RobotPy WPILib Documentation

Release master

RobotPy development team

February 17, 2017

RobotPy WPILib is the source code for a 100% python 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.

wpilib Package

This is the core of WPILib.

wpilib.ADXL345_I2C(port, range)	ADXL345 accelerometer device via i2c
wpilib.ADXL345_SPI(port, range)	ADXL345 accelerometer device via spi
wpilib.ADXL362(range[, port])	ADXL362 SPI Accelerometer.
wpilib.ADXRS450_Gyro([port])	Use a rate gyro to return the robots heading relative to a starting position.
wpilib.AnalogAccelerometer(channel)	Analog Accelerometer
wpilib.AnalogGyro(channel[, ...])	Interface to a gyro device via an *AnalogInput*
wpilib.AnalogInput(channel)	Analog input
wpilib.AnalogOutput(channel)	Analog output
wpilib.AnalogPotentiometer(channel)	Reads a potentiometer via an *AnalogInput*
wpilib.AnalogTrigger(channel)	Converts an analog signal into a digital signal
wpilib.AnalogTriggerOutput(...)	Represents a specific output from an *AnalogTrigger*
wpilib.BuiltInAccelerometer([range])	Built-in accelerometer device
wpilib.CameraServer	Provides a way to launch an out of process cscore-based camera service instance,
wpilib.CANJaguar(*args, **kwargs)	
wpilib.CANTalon(*args, **kwargs)	
wpilib.Compressor([module])	Class for operating a compressor connected to a PCM (Pneumatic Control Modul
wpilib.ControllerPower	Provides access to power levels on the roboRIO
wpilib.Counter(*args, **kwargs)	Counts the number of ticks on a *DigitalInput* channel.
wpilib.DigitalGlitchFilter()	Class to enable glitch filtering on a set of digital inputs.
wpilib.DigitalInput(channel)	Reads a digital input.
wpilib.DigitalOutput(channel)	Writes to a digital output

Table 1.1 – continued from previous page

wpilib.DigitalSource(channel, ...)	DigitalSource Interface.
wpilib.DoubleSolenoid(*args, ...)	Controls 2 channels of high voltage Digital Output on the PCM.
wpilib.DriverStation()	Provide access to the network communication data to / from the Driver Station.
wpilib.Encoder(*args, **kwargs)	Class to read quadrature encoders.
wpilib.Filter(source)	Superclass for filters
wpilib.GearTooth(channel[, ...])	Interface to the gear tooth sensor supplied by FIRST
wpilib.GyroBase()	GyroBase is the common base class for Gyro implementations such as AnalogG
wpilib.I2C(port, deviceAddress[, simPort])	I2C bus interface class.
wpilib.interfaces.GamepadBase(port)	GamepadBase Interface.
wpilib.interfaces.GenericHID(port)	GenericHID Interface.
wpilib.InterruptableSensorBase()	Base for sensors to be used with interrupts
wpilib.IterativeRobot()	IterativeRobot implements a specific type of Robot Program framework, extendin
wpilib.Jaguar(channel)	Texas Instruments / Vex Robotics Jaguar Speed Controller as a PWM device.
wpilib.Joystick(port[, ...])	Handle input from standard Joysticks connected to the Driver Station.
wpilib.LinearDigitalFilter(...)	This class implements a linear, digital filter.
wpilib.LiveWindow	The public interface for putting sensors and actuators on the LiveWindow.
wpilib.LiveWindowSendable	A special type of object that can be displayed on the live window.
wpilib.MotorSafety()	Provides mechanisms to safely shutdown motors if they aren't updated often enou
wpilib.PIDController(*args, ...)	Can be used to control devices via a PID Control Loop.
wpilib.PowerDistributionPanel([...])	Use to obtain voltage, current, temperature, power, and energy from the
wpilib.Preferences()	Provides a relatively simple way to save important values to the roboRIO to acces
wpilib.PWM(channel)	Raw interface to PWM generation in the FPGA.
wpilib.PWMSpeedController(channel)	Common base class for all PWM Speed Controllers.
wpilib.Relay(channel[, direction])	Controls VEX Robotics Spike style relay outputs.
wpilib.Resource(size)	Tracks resources in the program.
wpilib.RobotBase()	Implement a Robot Program framework.
wpilib.RobotDrive(*args, **kwargs)	Operations on a robot drivetrain based on a definition of the motor configuration.
wpilib.RobotState	Provides an interface to determine the current operating state of the robot code.
wpilib.SafePWM(channel)	A raw PWM interface that implements the MotorSafety interface
wpilib.SampleRobot()	A simple robot base class that knows the standard FRC competition states (disabl
wpilib.SD540(channel)	Mindsensors SD540 Speed Controller
wpilib.Sendable	The base interface for objects that can be sent over the network
wpilib.SendableChooser()	A useful tool for presenting a selection of options to be displayed on
wpilib.SensorBase	Base class for all sensors
wpilib.Servo(channel)	Standard hobby style servo
wpilib.SmartDashboard	The bridge between robot programs and the SmartDashboard on the laptop
wpilib.Solenoid(*args, **kwargs)	Solenoid class for running high voltage Digital Output.
wpilib.SolenoidBase(moduleNumber)	SolenoidBase class is the common base class for the Solenoid and DoubleSolenoi
wpilib.Spark(channel)	REV Robotics SPARK Speed Controller
wpilib.SPI(port[, simPort])	Represents a SPI bus port
wpilib.Talon(channel)	Cross the Road Electronics (CTRE) Talon and Talon SR Speed Controller via PW
wpilib.TalonSRX(channel)	Cross the Road Electronics (CTRE) Talon SRX Speed Controller via PWM
wpilib.Timer()	Provides time-related functionality for the robot
wpilib.Ultrasonic(pingChannel, ...)	Ultrasonic rangefinder control
wpilib.Utility	Contains global utility functions
wpilib.Victor(channel)	VEX Robotics Victor 888 Speed Controller via PWM
wpilib.VictorSP(channel)	VEX Robotics Victor SP Speed Controller via PWM
wpilib.XboxController(port)	Handle input from Xbox 360 or Xbox One controllers connected to the Driver Sta

ADXL345_I2C

class wpilib.**ADXL345_I2C**(*port*, *range*, *address=None*)
Bases: *wpilib.SensorBase*

ADXL345 accelerometer device via i2c

Constructor.

