 154
kXInputArcadeStick (*wpilib.interfaces.GenericHID.HIDType attribute*), 154
kXInputDancePad (*wpilib.interfaces.GenericHID.HIDType attribute*), 154
kXInputDrumKit (*wpilib.interfaces.GenericHID.HIDType attribute*), 154
kXInputFlightStick (*wpilib.interfaces.GenericHID.HIDType attribute*), 154
kXInputGamepad (*wpilib.interfaces.GenericHID.HIDType attribute*), 154
kXInputGuitar (*wpilib.interfaces.GenericHID.HIDType attribute*), 154
kXInputGuitar2 (*wpilib.interfaces.GenericHID.HIDType attribute*), 154
kXInputGuitar3 (*wpilib.interfaces.GenericHID.HIDType attribute*), 154
kXInputUnknown (*wpilib.interfaces.GenericHID.HIDType attribute*), 154
kXInputWheel (*wpilib.interfaces.GenericHID.HIDType attribute*), 154
kY (*wpilib.XboxController.Button attribute*), 122
kYAxis (*wpilib.Joystick.AxisType attribute*), 64
kYellow (*wpilib.Color attribute*), 32
kYellowGreen (*wpilib.Color attribute*), 32
kZAxis (*wpilib.Joystick.AxisType attribute*), 64

L

launch() (*wpilib.CameraServer class method*), 27
left (*wpilib.kinematics.DifferentialDriveWheelSpeeds attribute*), 162
left_fps (*wpilib.kinematics.DifferentialDriveWheelSpeeds attribute*), 162
length (*wpilib.CANData attribute*), 26
LinearFilter (*class in wpilib*), 67
LiveWindow (*class in wpilib*), 68
log() (*wpilib.geometry.Pose2d method*), 147
logger (*wpilib.RobotBase attribute*), 83

M

magnitude() (*wpilib.drive.Vector2d method*), 145
main() (*wpilib.RobotBase static method*), 83
maxAcceleration (*wpilib.trajectory.constraint.TrajectoryConstraint.M attribute*), 190
maxAcceleration() (*wpilib.trajectory.TrajectoryConfig method*), 181
maxAchievableAcceleration() (*wpilib.controller.ArmFeedforward method*), 125
maxAchievableAcceleration() (*wpilib.controller.ElevatorFeedforward method*), 127
maxAchievableAcceleration() (*wpilib.controller.SimpleMotorFeedforward method*), 136
maxAchievableAcceleration() (*wpilib.controller.SimpleMotorFeedforwardMeters method*), 137
maxAchievableVelocity() (*wpilib.controller.ArmFeedforward method*), 126
maxAchievableVelocity() (*wpilib.controller.ElevatorFeedforward method*), 127
maxAchievableVelocity() (*wpilib.controller.SimpleMotorFeedforward method*), 136
maxAchievableVelocity() (*wpilib.controller.SimpleMotorFeedforwardMeters method*), 138
maxVelocity() (*wpilib.trajectory.constraint.CentripetalAccelerationCo method*), 186
maxVelocity() (*wpilib.trajectory.constraint.DifferentialDriveKinematic method*), 187
maxVelocity() (*wpilib.trajectory.constraint.DifferentialDriveVoltageC method*), 188
maxVelocity() (*wpilib.trajectory.constraint.MecanumDriveKinematics method*), 189
maxVelocity() (*wpilib.trajectory.constraint.SwerveDrive3KinematicsC method*), 189
maxVelocity() (*wpilib.trajectory.constraint.SwerveDrive4KinematicsC method*), 189

maxVelocity() (*wpilib.trajectory.constraint.SwerveDrive6KinematicsConstraint method*), 190
maxVelocity() (*wpilib.trajectory.constraint.TrajectoryConstraint method*), 190
maxVelocity() (*wpilib.trajectory.TrajectoryConfig method*), 181
MecanumDrive (*class in wpilib.drive*), 143
MecanumDriveKinematics (*class in wpilib.kinematics*), 162
MecanumDriveKinematicsConstraint (*class in wpilib.trajectory.constraint*), 189
MecanumDriveOdometry (*class in wpilib.kinematics*), 164
MecanumDriveWheelSpeeds (*class in wpilib.kinematics*), 165
MedianFilter (*class in wpilib*), 69
minAcceleration (*wpilib.trajectory.constraint.TrajectoryConstraint.MinMax attribute*), 190
minAchievableAcceleration() (*wpilib.controller.ArmFeedforward method*), 126
minAchievableAcceleration() (*wpilib.controller.ElevatorFeedforward method*), 127
minAchievableAcceleration() (*wpilib.controller.SimpleMotorFeedforward method*), 136
minAchievableAcceleration() (*wpilib.controller.SimpleMotorFeedforwardMeters method*), 138
minAchievableVelocity() (*wpilib.controller.ArmFeedforward method*), 126
minAchievableVelocity() (*wpilib.controller.ElevatorFeedforward method*), 128
minAchievableVelocity() (*wpilib.controller.SimpleMotorFeedforward method*), 137
minAchievableVelocity() (*wpilib.controller.SimpleMotorFeedforwardMeters method*), 138
minMaxAcceleration() (*wpilib.trajectory.constraint.CentripetalAccelerationConstraint method*), 187
minMaxAcceleration() (*wpilib.trajectory.constraint.DifferentialDriveKinematicsConstraint method*), 187
minMaxAcceleration() (*wpilib.trajectory.constraint.DifferentialDriveVoltageConstraint method*), 189
minMaxAcceleration() (*wpilib.trajectory.constraint.MecanumDriveKinematicsConstraint method*), 189
minMaxAcceleration() (*wpilib.trajectory.constraint.SwerveDrive3KinematicsConstraint method*), 189
minMaxAcceleration() (*wpilib.trajectory.constraint.SwerveDrive4KinematicsConstraint method*), 189
minMaxAcceleration() (*wpilib.trajectory.constraint.SwerveDrive6KinematicsConstraint method*), 190
minMaxAcceleration() (*wpilib.trajectory.constraint.TrajectoryConstraint method*), 190
MotorSafety (*class in wpilib*), 69
movingAverage() (*wpilib.LinearFilter static method*), 68