Parameters

- **port** (*I2C.Port*) – The I2C port the accelerometer is attached to.
- **range** (*ADXL345_I2C.Range*) – The range (+ or -) that the accelerometer will measure.
- **address** – the I2C address of the accelerometer (0x1D or 0x53)

class Axes
Bases: object

kX = 0

kY = 2

kZ = 4

class ADXL345_I2C.**Range**
Bases: object

k16G = 3

k2G = 0

k4G = 1

k8G = 2

ADXL345_I2C.**free**()

ADXL345_I2C.**getAcceleration**(*axis*)
Get the acceleration of one axis in Gs.

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

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

ADXL345_I2C.**getAccelerations**()
Get the acceleration of all axes in Gs.

Returns X,Y,Z tuple of acceleration measured on all axes of the ADXL345 in Gs.

ADXL345_I2C.**getX**()
Get the x axis acceleration

Returns The acceleration along the x axis in g-forces

ADXL345_I2C.**getY**()
Get the y axis acceleration

Returns The acceleration along the y axis in g-forces

ADXL345_I2C.**getZ**()
Get the z axis acceleration

Returns The acceleration along the z axis in g-forces

ADXL345_I2C.**kAddress = 29**

ADXL345_I2C.**kDataFormatRegister** = 49

ADXL345_I2C.**kDataFormat_FullRes** = 8

ADXL345_I2C.**kDataFormat_IntInvert** = 32

ADXL345_I2C.**kDataFormat_Justify** = 4

ADXL345_I2C.**kDataFormat_SPI** = 64

ADXL345_I2C.**kDataFormat_SelfTest** = 128

ADXL345_I2C.**kDataRegister** = 50

ADXL345_I2C.**kGsPerLSB** = 0.00390625

ADXL345_I2C.**kPowerCtlRegister** = 45

ADXL345_I2C.**kPowerCtl_AutoSleep** = 16

ADXL345_I2C.**kPowerCtl_Link** = 32

ADXL345_I2C.**kPowerCtl_Measure** = 8

ADXL345_I2C.**kPowerCtl_Sleep** = 4

ADXL345_I2C.**setRange**(*range*)
Set the measuring range of the accelerometer.

Parameters **range** (*ADXL345_I2C.Range*) – The maximum acceleration, positive or negative, that the accelerometer will measure.

ADXL345_SPI

class wpilib.**ADXL345_SPI**(*port*, *range*)
Bases: *wpilib.SensorBase*

ADXL345 accelerometer device via spi

Constructor. Use this when the device is the first/only device on the bus

Parameters

- **port** (*SPI.Port*) – The SPI port that the accelerometer is connected to
- **range** (*ADXL345_SPI.Range*) – The range (+ or -) that the accelerometer will measure.

class **Axes**
Bases: object

kX = 0

kY = 2

kZ = 4

class ADXL345_SPI.**Range**
Bases: object

k16G = 3

k2G = 0

k4G = 1

k8G = 2

ADXL345_SPI.**free**()

ADXL345_SPI.**getAcceleration**(*axis*)
Get the acceleration of one axis in Gs.

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

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

ADXL345_SPI.**getAccelerations**()
Get the acceleration of all axes in Gs.

Returns X,Y,Z tuple of acceleration measured on all axes of the ADXL345 in Gs.

ADXL345_SPI.**getX**()
Get the x axis acceleration

Returns The acceleration along the x axis in g-forces

ADXL345_SPI.**getY**()
Get the y axis acceleration

Returns The acceleration along the y axis in g-forces

ADXL345_SPI.**getZ**()
Get the z axis acceleration

Returns The acceleration along the z axis in g-forces

ADXL345_SPI.**kAddress_MultiByte = 64**

ADXL345_SPI.**kAddress_Read = 128**

ADXL345_SPI.**kDataFormatRegister = 49**

ADXL345_SPI.**kDataFormat_FullRes = 8**

ADXL345_SPI.**kDataFormat_IntInvert = 32**

ADXL345_SPI.**kDataFormat_Justify = 4**

ADXL345_SPI.**kDataFormat_SPI = 64**

ADXL345_SPI.**kDataFormat_SelfTest = 128**

ADXL345_SPI.**kDataRegister = 50**

ADXL345_SPI.**kGsPerLSB = 0.00390625**

ADXL345_SPI.**kPowerCtlRegister = 45**

ADXL345_SPI.**kPowerCtl_AutoSleep = 16**

ADXL345_SPI.**kPowerCtl_Link = 32**

ADXL345_SPI.**kPowerCtl_Measure = 8**

ADXL345_SPI.**kPowerCtl_Sleep = 4**

ADXL345_SPI.**setRange**(*range*)
Set the measuring range of the accelerometer.

Parameters **range** (*ADXL345_SPI.Range*) – The maximum acceleration, positive or negative, that the accelerometer will measure.

ADXL362

class wpilib.**ADXL362**(*range*, *port=None*)
Bases: *wpilib.SensorBase*

ADXL362 SPI Accelerometer.

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

Constructor.

Parameters

- **range** (*ADXL362.Range*) – The range (+ or -) that the accelerometer will measure.
- **port** (*SPI.Port*) – The SPI port that the accelerometer is connected to

class Axes
Bases: object

kX = 0

kY = 2

kZ = 4

class ADXL362.**Range**
Bases: object

k16G = 3

k2G = 0

k4G = 1

k8G = 2

ADXL362.**free**()

ADXL362.**getAcceleration**(*axis*)
Get the acceleration of one axis in Gs.

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

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

ADXL362.**getAccelerations**()
Get the acceleration of all axes in Gs.

Returns X,Y,Z tuple of acceleration measured on all axes in Gs.

ADXL362.**getX**()
Get the x axis acceleration

Returns The acceleration along the x axis in g-forces

ADXL362.**getY**()
Get the y axis acceleration

Returns The acceleration along the y axis in g-forces

ADXL362.**getZ**()
Get the z axis acceleration

Returns The acceleration along the z axis in g-forces

ADXL362.**kDataRegister = 14**

ADXL362.**kFilterCtlRegister** = 44

ADXL362.**kFilterCtl_ODR_100Hz** = 3

ADXL362.**kFilterCtl_Range2G** = 0

ADXL362.**kFilterCtl_Range4G** = 64

ADXL362.**kFilterCtl_Range8G** = 128

ADXL362.**kPartIdRegister** = 2

ADXL362.**kPowerCtlRegister** = 45

ADXL362.**kPowerCtl_AutoSleep** = 4

ADXL362.**kPowerCtl_Measure** = 2

ADXL362.**kPowerCtl_UltraLowNoise** = 32

ADXL362.**kRegRead** = 11

ADXL362.**kRegWrite** = 10

ADXL362.**setRange**(*range*)
Set the measuring range of the accelerometer.

Parameters **range** (*ADXL362.Range*) – The maximum acceleration, positive or negative, that the accelerometer will measure.

ADXRS450_Gyro

class wpilib.**ADXRS450_Gyro**(*port=None*)
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.

Constructor.

Parameters **port** (*SPI.Port*) – The SPI port that the gyro is connected to

calibrate()
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.

Note: Usually you don't need to call this, as it's called when the object is first created. If you do, it will freeze your robot for 5 seconds

---

free()
Delete (free) the spi port used for the gyro and stop accumulating.

getAngle()
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.

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

getRate()
Return the rate of rotation of the gyro

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

Returns the current rate in degrees per second

kCalibrationSampleTime = 5.0

kDegreePerSecondPerLSB = 0.0125

kFaultRegister = 10

kHiCSTRegister = 6

kLoCSTRegister = 4

kPIDRegister = 12

kQuadRegister = 8

kRateRegister = 0

kSNHighRegister = 14

kSNLowRegister = 16

kSamplePeriod = 0.001

kTemRegister = 2

reset()
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.

AnalogAccelerometer

class wpilib.**AnalogAccelerometer**(*channel*)
Bases: *wpilib.LiveWindowSendable*

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.

Constructor. Create a new instance of Accelerometer from either an existing AnalogChannel or from an analog channel port index.

Parameters **channel** – port index or an already initialized *AnalogInput*

class PIDSourceType
Bases: object

A description for the type of output value to provide to a *PIDController*

kDisplacement = 0

kRate = 1

AnalogAccelerometer.**free**()

AnalogAccelerometer.**getAcceleration**()
Return the acceleration in Gs.

The acceleration is returned units of Gs.

Returns The current acceleration of the sensor in Gs.

Return type float

AnalogAccelerometer.**getPIDSourceType**()

AnalogAccelerometer.**pidGet**()
Get the Acceleration for the PID Source parent.

Returns The current acceleration in Gs.

Return type float

AnalogAccelerometer.**setPIDSourceType**(*pidSource*)
Set which parameter you are using as a process control variable.

Parameters **pidSource** (*PIDSource.PIDSourceType*) – An enum to select the parameter.

AnalogAccelerometer.**setSensitivity**(*sensitivity*)
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** (*float*) – The sensitivity of accelerometer in Volts per G.

AnalogAccelerometer.**setZero**(*zero*)
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** (*float*) – The zero G voltage.

AnalogGyro

class wpilib.**AnalogGyro**(*channel*, *center=None*, *offset=None*)
Bases: *wpilib.GyroBase*

Interface to a gyro device via an *AnalogInput*

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.

Gyro constructor.

Also initializes 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.

Parameters

- **channel** – The analog channel index or AnalogInput object that the gyro is connected to. Gyros can only be used on on-board channels 0-1.
- **center** (*int*) – Preset uncalibrated value to use as the accumulator center value
- **offset** (*float*) – Preset uncalibrated value to use as the gyro offset

class PIDSourceType

Bases: object

A description for the type of output value to provide to a *PIDController*

kDisplacement = 0

kRate = 1

AnalogGyro.**calibrate**()

See *Gyro.calibrate()*

AnalogGyro.**free**()

See *Gyro.free()*

AnalogGyro.**getAngle**()

See *Gyro.getAngle()*

AnalogGyro.**getCenter**()

Return the gyro center value set during calibration to use as a future preset

Returns the current center value

AnalogGyro.**getOffset**()

Return the gyro offset value set during calibration to use as a future preset

Returns the current offset value

AnalogGyro.**getRate**()

See *Gyro.getRate()*

AnalogGyro.**kAverageBits = 0**

AnalogGyro.**kCalibrationSampleTime = 5.0**

AnalogGyro.**kDefaultVoltsPerDegreePerSecond = 0.007**

AnalogGyro.**kOversampleBits = 10**

AnalogGyro.**kSamplesPerSecond = 50.0**

AnalogGyro.**reset**()

See *Gyro.reset()*

AnalogGyro.**setDeadband**(*volts*)

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** (*float*) – The size of the deadband in volts

AnalogGyro.**setSensitivity**(*voltsPerDegreePerSecond*)

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** (*float*) – The sensitivity in Volts/degree/second

AnalogInput

class wpilib.**AnalogInput**(*channel*)

Bases: *wpilib.SensorBase*

Analog input

Each analog channel is read from hardware as a 12-bit number representing 0V to 5V.