N

name (*wpilib.ADXL345_I2C.Axes attribute*), 8
name (*wpilib.ADXL345_SPI.Axes attribute*), 9
name (*wpilib.ADXL362.Axes attribute*), 10
name (*wpilib.AnalogTriggerType attribute*), 24
name (*wpilib.Counter.Mode attribute*), 35
name (*wpilib.DoubleSolenoid.Value attribute*), 43
name (*wpilib.drive.RobotDriveBase.MotorType attribute*), 145
name (*wpilib.DriverStation.Alliance attribute*), 44
name (*wpilib.DriverStation.MatchType attribute*), 45
name (*wpilib.Encoder.IndexingType attribute*), 53
name (*wpilib.I2C.Port attribute*), 58
name (*wpilib.interfaces.Accelerometer.Range attribute*), 152
name (*wpilib.interfaces.CounterBase.EncodingType attribute*), 152
name (*wpilib.interfaces.GenericHID.Hand attribute*), 154
name (*wpilib.interfaces.GenericHID.HIDType attribute*), 154
name (*wpilib.interfaces.GenericHID.RumbleType attribute*), 154
name (*wpilib.interfaces.PIDSourceType attribute*), 157
name (*wpilib.InterruptableSensorBase.WaitResult attribute*), 60
name (*wpilib.Joystick.AxisType attribute*), 64
name (*wpilib.Joystick.ButtonType attribute*), 64
name (*wpilib.PWM.PeriodMultiplier attribute*), 72
name (*wpilib.Relay.Direction attribute*), 81
name (*wpilib.Relay.Value attribute*), 81
name (*wpilib.SerialPort.FlowControl attribute*), 101
name (*wpilib.SerialPort.Parity attribute*), 102
name (*wpilib.SerialPort.Port attribute*), 102
name (*wpilib.SerialPort.StopBits attribute*), 102
name (*wpilib.SerialPort.WriteBufferMode attribute*), 103
name (*wpilib.SPI.Port attribute*), 87
name (*wpilib.Ultrasonic.DistanceUnit attribute*), 118

name (*wpilib.XboxController.Axis attribute*), 121
name (*wpilib.XboxController.Button attribute*), 122
NidecBrushless (*class in wpilib*), 70
norm() (*wpilib.geometry.Translation2d method*), 150
normalize() (*wpilib.kinematics.DifferentialDriveWheelSpeeds method*), 162
normalize() (*wpilib.kinematics.MecanumDriveWheelSpeeds method*), 165
normalizeWheelSpeeds() (*wpilib.kinematics.SwerveDrive3Kinematics static method*), 166
normalizeWheelSpeeds() (*wpilib.kinematics.SwerveDrive4Kinematics static method*), 169
normalizeWheelSpeeds() (*wpilib.kinematics.SwerveDrive6Kinematics static method*), 171
normFeet() (*wpilib.geometry.Translation2d method*), 150
Notifier (*class in wpilib*), 71

O

omega (*wpilib.kinematics.ChassisSpeeds attribute*), 160
omega_dps (*wpilib.kinematics.ChassisSpeeds attribute*), 160