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 channel. :param channel: The channel number to represent. 0-3 are on-board 4-7 are on the MXP port.

class **PIDSourceType**

Bases: object

A description for the type of output value to provide to a *PIDController*

kDisplacement = 0

kRate = 1

AnalogInput.**channels = <wpilib.resource.Resource object>**

AnalogInput.**free**()

AnalogInput.**getAccumulatorCount**()

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.

AnalogInput.**getAccumulatorOutput**()

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.

Returns tuple of (value, count)

AnalogInput.**getAccumulatorValue**()

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().

AnalogInput.**getAverageBits**()

Get the number of averaging bits. 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 The number of averaging bits.

AnalogInput.**getAverageValue**()
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 this channel. Use *getAverageVoltage()* to get the analog value in calibrated units.

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

AnalogInput.**getAverageVoltage**()
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.

AnalogInput.**getChannel**()
Get the channel number.

Returns The channel number.

static AnalogInput.**getGlobalSampleRate**()
Get the current sample rate.

This assumes one entry in the scan list. This is a global setting for all channels.

Returns Sample rate.

AnalogInput.**getLSBWeight**()
Get the factory scaling least significant bit weight constant. The least significant bit weight constant for the channel that was calibrated in manufacturing and stored in an eeprom.

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

Returns Least significant bit weight.

AnalogInput.**getOffset**()
Get the factory scaling offset constant. The offset constant for the channel that was calibrated in manufacturing and stored in an eeprom.

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

Returns Offset constant.

AnalogInput.**getOversampleBits**()
Get the number of oversample bits. 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 The number of oversample bits.

AnalogInput.**getPIDSourceType**()

See *PIDSource.getPIDSourceType()*

AnalogInput.**getValue**()
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. 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.

AnalogInput.**getVoltage**()
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.

AnalogInput.**initAccumulator**()
Initialize the accumulator.

AnalogInput.**isAccumulatorChannel**()
Is the channel attached to an accumulator.

Returns The analog channel is attached to an accumulator.

AnalogInput.**kAccumulatorChannels = (0, 1)**

AnalogInput.**kAccumulatorSlot = 1**

AnalogInput.**pidGet**()
Get the average voltage for use with PIDController

Returns the average voltage

AnalogInput.**port**

AnalogInput.**resetAccumulator**()
Resets the accumulator to the initial value.

AnalogInput.**setAccumulatorCenter**(*center*)
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 make integration work and to take the device offset into account when integrating.

This center value is based on the output of the oversampled and averaged source from channel 1. Because of this, any non-zero oversample bits will affect the size of the value for this field.

AnalogInput.**setAccumulatorDeadband**(*deadband*)
Set the accumulator's deadband.

AnalogInput.**setAccumulatorInitialValue**(*initialValue*)
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.

AnalogInput.**setAverageBits**(*bits*)
Set the number of averaging bits. This sets the number of averaging bits. The actual number of averaged samples is 2^bits. The averaging is done automatically in the FPGA.

Parameters **bits** – The number of averaging bits.

static AnalogInput.**setGlobalSampleRate**(*samplesPerSecond*)
Set the sample rate per channel.

This is a global setting for all channels. The maximum rate is 500kS/s divided by the number of channels in use. This is 62500 samples/s per channel if all 8 channels are used.

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

AnalogInput.**setOversampleBits**(*bits*)
Set the number of oversample bits. This sets the number of oversample bits. The actual number of oversampled values is 2^bits. The oversampling is done automatically in the FPGA.

Parameters **bits** – The number of oversample bits.

AnalogInput.**setPIDSourceType**(*pidSource*)

See *PIDSource.setPIDSourceType()*

AnalogOutput

class wpilib.**AnalogOutput**(*channel*)
Bases: *wpilib.SensorBase*

Analog output

Construct an analog output on a specified MXP channel.

Parameters **channel** – The channel number to represent.

channels = <wpilib.resource.Resource object>

free()
Channel destructor.

getChannel()
Get the channel of this AnalogOutput.

getVoltage()

port

setVoltage(*voltage*)

AnalogPotentiometer

class wpilib.**AnalogPotentiometer**(*channel*, *fullRange=1.0*, *offset=0.0*)
Bases: *wpilib.LiveWindowSendable*

Reads a potentiometer via an *AnalogInput*

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.

AnalogPotentiometer constructor.

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.

Parameters

- **channel** (int or *AnalogInput*) – The analog channel this potentiometer is plugged into.
- **fullRange** (*float*) – The scaling to multiply the fraction by to get a meaningful unit. Defaults to 1.0 if unspecified.
- **offset** (*float*) – The offset to add to the scaled value for controlling the zero value. Defaults to 0.0 if unspecified.

class PIDSourceType
Bases: object

A description for the type of output value to provide to a *PIDController*

kDisplacement = 0

kRate = 1

AnalogPotentiometer.**free**()

AnalogPotentiometer.**get**()
Get the current reading of the potentiometer.

Returns The current position of the potentiometer.

Return type float

AnalogPotentiometer.**getPIDSourceType**()

AnalogPotentiometer.**pidGet**()
Implement the PIDSource interface.

Returns The current reading.

Return type float

AnalogPotentiometer.**setPIDSourceType**(*pidSource*)
Set which parameter you are using as a process control variable.

Parameters **pidSource** (*PIDSource.PIDSourceType*) – An enum to select the parameter.

AnalogTrigger

class wpilib.**AnalogTrigger**(*channel*)
Bases: object

Converts an analog signal into a digital signal

An analog trigger is a way to convert an analog signal into a digital signal using resources built into the FPGA. The resulting digital signal can then be used directly or fed into other digital components of the FPGA such as the counter or encoder modules. The analog trigger module works by comparing analog signals to a voltage range set by the code. The specific return types and meanings depend on the analog trigger mode in use.

Constructor for an analog trigger given a channel number or analog input.

Parameters **channel** – the port index or *AnalogInput* to use for the analog trigger. Treated as an AnalogInput if the provided object has a getChannel function.

class AnalogTriggerType
Bases: object

Defines the state in which the *AnalogTrigger* triggers

kFallingPulse = 3

kInWindow = 0

kRisingPulse = 2

kState = 1

AnalogTrigger.**createOutput**(*type*)
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 An AnalogTriggerOutput object.

AnalogTrigger.**free**()
Release the resources used by this object

AnalogTrigger.**getInWindow**()
Return the InWindow output of the analog trigger. True if the analog input is between the upper and lower limits.

Returns The InWindow output of the analog trigger.

AnalogTrigger.**getIndex**()
Return the index of the analog trigger. This is the FPGA index of this analog trigger instance.

Returns The index of the analog trigger.

AnalogTrigger.**getTriggerState**()
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 The TriggerState output of the analog trigger.

AnalogTrigger.**port**

AnalogTrigger.**setAveraged**(*useAveragedValue*)
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** – True to use an averaged value, False otherwise

AnalogTrigger.**setFiltered**(*useFilteredValue*)
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** – True to use a filterd value, False otherwise

AnalogTrigger.**setLimitsRaw**(*lower*, *upper*)
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 raw limit
- **upper** – the upper raw limit

AnalogTrigger.**setLimitsVoltage**(*lower*, *upper*)
Set the upper and lower limits of the analog trigger. The limits are given as floating point voltage values.

Parameters

- **lower** – the lower voltage limit
- **upper** – the upper voltage limit

AnalogTriggerOutput

class wpilib.**AnalogTriggerOutput**(*trigger*, *outputType*)
Bases: *wpilib.DigitalSource*

Represents a specific output from an *AnalogTrigger*

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 :class:.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 couter 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** – The trigger for which this is an output.
- **outputType** – An enum that specifies the output on the trigger to represent.

class AnalogTriggerType

Bases: object

Defines the state in which the *AnalogTrigger* triggers

kFallingPulse = 3

kInWindow = 0

kRisingPulse = 2

kState = 1

AnalogTriggerOutput.**free**()

AnalogTriggerOutput.**get**()

Get the state of the analog trigger output.

Returns The state of the analog trigger output.

Return type *AnalogTriggerType*

AnalogTriggerOutput.**getAnalogTriggerTypeForRouting**()

AnalogTriggerOutput.**getChannel**()

AnalogTriggerOutput.**getPortHandleForRouting**()

BuiltInAccelerometer

class wpilib.**BuiltInAccelerometer**(*range=2*)

Bases: *wpilib.LiveWindowSendable*

Built-in accelerometer device

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

Constructor.

Parameters **range** (*Accelerometer.Range*) – The range the accelerometer will measure. Defaults to +/-8g if unspecified.

class Range

Bases: object

k16G = 3

k2G = 0

k4G = 1

k8G = 2

BuiltInAccelerometer.**free**()

BuiltInAccelerometer.**getX**()

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

Return type float

BuiltInAccelerometer.**getY**()

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

Return type float

BuiltInAccelerometer.**getZ**()

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

Return type float

BuiltInAccelerometer.**setRange**(*range*)

Set the measuring range of the accelerometer.

Parameters **range** (*BuiltInAccelerometer.Range*) – The maximum acceleration, positive or negative, that the accelerometer will measure.

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()

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** – 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).

CANJaguar

class wpilib.**CANJaguar**(**args*, ***kwargs*)
Bases: object

CANTalon

class wpilib.**CANTalon**(**args*, ***kwargs*)
Bases: object

Compressor

class wpilib.**Compressor**(*module=None*)
Bases: *wpilib.SensorBase*

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 the 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.

Makes a new instance of the compressor using the provided CAN device ID.

Parameters **module** – The PCM CAN device ID. (0 - 62 inclusive)

clearAllPCMStickyFaults()
Clear ALL sticky faults inside PCM that Compressor is wired to.

If a sticky fault is set, then it will be persistently cleared. The compressor might momentarily disable while the flags are being cleared. Doo not call this method too frequently, otherwise normal compressor functionality may be prevented.

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

enabled()
Get the enabled status of the compressor.

Returns True if the compressor is on

Return type bool

getClosedLoopControl()
Gets the current operating mode of the PCM.

Returns True if compressor is operating on closed-loop mode

Return type bool

getCompressorCurrent()
Get the current being used by the compressor.

Returns Current consumed by the compressor in amps

Return type float

getCompressorCurrentTooHighFault()

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

getCompressorCurrentTooHighStickyFault()

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

getCompressorNotConnectedFault()

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

getCompressorNotConnectedStickyFault()

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

getCompressorShortedFault()

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

getCompressorShortedStickyFault()

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

getPressureSwitchValue()
Get the pressure switch value.

Returns True if the pressure is low

Return type bool

setClosedLoopControl(*on*)
Set the PCM in closed loop control mode.

Parameters **on** (*bool*) – If True sets the compressor to be in closed loop control mode (default)

start()
Start the compressor running in closed loop control mode.

Use the method in cases where you would like to manually stop and start the compressor for applications such as conserving battery or making sure that the compressor motor doesn't start during critical operations.

stop()
Stop the compressor from running in closed loop control mode.

Use the method in cases where you would like to manually stop and start the compressor for applications such as conserving battery or making sure that the compressor motor doesn't start during critical operations.