P

parameterize() (*wpilib.spline.SplineParameterizer static method*), 178
percentBusUtilization (*wpilib.CANStatus attribute*), 27
PIDController (*class in wpilib.controller*), 128
pidGet() (*wpilib.AnalogAccelerometer method*), 13
pidGet() (*wpilib.AnalogInput method*), 19
pidGet() (*wpilib.AnalogPotentiometer method*), 21
pidGet() (*wpilib.Encoder method*), 54
pidGet() (*wpilib.GyroBase method*), 58
pidGet() (*wpilib.interfaces.PIDSource method*), 157
pidGet() (*wpilib.Ultrasonic method*), 118
PIDOutput (*class in wpilib.interfaces*), 157
PIDSource (*class in wpilib.interfaces*), 157
PIDSourceType (*class in wpilib.interfaces*), 157
pidWrite() (*wpilib.interfaces.PIDOutput method*), 157
pidWrite() (*wpilib.NidecBrushless method*), 71
pidWrite() (*wpilib.PWMSpeedController method*), 75
pidWrite() (*wpilib.SpeedControllerGroup method*), 115
ping() (*wpilib.Ultrasonic method*), 118
pose (*wpilib.trajectory.Trajectory.State attribute*), 179
Pose2d (*class in wpilib.geometry*), 146
position (*wpilib.trajectory.TrapezoidProfile.State attribute*), 185
postListenerTask() (*wpilib.SmartDashboard static method*), 108
Potentiometer (*class in wpilib.interfaces*), 157
PowerDistributionPanel (*class in wpilib*), 77
Preferences (*class in wpilib*), 78
printEpochs() (*wpilib.Watchdog method*), 120
ProfiledPIDController (*class in wpilib.controller*), 130
publish() (*wpilib.SendableRegistry method*), 98
pulse() (*wpilib.DigitalOutput method*), 42
putBoolean() (*wpilib.Preferences method*), 79
putBoolean() (*wpilib.SmartDashboard static method*), 108
putBooleanArray() (*wpilib.SmartDashboard static method*), 108
putData() (*wpilib.SmartDashboard static method*), 109
putDouble() (*wpilib.Preferences method*), 79
putFloat() (*wpilib.Preferences method*), 80
putInt() (*wpilib.Preferences method*), 80
putLong() (*wpilib.Preferences method*), 80
putNumber() (*wpilib.SmartDashboard static method*), 109
putNumberArray() (*wpilib.SmartDashboard static method*), 109
putRaw() (*wpilib.SmartDashboard static method*), 109
putString() (*wpilib.Preferences method*), 80
putString() (*wpilib.SmartDashboard static method*), 110
putStringArray() (*wpilib.SmartDashboard static method*), 110
putValue() (*wpilib.SmartDashboard static method*), 110
PWM (*class in wpilib*), 72
PWM.PeriodMultiplier (*class in wpilib*), 72
PWMSparkMax (*class in wpilib*), 74
PWMSpeedController (*class in wpilib*), 75
PWMTalonFX (*class in wpilib*), 75
PWMTalonSRX (*class in wpilib*), 76
PWMVenom (*class in wpilib*), 76
PWMVictorSPX (*class in wpilib*), 77

Q

quinticControlVectorsFromWaypoints() (*wpilib.spline.SplineHelper static method*), 177
QuinticHermiteSpline (*class in wpilib.spline*), 175
quinticSplinesFromControlVectors() (*wpilib.spline.SplineHelper static method*), 178

R

radians() (*wpilib.geometry.Rotation2d method*), 148
RamseteController (*class in wpilib.controller*), 134
read() (*wpilib.I2C method*), 58

read() (*wpilib.SerialPort method*), 103
read() (*wpilib.SPI method*), 88
readAutoReceivedData() (*wpilib.SPI method*), 88
readFallingTimestamp() (*wpilib.InterruptableSensorBase method*), 60
readOnly() (*wpilib.I2C method*), 59
readPacketLatest() (*wpilib.CAN method*), 25
readPacketNew() (*wpilib.CAN method*), 25
readPacketTimeout() (*wpilib.CAN method*), 25
readRisingTimestamp() (*wpilib.InterruptableSensorBase method*), 61
rearLeft (*wpilib.kinematics.MecanumDriveWheelSpeeds attribute*), 165
rearLeft_fps (*wpilib.kinematics.MecanumDriveWheelSpeeds attribute*), 165
rearRight (*wpilib.kinematics.MecanumDriveWheelSpeeds attribute*), 165
rearRight_fps (*wpilib.kinematics.MecanumDriveWheelSpeeds attribute*), 165
receiveErrorCount (*wpilib.CANStatus attribute*), 27
red (*wpilib.Color attribute*), 32
red (*wpilib.Color8Bit attribute*), 32
relativeTo() (*wpilib.geometry.Pose2d method*), 147
relativeTo() (*wpilib.trajectory.Trajectory method*), 180
Relay (*class in wpilib*), 81
Relay.Direction (*class in wpilib*), 81
Relay.Value (*class in wpilib*), 81
remove() (*wpilib.DigitalGlitchFilter method*), 39
remove() (*wpilib.Preferences method*), 80
remove() (*wpilib.SendableRegistry method*), 98
removeAll() (*wpilib.Preferences method*), 80
reportError() (*wpilib.DriverStation static method*), 49
reportWarning() (*wpilib.DriverStation static method*), 49
requestInterrupts() (*wpilib.InterruptableSensorBase method*), 61
reset() (*wpilib.ADXRS450_Gyro method*), 11
reset() (*wpilib.AnalogEncoder method*), 14
reset() (*wpilib.AnalogGyro method*), 16
reset() (*wpilib.controller.PIDController method*), 129
reset() (*wpilib.controller.ProfiledPIDController method*), 132
reset() (*wpilib.Counter method*), 36
reset() (*wpilib.DutyCycleEncoder method*), 51
reset() (*wpilib.Encoder method*), 54
reset() (*wpilib.interfaces.CounterBase method*), 153
reset() (*wpilib.interfaces.Gyro method*), 156
reset() (*wpilib.LinearFilter method*), 68
reset() (*wpilib.MedianFilter method*), 69
reset() (*wpilib.SerialPort method*), 103
reset() (*wpilib.SlewRateLimiter method*), 105
reset() (*wpilib.Timer method*), 116
reset() (*wpilib.Watchdog method*), 120
resetAccumulator() (*wpilib.AnalogInput method*), 19
resetAccumulator() (*wpilib.SPI method*), 89
resetPosition() (*wpilib.kinematics.DifferentialDriveOdometry method*), 161
resetPosition() (*wpilib.kinematics.MecanumDriveOdometry method*), 164
resetPosition() (*wpilib.kinematics.SwerveDrive3Odometry method*), 167
resetPosition() (*wpilib.kinematics.SwerveDrive4Odometry method*), 170
resetPosition() (*wpilib.kinematics.SwerveDrive6Odometry method*), 173
resetTotalEnergy() (*wpilib.PowerDistributionPanel method*), 78
right (*wpilib.kinematics.DifferentialDriveWheelSpeeds attribute*), 162
right_fps (*wpilib.kinematics.DifferentialDriveWheelSpeeds attribute*), 162
RobotBase (*class in wpilib*), 82
RobotController (*class in wpilib*), 84
RobotDriveBase (*class in wpilib.drive*), 144
RobotDriveBase.MotorType (*class in wpilib.drive*), 144
robotInit() (*wpilib.IterativeRobotBase method*), 62
robotPeriodic() (*wpilib.IterativeRobotBase method*), 63
RobotState (*class in wpilib*), 85
rotate() (*wpilib.drive.Vector2d method*), 145
rotateBy() (*wpilib.geometry.Rotation2d method*), 148
rotateBy() (*wpilib.geometry.Translation2d method*), 150
rotation() (*wpilib.geometry.Pose2d method*), 147
rotation() (*wpilib.geometry.Transform2d method*), 149
Rotation2d (*class in wpilib.geometry*), 147
run() (*in module wpilib*), 6

S

sample() (*wpilib.trajectory.Trajectory method*), 180
scalarProject() (*wpilib.drive.Vector2d method*), 145
SD540 (*class in wpilib*), 86
Sendable (*class in wpilib*), 90
SendableBase (*class in wpilib*), 91
SendableBuilder (*class in wpilib*), 91
SendableBuilderImpl (*class in wpilib*), 93

SendableChooser (*class in wpilib*), 95
SendableRegistry (*class in wpilib*), 95
SensorUtil (*class in wpilib*), 99
serializeTrajectory() (*wpilib.trajectory.TrajectoryUtil static method*), 184
SerialPort (*class in wpilib*), 100
SerialPort.FlowControl (*class in wpilib*), 101
SerialPort.Parity (*class in wpilib*), 101
SerialPort.Port (*class in wpilib*), 102
SerialPort.StopBits (*class in wpilib*), 102
SerialPort.WriteBufferMode (*class in wpilib*), 102
Servo (*class in wpilib*), 104
set() (*wpilib.DigitalOutput method*), 42
set() (*wpilib.DoubleSolenoid method*), 44
set() (*wpilib.Error method*), 56
set() (*wpilib.interfaces.SpeedController method*), 158
set() (*wpilib.NidecBrushless method*), 71
set() (*wpilib.PWMSpeedController method*), 75
set() (*wpilib.Relay method*), 82
set() (*wpilib.Servo method*), 105
set() (*wpilib.Solenoid method*), 112
set() (*wpilib.SpeedControllerGroup method*), 115
setAccumulatorCenter() (*wpilib.AnalogInput method*), 19
setAccumulatorCenter() (*wpilib.SPI method*), 89
setAccumulatorDeadband() (*wpilib.AnalogInput method*), 19
setAccumulatorDeadband() (*wpilib.SPI method*), 89
setAccumulatorInitialValue() (*wpilib.AnalogInput method*), 19
setAccumulatorIntegratedCenter() (*wpilib.SPI method*), 89
setActuator() (*wpilib.SendableBuilder method*), 93
setActuator() (*wpilib.SendableBuilderImpl method*), 94
setAngle() (*wpilib.Servo method*), 105
setAutomaticMode() (*wpilib.Ultrasonic static method*), 118
setAutoTransmitData() (*wpilib.SPI method*), 89
setAverageBits() (*wpilib.AnalogInput method*), 19
setAveraged() (*wpilib.AnalogTrigger method*), 22
setBitTiming() (*wpilib.AddressableLED method*), 12
setBounds() (*wpilib.PWM method*), 73
setChipSelectActiveHigh() (*wpilib.SPI method*), 89
setChipSelectActiveLow() (*wpilib.SPI method*), 89
setClockActiveHigh() (*wpilib.SPI method*), 89
setClockActiveLow() (*wpilib.SPI method*), 89
setClockRate() (*wpilib.SPI method*), 89
setClosedLoopControl() (*wpilib.Compressor method*), 34
setConnectedFrequencyThreshold() (*wpilib.DutyCycleEncoder method*), 51
setConstraints() (*wpilib.controller.ProfiledPIDController method*), 133
setCurrentThreadPriority() (*in module wpilib*), 6
setD() (*wpilib.controller.PIDController method*), 129
setD() (*wpilib.controller.ProfiledPIDController method*), 133
setData() (*wpilib.AddressableLED method*), 12
setDeadband() (*wpilib.AnalogGyro method*), 16
setDeadband() (*wpilib.drive.RobotDriveBase method*), 145
setDefaultBoolean() (*wpilib.SmartDashboard static method*), 110
setDefaultBooleanArray() (*wpilib.SmartDashboard static method*), 110
setDefaultNumber() (*wpilib.SmartDashboard static method*), 110
setDefaultNumberArray() (*wpilib.SmartDashboard static method*), 111
setDefaultOption() (*wpilib.SendableChooser method*), 95
setDefaultRaw() (*wpilib.SmartDashboard static method*), 111
setDefaultString() (*wpilib.SmartDashboard static method*), 111
setDefaultStringArray() (*wpilib.SmartDashboard static method*), 111
setDefaultValue() (*wpilib.SmartDashboard static method*), 111
setDisabled() (*wpilib.PWM method*), 73
setDistancePerPulse() (*wpilib.Encoder method*), 54
setDistancePerRotation() (*wpilib.AnalogEncoder method*), 14
setDistancePerRotation() (*wpilib.DutyCycleEncoder method*), 51
setDistanceUnits() (*wpilib.Ultrasonic method*), 118
setDownSource() (*wpilib.Counter method*), 36
setDownSourceEdge() (*wpilib.Counter method*), 37
setEnabled() (*wpilib.controller.RamseteController method*), 135
setEnabled() (*wpilib.LiveWindow method*), 69
setEnabled() (*wpilib.Ultrasonic method*), 119
setEndVelocity() (*wpilib.trajectory.TrajectoryConfig method*), 181
setErrnoError() (*wpilib.ErrorBase method*), 56