ControllerPower

class wpilib.**ControllerPower**
Bases: object

Provides access to power levels on the roboRIO

static getCurrent3V3()
Get the current output of the 3.3V rail

Returns The controller 3.3V rail output current value in Amps

Return type float

static getCurrent5V()
Get the current output of the 5V rail

Returns The controller 5V rail output current value in Amps

Return type float

static getCurrent6V()
Get the current output of the 6V rail

Returns The controller 6V rail output current value in Amps

Return type float

static getEnabled3V3()
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 True if enabled, False otherwise

Return type bool

static getEnabled5V()
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 True if enabled, False otherwise

Return type bool

static getEnabled6V()
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 True if enabled, False otherwise

Return type bool

static getFaultCount3V3()
Get the count of the total current faults on the 3.3V rail since the controller has booted

Returns The number of faults

Return type int

static getFaultCount5V()
Get the count of the total current faults on the 5V rail since the controller has booted

Returns The number of faults

Return type int

static getFaultCount6V()
Get the count of the total current faults on the 6V rail since the controller has booted

Returns The number of faults

Return type int

static getInputCurrent()
Get the input current to the robot controller

Returns The controller input current value in Amps

Return type float

static getInputVoltage()
Get the input voltage to the robot controller

Returns The controller input voltage value in Volts

Return type float

static getVoltage3V3()
Get the voltage of the 3.3V rail

Returns The controller 3.3V rail voltage value in Volts

Return type float

static getVoltage5V()
Get the voltage of the 5V rail

Returns The controller 5V rail voltage value in Volts

Return type float

static getVoltage6V()
Get the voltage of the 6V rail

Returns The controller 6V rail voltage value in Volts

Return type float

Counter

class wpilib.**Counter**(**args*, ***kwargs*)
Bases: *wpilib.SensorBase*

Counts the number of ticks on a *DigitalInput* 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.

Counter constructor.

The counter will start counting immediately.

Positional arguments may be either channel numbers, *DigitalSource* sources, or *AnalogTrigger* sources in the following order:

A "source" is any valid single-argument input to *setUpSource()* and *setDownSource()*

•(none)

•upSource

•upSource, down source

And, to keep consistency with Java wpilib. - encodingType, up source, down source, inverted

If the passed object has a *getPortHandleForRouting* function, it is assumed to be a DigitalSource. If the passed object has a *createOutput* function, it is assumed to be an AnalogTrigger.

In addition, extra keyword parameters may be provided for mode, inverted, and encodingType.

Parameters

- **upSource** – The source (channel num, DigitalInput, or AnalogTrigger) that should be used for up counting.
- **downSource** – The source (channel num, DigitalInput, or AnalogTrigger) that should be used for down counting or direction control.
- **mode** – How and what the counter counts (see *Mode*). Defaults to *Mode.kTwoPulse* for zero or one source, and *Mode.kExternalDirection* for two sources.
- **inverted** – Flips the direction of counting. Defaults to False if unspecified. Only used when two sources are specified.
- **encodingType** (*Counter.EncodingType*) – Either k1X or k2X to indicate 1X or 2X decoding. 4X decoding is not supported by Counter; use *Encoder* instead. Defaults to k1X if unspecified. Only used when two sources are specified.

class EncodingType
Bases: object

The number of edges for the counterbase to increment or decrement on

k1X = 0

k2X = 1

k4X = 2

class Counter.**Mode**
Bases: object

Mode determines how and what the counter counts

kExternalDirection = 3
external direction mode

kPulseLength = 2
pulse length mode

kSemiperiod = 1
semi period mode

kTwoPulse = 0
two pulse mode

class Counter.**PIDSourceType**
Bases: object

A description for the type of output value to provide to a *PIDController*

kDisplacement = 0

kRate = 1

Counter.**allocatedDownSource = False**

Counter.**allocatedUpSource = False**

Counter.**clearDownSource**()
Disable the down counting source to the counter.

Counter.**clearUpSource**()
Disable the up counting source to the counter.

Counter.**counter**

Counter.**free**()

Counter.**get**()
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.

Counter.**getDirection**()
The last direction the counter value changed.

Returns The last direction the counter value changed.

Return type bool

Counter.**getDistance**()
Read the current scaled counter value. Read the value at this instant, scaled by the distance per pulse (defaults to 1).

Returns Scaled value

Return type float

Counter.**getFPGAIndex**()

Returns The Counter's FPGA index.

Counter.**getPIDSourceType**()

Counter.**getPeriod**()
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 of the last two pulses in units of seconds.

Return type float

Counter.**getRate**()
Get the current rate of the Counter. Read the current rate of the counter accounting for the distance per pulse value. The default value for distance per pulse (1) yields units of pulses per second.

Returns The rate in units/sec

Return type float

Counter.**getSamplesToAverage**()
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)

Return type int

Counter.**getStopped**()
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.

Return type bool

Counter.**pidGet**()

Counter.**reset**()
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.

Counter.**setDistancePerPulse**(*distancePerPulse*)
Set the distance per pulse for this counter. This sets the multiplier used to determine the distance driven based on the count value from the encoder. Set this value based on the Pulses per Revolution and factor in any gearing reductions. This distance can be in any units you like, linear or angular.

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

Counter.**setDownSource**(**args*, ***kwargs*)
Set the down counting source for the counter.

This function accepts either a digital channel index, a *DigitalSource*, or an *AnalogTrigger* as positional arguments:

•source

•channel

•analogTrigger

•analogTrigger, triggerType

For positional arguments, if the passed object has a *getChannelForRouting* function, it is assumed to be a DigitalSource. If the passed object has a *createOutput* function, it is assumed to be an AnalogTrigger.

Alternatively, sources and/or channels may be passed as keyword arguments. The behavior of specifying both a source and a number for the same channel is undefined, as is passing both a positional and a keyword argument for the same channel.

Parameters

- **channel** (*int*) – the DIO channel to use as the down source. 0-9 are on-board, 10-25 are on the MXP
- **source** (DigitalInput) – The digital source to count
- **analogTrigger** (AnalogTrigger) – The analog trigger object that is used for the Up Source
- **triggerType** (AnalogTriggerType) – The analog trigger output that will trigger the counter. Defaults to kState if not specified.

Counter.**setDownSourceEdge**(*risingEdge*, *fallingEdge*)
Set the edge sensitivity on an down counting source. Set the down source to either detect rising edges or falling edges.

Parameters

- **risingEdge** (*bool*) – True to count rising edge
- **fallingEdge** (*bool*) – True to count falling edge

Counter.**setExternalDirectionMode**()
Set external direction mode on this counter. Counts are sourced on the Up counter input. The Down counter input represents the direction to count.

Counter.**setMaxPeriod**(*maxPeriod*)
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** (*float or int*) – The maximum period where the counted device is considered moving in seconds.

Counter.**setPIDSourceType**(*pidSource*)
Set which parameter of the encoder you are using as a process control variable. The counter class supports the rate and distance parameters.

Parameters **pidSource** (*Counter.PIDSourceType*) – An enum to select the parameter.

Counter.**setPulseLengthMode**(*threshold*)
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** (*float, int*) – The pulse length beyond which the counter counts the opposite direction. Units are seconds.

Counter.**setReverseDirection**(*reverseDirection*)
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** (*bool*) – True if the value counted should be negated.

Counter.**setSamplesToAverage**(*samplesToAverage*)
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** (*int*) – The number of samples to average from 1 to 127.

Counter.**setSemiPeriodMode**(*highSemiPeriod*)
Set Semi-period mode on this counter. Counts up on both rising and falling edges.

Parameters **highSemiPeriod** (*bool*) – True to count up on both rising and falling

Counter.**setUpDownCounterMode**()
Set standard up / down counting mode on this counter. Up and down counts are sourced independently from two inputs.

Counter.**setUpSource**(**args*, ***kwargs*)
Set the up counting source for the counter.

This function accepts either a digital channel index, a *DigitalSource*, or an *AnalogTrigger* as positional arguments:

•source

•channel

•analogTrigger

•analogTrigger, triggerType

For positional arguments, if the passed object has a *getPortHandleForRouting* function, it is assumed to be a DigitalSource. If the passed object has a *createOutput* function, it is assumed to be an AnalogTrigger.

Alternatively, sources and/or channels may be passed as keyword arguments. The behavior of specifying both a source and a number for the same channel is undefined, as is passing both a positional and a keyword argument for the same channel.

Parameters

- **channel** (*int*) – the DIO channel to use as the up source. 0-9 are on-board, 10-25 are on the MXP
- **source** (DigitalInput) – The digital source to count
- **analogTrigger** (AnalogTrigger) – The analog trigger object that is used for the Up Source
- **triggerType** (AnalogTriggerType) – The analog trigger output that will trigger the counter. Defaults to kState if not specified.

Counter.**setUpSourceEdge**(*risingEdge*, *fallingEdge*)

Set the edge sensitivity on an up counting source. Set the up source to either detect rising edges or falling edges.

Parameters

- **risingEdge** (*bool*) – True to count rising edge
- **fallingEdge** (*bool*) – True to count falling edge

Counter.**setUpdateWhenEmpty**(*enabled*)

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 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** (*bool*) – True to continue updating

DigitalGlitchFilter

class wpilib.**DigitalGlitchFilter**

Bases: *wpilib.SensorBase*

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(*input*)

Assigns the *DigitalSource*, *Encoder*, or *Counter* to this glitch filter.

Parameters **input** – The object to add

filterAllocated = [False, False, False]

free()

getPeriodCycles()

Gets the number of FPGA cycles that the input must hold steady to pass through this glitch filter.

Returns The number of cycles.

getPeriodNanoSeconds()
Gets the number of nanoseconds that the input must hold steady to pass through this glitch filter.

Returns The number of nanoseconds.

mutex = <unlocked _thread.lock object>

remove(*input*)
Removes this filter from the given input object

setPeriodCycles(*fpga_cycles*)
Sets the number of FPGA cycles that the input must hold steady to pass through this glitch filter.

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

setPeriodNanoSeconds(*nanoseconds*)
Sets the number of nanoseconds that the input must hold steady to pass through this glitch filter.

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

DigitalInput

class wpilib.**DigitalInput**(*channel*)
Bases: *wpilib.DigitalSource*

Reads 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** (*int*) – the DIO channel for the digital input. 0-9 are on-board, 10-25 are on the MXP

free()

get()
Get the value from a digital input channel. Retrieve the value of a single digital input channel from the FPGA.

Returns the state of the digital input

Return type bool

getAnalogTriggerTypeForRouting()
Get the analog trigger type.

Returns false

Return type int

getChannel()
Get the channel of the digital input.

Returns The GPIO channel number that this object represents.

Return type int

getPortHandleForRouting()
Get the HAL Port Handle.

Returns The HAL Handle to the specified source

isAnalogTrigger()
Is this an analog trigger.

Returns true if this is an analog trigger

Return type bool

DigitalOutput

class wpilib.**DigitalOutput**(*channel*)
Bases: *wpilib.DigitalSource*

Writes to a digital output

Other devices that are implemented elsewhere will automatically allocate digital inputs and outputs as required.

Create an instance of a digital output.

Parameters **channel** – the DIO channel for the digital output. 0-9 are on-board, 10-25 are on the MXP

disablePWM()
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*)
Enable a PWM Output on this line.

Allocate one of the 6 DO PWM generator resources.

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** (*float*) – The duty-cycle to start generating. [0..1]

free()
Free the resources associated with a digital output.

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

Returns the state of the digital output

Return type bool

getAnalogTriggerTypeForRouting()
Get the analog trigger type.