setError() (*wpilib.ErrorBase method*), 56
setErrorHandler() (*wpilib.trajectory.TrajectoryGenerator static method*), 183
setErrorRange() (*wpilib.ErrorBase method*), 57
setExpiration() (*wpilib.MotorSafety method*), 70
setExternalDirectionMode() (*wpilib.Counter method*), 37
setFiltered() (*wpilib.AnalogTrigger method*), 22
setFlags() (*wpilib.SmartDashboard static method*), 111
setFlowControl() (*wpilib.SerialPort method*), 103
setGlobalError() (*wpilib.ErrorBase static method*), 57
setGlobalWPIError() (*wpilib.ErrorBase static method*), 57
setGoal() (*wpilib.controller.ProfiledPIDController method*), 133
setHandler() (*wpilib.Notifier method*), 71
setHSV() (*wpilib.AddressableLED.LEDData method*), 12
setI() (*wpilib.controller.PIDController method*), 129
setI() (*wpilib.controller.ProfiledPIDController method*), 133
setImaqError() (*wpilib.ErrorBase method*), 57
setIndexSource() (*wpilib.Encoder method*), 54
setIntegratorRange() (*wpilib.controller.PIDController method*), 130
setIntegratorRange() (*wpilib.controller.ProfiledPIDController method*), 133
setInverted() (*wpilib.interfaces.SpeedController method*), 158
setInverted() (*wpilib.NidecBrushless method*), 71
setInverted() (*wpilib.PWMSpeedController method*), 75
setInverted() (*wpilib.SpeedControllerGroup method*), 115
setKinematics() (*wpilib.trajectory.TrajectoryConfig method*), 181
setLED() (*wpilib.AddressableLED.LEDData method*), 12
setLength() (*wpilib.AddressableLED method*), 12
setLimitsDutyCycle() (*wpilib.AnalogTrigger method*), 22
setLimitsRaw() (*wpilib.AnalogTrigger method*), 22
setLimitsVoltage() (*wpilib.AnalogTrigger method*), 22
setLSBFirst() (*wpilib.SPI method*), 89
setMaxOutput() (*wpilib.drive.RobotDriveBase method*), 145
setMaxPeriod() (*wpilib.Counter method*), 37
setMaxPeriod() (*wpilib.Encoder method*), 54
setMaxPeriod() (*wpilib.interfaces.CounterBase method*), 153
setMinRate() (*wpilib.Encoder method*), 55
setMSBFirst() (*wpilib.SPI method*), 89
setName() (*wpilib.Notifier method*), 71
setName() (*wpilib.SendableRegistry method*), 98
setOffline() (*wpilib.Servo method*), 105
setOutput() (*wpilib.interfaces.GenericHID method*), 156
setOutputs() (*wpilib.interfaces.GenericHID method*), 156
setOversampleBits() (*wpilib.AnalogInput method*), 19
setP() (*wpilib.controller.PIDController method*), 130
setP() (*wpilib.controller.ProfiledPIDController method*), 133
setPeriodCycles() (*wpilib.DigitalGlitchFilter method*), 40
setPeriodMultiplier() (*wpilib.PWM method*), 73
setPeriodNanoSeconds() (*wpilib.DigitalGlitchFilter method*), 40
setPersistent() (*wpilib.SmartDashboard static method*), 112
setPID() (*wpilib.controller.PIDController method*), 130
setPID() (*wpilib.controller.ProfiledPIDController method*), 133
setPIDSourceType() (*wpilib.interfaces.PIDSource method*), 157
setPIDSourceType() (*wpilib.interfaces.Potentiometer method*), 158
setPIDSourceType() (*wpilib.Ultrasonic method*), 119
setPosition() (*wpilib.PWM method*), 74
setPulseDuration() (*wpilib.Solenoid method*), 112
setPulseLengthMode() (*wpilib.Counter method*), 37
setPWMRate() (*wpilib.DigitalOutput method*), 42
setQuickStopAlpha() (*wpilib.drive.DifferentialDrive method*), 140
setQuickStopThreshold() (*wpilib.drive.DifferentialDrive method*), 140
setRange() (*wpilib.ADXL345_I2C method*), 8
setRange() (*wpilib.ADXL345_SPI method*), 9
setRange() (*wpilib.ADXL362 method*), 10
setRange() (*wpilib.BuiltInAccelerometer method*), 24
setRange() (*wpilib.interfaces.Accelerometer method*), 152
setRaw() (*wpilib.PWM method*), 74
setRawBounds() (*wpilib.PWM method*), 74

setReadBufferSize() (*wpilib.SerialPort method*), 103
setReversed() (*wpilib.trajectory.TrajectoryConfig method*), 182
setReverseDirection() (*wpilib.Counter method*), 37
setReverseDirection() (*wpilib.Encoder method*), 55
setRGB() (*wpilib.AddressableLED.LEDData method*), 12
setRightSideInverted() (*wpilib.drive.DifferentialDrive method*), 141
setRightSideInverted() (*wpilib.drive.MecanumDrive method*), 144
setRobotPose() (*wpilib.simulation.Field2d method*), 174
setRumble() (*wpilib.interfaces.GenericHID method*), 156
setSafeState() (*wpilib.SendableBuilder method*), 93
setSafeState() (*wpilib.SendableBuilderImpl method*), 94
setSafetyEnabled() (*wpilib.MotorSafety method*), 70
setSampleDataOnFalling() (*wpilib.SPI method*), 89
setSampleDataOnLeadingEdge() (*wpilib.SPI method*), 90
setSampleDataOnRising() (*wpilib.SPI method*), 90
setSampleDataOnTrailingEdge() (*wpilib.SPI method*), 90
setSampleRate() (*wpilib.AnalogInput static method*), 19
setSamplesToAverage() (*wpilib.Counter method*), 37
setSamplesToAverage() (*wpilib.Encoder method*), 55
setSemiPeriodMode() (*wpilib.Counter method*), 37
setSensitivity() (*wpilib.AnalogAccelerometer method*), 13
setSensitivity() (*wpilib.AnalogGyro method*), 16
setSetpoint() (*wpilib.controller.PIDController method*), 130
setSimDevice() (*wpilib.AnalogInput method*), 19
setSimDevice() (*wpilib.DigitalInput method*), 41
setSimDevice() (*wpilib.DigitalOutput method*), 42
setSimDevice() (*wpilib.Encoder method*), 55
setSmartDashboardType() (*wpilib.SendableBuilder method*), 93
setSmartDashboardType() (*wpilib.SendableBuilderImpl method*), 94
setSpeed() (*wpilib.PWM method*), 74
setStartVelocity() (*wpilib.trajectory.TrajectoryConfig method*), 182
setSubsystem() (*wpilib.SendableRegistry method*), 99
setSyncTime() (*wpilib.AddressableLED method*), 12
setTable() (*wpilib.SendableBuilderImpl method*), 94
setThreadPriority() (*in module wpilib*), 7
setThrottleChannel() (*wpilib.Joystick method*), 66
setTimeout() (*wpilib.SerialPort method*), 103
setTimeout() (*wpilib.Watchdog method*), 120
setTolerance() (*wpilib.controller.PIDController method*), 130
setTolerance() (*wpilib.controller.ProfiledPIDController method*), 134
setTolerance() (*wpilib.controller.RamseteController method*), 135
setTwistChannel() (*wpilib.Joystick method*), 66
setUpdateTable() (*wpilib.SendableBuilder method*), 93
setUpdateTable() (*wpilib.SendableBuilderImpl method*), 94
setUpdateWhenEmpty() (*wpilib.Counter method*), 38
setUpDownCounterMode() (*wpilib.Counter method*), 37
setUpSource() (*wpilib.Counter method*), 37
setUpSourceEdge() (*wpilib.Counter method*), 38
setUpSourceEdge() (*wpilib.InterruptableSensorBase method*), 61
setVoltage() (*wpilib.AnalogOutput method*), 20
setVoltage() (*wpilib.interfaces.SpeedController method*), 158
setWPIError() (*wpilib.ErrorBase method*), 57
setWriteBufferMode() (*wpilib.SerialPort method*), 104
setWriteBufferSize() (*wpilib.SerialPort method*), 104
setXChannel() (*wpilib.Joystick method*), 66
setYChannel() (*wpilib.Joystick method*), 66
setZChannel() (*wpilib.Joystick method*), 67
setZero() (*wpilib.AnalogAccelerometer method*), 13
setZeroLatch() (*wpilib.PWM method*), 74
SimpleMotorFeedforward (*class in wpilib.controller*), 135
SimpleMotorFeedforwardMeters (*class in wpilib.controller*), 137
sin() (*wpilib.geometry.Rotation2d method*), 148
singlePoleIIR() (*wpilib.LinearFilter static method*), 68
SlewRateLimiter (*class in wpilib*), 105
SmartDashboard (*class in wpilib*), 106

Solenoid (*class in wpilib*), 112
SolenoidBase (*class in wpilib*), 113
Spark (*class in wpilib*), 114
speed (*wpilib.kinematics.SwerveModuleState attribute*), 174
speed_fps (*wpilib.kinematics.SwerveModuleState attribute*), 174
SpeedController (*class in wpilib.interfaces*), 158
SpeedControllerGroup (*class in wpilib*), 114
SPI (*class in wpilib*), 86
SPI.Port (*class in wpilib*), 86
Spline3 (*class in wpilib.spline*), 176
Spline3.ControlVector (*class in wpilib.spline*), 176
Spline5 (*class in wpilib.spline*), 176
Spline5.ControlVector (*class in wpilib.spline*), 176
SplineHelper (*class in wpilib.spline*), 177
SplineParameterizer (*class in wpilib.spline*), 178
splinePointsFromSplines() (*wpilib.trajectory.TrajectoryGenerator static method*), 183
start() (*wpilib.AddressableLED method*), 13
start() (*wpilib.Compressor method*), 34
start() (*wpilib.Timer method*), 117
startAutoRate() (*wpilib.SPI method*), 90
startAutoTrigger() (*wpilib.SPI method*), 90
startCompetition() (*wpilib.IterativeRobot method*), 62
startCompetition() (*wpilib.RobotBase method*), 83
startCompetition() (*wpilib.TimedRobot method*), 115
startListeners() (*wpilib.SendableBuilderImpl method*), 94
startLiveWindowMode() (*wpilib.SendableBuilderImpl method*), 94
startPeriodic() (*wpilib.Notifier method*), 71
startPulse() (*wpilib.Solenoid method*), 113
startSingle() (*wpilib.Notifier method*), 71
startVelocity() (*wpilib.trajectory.TrajectoryConfig method*), 182
states() (*wpilib.trajectory.Trajectory method*), 180
statusIsFatal() (*wpilib.ErrorBase method*), 57
stop() (*wpilib.AddressableLED method*), 13
stop() (*wpilib.Compressor method*), 34
stop() (*wpilib.Notifier method*), 71
stop() (*wpilib.Timer method*), 117
stopAuto() (*wpilib.SPI method*), 90
stopListeners() (*wpilib.SendableBuilderImpl method*), 94
stopLiveWindowMode() (*wpilib.SendableBuilderImpl method*), 94
stopMotor() (*wpilib.drive.DifferentialDrive method*), 141
stopMotor() (*wpilib.drive.KilloughDrive method*), 143
stopMotor() (*wpilib.drive.MecanumDrive method*), 144
stopMotor() (*wpilib.drive.RobotDriveBase method*), 145
stopMotor() (*wpilib.interfaces.SpeedController method*), 158
stopMotor() (*wpilib.MotorSafety method*), 70
stopMotor() (*wpilib.NidecBrushless method*), 71
stopMotor() (*wpilib.PWM method*), 74
stopMotor() (*wpilib.PWMSpeedController method*), 75
stopMotor() (*wpilib.Relay method*), 82
stopMotor() (*wpilib.SpeedControllerGroup method*), 115
stopPacketRepeating() (*wpilib.CAN method*), 26
suppressTimeoutMessage() (*wpilib.Watchdog method*), 120
SwerveDrive3Kinematics (*class in wpilib.kinematics*), 166
SwerveDrive3KinematicsConstraint (*class in wpilib.trajectory.constraint*), 189
SwerveDrive3Odometry (*class in wpilib.kinematics*), 167
SwerveDrive4Kinematics (*class in wpilib.kinematics*), 168
SwerveDrive4KinematicsConstraint (*class in wpilib.trajectory.constraint*), 189
SwerveDrive4Odometry (*class in wpilib.kinematics*), 170
SwerveDrive6Kinematics (*class in wpilib.kinematics*), 171
SwerveDrive6KinematicsConstraint (*class in wpilib.trajectory.constraint*), 190
SwerveDrive6Odometry (*class in wpilib.kinematics*), 172
SwerveModuleState (*class in wpilib.kinematics*), 174

T

t (*wpilib.trajectory.Trajectory.State attribute*), 179
Talon (*class in wpilib*), 115
tan() (*wpilib.geometry.Rotation2d method*), 148
tankDrive() (*wpilib.drive.DifferentialDrive method*), 141
teleopInit() (*wpilib.IterativeRobotBase method*), 63
teleopPeriodic() (*wpilib.IterativeRobotBase method*), 63
testInit() (*wpilib.IterativeRobotBase method*), 63

testPeriodic() (*wpilib.IterativeRobotBase method*), 63
TimedRobot (*class in wpilib*), 115
timeLeftUntil() (*wpilib.trajectory.TrapezoidProfile method*), 186
timeParameterizeTrajectory() (*wpilib.trajectory.TrajectoryParameterizer static method*), 184
Timer (*class in wpilib*), 116
timestamp (*wpilib.CANData attribute*), 26
toChassisSpeeds() (*wpilib.kinematics.DifferentialDriveKinematics method*), 160
toChassisSpeeds() (*wpilib.kinematics.MecanumDriveKinematics method*), 163
toChassisSpeeds() (*wpilib.kinematics.SwerveDrive3Kinematics method*), 166
toChassisSpeeds() (*wpilib.kinematics.SwerveDrive4Kinematics method*), 169
toChassisSpeeds() (*wpilib.kinematics.SwerveDrive6Kinematics method*), 172
toPathweaverJson() (*wpilib.trajectory.TrajectoryUtil static method*), 184
toSwerveModuleStates() (*wpilib.kinematics.SwerveDrive3Kinematics method*), 166
toSwerveModuleStates() (*wpilib.kinematics.SwerveDrive4Kinematics method*), 169
toSwerveModuleStates() (*wpilib.kinematics.SwerveDrive6Kinematics method*), 172
totalTime() (*wpilib.trajectory.Trajectory method*), 180
totalTime() (*wpilib.trajectory.TrapezoidProfile method*), 186
toWheelSpeeds() (*wpilib.kinematics.DifferentialDriveKinematics method*), 160
toWheelSpeeds() (*wpilib.kinematics.MecanumDriveKinematics method*), 163
trackWidth (*wpilib.kinematics.DifferentialDriveKinematics attribute*), 160
Trajectory (*class in wpilib.trajectory*), 179
Trajectory.State (*class in wpilib.trajectory*), 179
TrajectoryConfig (*class in wpilib.trajectory*), 180
TrajectoryConstraint (*class in wpilib.trajectory.constraint*), 190
TrajectoryConstraint.MinMax (*class in wpilib.trajectory.constraint*), 190
TrajectoryGenerator (*class in wpilib.trajectory*), 182
TrajectoryParameterizer (*class in wpilib.trajectory*), 184
TrajectoryUtil (*class in wpilib.trajectory*), 184
transaction() (*wpilib.I2C method*), 59
transaction() (*wpilib.SPI method*), 90
Transform2d (*class in wpilib.geometry*), 149
transformBy() (*wpilib.geometry.Pose2d method*), 147
transformBy() (*wpilib.trajectory.Trajectory method*), 180
translation() (*wpilib.geometry.Pose2d method*), 147
translation() (*wpilib.geometry.Transform2d method*), 149
Translation2d (*class in wpilib.geometry*), 149
transmitErrorCount (*wpilib.CANStatus attribute*), 27
TrapezoidProfile (*class in wpilib.trajectory*), 185
TrapezoidProfile.Constraints (*class in wpilib.trajectory*), 185
TrapezoidProfile.State (*class in wpilib.trajectory*), 185
Twist2d (*class in wpilib.geometry*), 151
txFullCount (*wpilib.CANStatus attribute*), 27

U

Ultrasonic (*class in wpilib*), 117
Ultrasonic.DistanceUnit (*class in wpilib*), 117
update() (*wpilib.kinematics.DifferentialDriveOdometry method*), 161
update() (*wpilib.kinematics.MecanumDriveOdometry method*), 164
update() (*wpilib.kinematics.SwerveDrive3Odometry method*), 168
update() (*wpilib.kinematics.SwerveDrive4Odometry method*), 170
update() (*wpilib.kinematics.SwerveDrive6Odometry method*), 173
update() (*wpilib.SendableRegistry method*), 99
updateDutyCycle() (*wpilib.DigitalOutput method*), 42
updateTable() (*wpilib.SendableBuilderImpl method*), 94
updateValues() (*wpilib.LiveWindow method*), 69
updateValues() (*wpilib.SmartDashboard static method*), 112
updateWithTime() (*wpilib.kinematics.MecanumDriveOdometry method*), 164
updateWithTime() (*wpilib.kinematics.SwerveDrive3Odometry method*), 168
updateWithTime() (*wpilib.kinematics.SwerveDrive4Odometry method*), 170

updateWithTime() (*wpilib.kinematics.SwerveDrive6Odometry method*), 173

V

Vector2d (*class in wpilib.drive*), 145
velocity (*wpilib.trajectory.Trajectory.State attribute*), 179
velocity (*wpilib.trajectory.TrapezoidProfile.State attribute*), 185
verifySensor() (*wpilib.I2C method*), 59
Victor (*class in wpilib*), 119
VictorSP (*class in wpilib*), 119
vx (*wpilib.kinematics.ChassisSpeeds attribute*), 160
vx_fps (*wpilib.kinematics.ChassisSpeeds attribute*), 160
vy (*wpilib.kinematics.ChassisSpeeds attribute*), 160
vy_fps (*wpilib.kinematics.ChassisSpeeds attribute*), 160

W

wait() (*in module wpilib*), 7
waitForData() (*wpilib.DriverStation method*), 49
waitForInterrupt() (*wpilib.InterruptableSensorBase method*), 61
wakeupWaitForData() (*wpilib.DriverStation method*), 49
Watchdog (*class in wpilib*), 120
write() (*wpilib.I2C method*), 59
write() (*wpilib.SerialPort method*), 104
write() (*wpilib.SPI method*), 90
writeBulk() (*wpilib.I2C method*), 60
writePacket() (*wpilib.CAN method*), 26
writePacketRepeating() (*wpilib.CAN method*), 26
writeRTRFrame() (*wpilib.CAN method*), 26

X

x (*wpilib.drive.Vector2d attribute*), 145
x (*wpilib.geometry.Translation2d attribute*), 150
x (*wpilib.spline.Spline3.ControlVector attribute*), 176
x (*wpilib.spline.Spline5.ControlVector attribute*), 177
X() (*wpilib.geometry.Translation2d method*), 150
x_feet (*wpilib.geometry.Translation2d attribute*), 150
XAxis (*wpilib.ADXL345_I2C.AllAxes attribute*), 7
XAxis (*wpilib.ADXL345_SPI.AllAxes attribute*), 8
XAxis (*wpilib.ADXL362.AllAxes attribute*), 10
XboxController (*class in wpilib*), 121
XboxController.Axis (*class in wpilib*), 121
XboxController.Button (*class in wpilib*), 121

Y

y (*wpilib.drive.Vector2d attribute*), 145
y (*wpilib.geometry.Translation2d attribute*), 150
y (*wpilib.spline.Spline3.ControlVector attribute*), 176
y (*wpilib.spline.Spline5.ControlVector attribute*), 177
Y() (*wpilib.geometry.Translation2d method*), 150
y_feet (*wpilib.geometry.Translation2d attribute*), 150
YAxis (*wpilib.ADXL345_I2C.AllAxes attribute*), 7
YAxis (*wpilib.ADXL345_SPI.AllAxes attribute*), 8
YAxis (*wpilib.ADXL362.AllAxes attribute*), 10

Z

ZAxis (*wpilib.ADXL345_I2C.AllAxes attribute*), 7
ZAxis (*wpilib.ADXL345_SPI.AllAxes attribute*), 8
ZAxis (*wpilib.ADXL362.AllAxes attribute*), 10