Returns false

Return type int

getChannel()

Returns The GPIO channel number that this object represents.

getPortHandleForRouting()
Get the HAL Port Handle.

Returns The HAL Handle to the specified source

isAnalogTrigger()
Is this an analog trigger.

Returns true if this is an analog trigger

Return type bool

isPulsing()
Determine if the pulse is still going. Determine if a previously started pulse is still going.

Returns True if pulsing

Return type bool

pulse(*pulseLength*, **args*)
Generate a single pulse. There can only be a single pulse going at any time.

Parameters

- **channel** – Unused. Deprecated 2017.1.1.
- **pulseLength** (*float*) – The length of the pulse.

pwmGenerator

set(*value*)
Set the value of a digital output.

Parameters **value** (*bool*) – True is on, off is False

setPWMRate(*rate*)
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 channnels.

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

updateDutyCycle(*dutyCycle*)
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** (*float*) – The duty-cycle to change to. [0..1]

DigitalSource

class wpilib.**DigitalSource**(*channel*, *input*)
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.

Parameters

- **channel** (*int*) – Port for the digital input
- **input** (*int*) – True if input, False otherwise

channels = <wpilib.resource.Resource object>

free()

getChannel()

handle

isAnalogTrigger()

DoubleSolenoid

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

Controls 2 channels of high voltage Digital Output on the PCM.

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

Constructor.

Arguments can be supplied as positional or keyword. Acceptable positional argument combinations are:

•forwardChannel, reverseChannel

•moduleNumber, forwardChannel, reverseChannel

Alternatively, the above names can be used as keyword arguments.

Parameters

- **moduleNumber** – The module number of the solenoid module to use.
- **forwardChannel** – The forward channel number on the module to control (0..7)
- **reverseChannel** – The reverse channel number on the module to control (0..7)

class Value
Bases: object

Possible values for a DoubleSolenoid.

kForward = 1

kOff = 0

kReverse = 2

DoubleSolenoid.**free**()
Mark the solenoid as freed.

DoubleSolenoid.**get**()
Read the current value of the solenoid.

Returns The current value of the solenoid.

Return type *DoubleSolenoid.Value*

DoubleSolenoid.**isFwdSolenoidBlackListed**()

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.

DoubleSolenoid.**isRevSolenoidBlackListed**()

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.

DoubleSolenoid.**set**(*value*)
Set the value of a solenoid.

Parameters **value** (*DoubleSolenoid.Value*) – The value to set (Off, Forward, Reverse)

DriverStation

class wpilib.**DriverStation**
Bases: object

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

DriverStation constructor.

The single DriverStation instance is created statically with the instance static member variable, you should never create a DriverStation instance.

class **Alliance**
Bases: object

The robot alliance that the robot is a part of

Blue = 1

Invalid = 2

Red = 0

DriverStation.**InAutonomous**(*entering*)
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

DriverStation.**InDisabled**(*entering*)
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

DriverStation.**InOperatorControl**(*entering*)
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

DriverStation.**InTest**(*entering*)
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

DriverStation.**getAlliance**()
Get the current alliance from the FMS.

Returns The current alliance

Return type *DriverStation.Alliance*

DriverStation.**getBatteryVoltage**()
Read the battery voltage.

Returns The battery voltage in Volts.

classmethod DriverStation.**getInstance**()
Gets the global instance of the DriverStation

Returns *DriverStation*

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

Parameters

- **stick** (*int*) – The joystick port number
- **axis** (*int*) – The target axis

:returns An integer that reports type of axis the axis is reporting to be

DriverStation.**getJoystickIsXbox**(*stick*)
Gets the value of isXbox on a joystick

Parameters **stick** (*int*) – The joystick port number

:returns A boolean that returns the value of isXbox

DriverStation.**getJoystickName**(*stick*)
Gets the name of a joystick

Parameters **stick** (*int*) – The joystick port number

:returns The joystick name.

DriverStation.**getJoystickType**(*stick*)
Gets the value of type on a joystick

Parameters **stick** (*int*) – The joystick port number

:returns An integer that returns the value of type.

DriverStation.**getLocation**()
Gets the location of the team's driver station controls.

Returns The location of the team's driver station controls: 1, 2, or 3

DriverStation.**getMatchTime**()
Return the approximate match time. The FMS does not currently send the 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 argue with referees or guarantee that a function will trigger before a 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) in seconds

DriverStation.**getStickAxis**(*stick*, *axis*)
Get the value of the axis on a joystick. This depends on the mapping of the joystick connected to the specified port.

Parameters

- **stick** (*int*) – The joystick port number
- **axis** (*int*) – The analog axis value to read from the joystick.

Returns The value of the axis on the joystick.

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

Parameters **stick** (*int*) – The joystick port number

Returns The number of axes on the indicated joystick

DriverStation.**getStickButton**(*stick*, *button*)
The state of a button on the joystick. Button indexes begin at 1.

Parameters

- **stick** (*int*) – The joystick port number
- **button** (*int*) – The button index, beginning at 1.

Returns The state of the button.

DriverStation.**getStickButtonCount**(*stick*)
Gets the number of buttons on a joystick

Parameters **stick** (*int*) – The joystick port number

Returns The number of buttons on the indicated joystick.

DriverStation.**getStickButtons**(*stick*)
The state of all the buttons on the joystick.

Parameters **stick** (*int*) – The joystick port number

Returns The state of all buttons, as a bit array.

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

Parameters

- **stick** (*int*) – The joystick port number
- **pov** (*int*) – which POV

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

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

Parameters **stick** (*int*) – The joystick port number

Returns The number of POVs on the indicated joystick

DriverStation.**isAutonomous**()
Gets a value indicating whether the Driver Station requires the robot to be running in autonomous mode.

Returns True if autonomous mode should be enabled, False otherwise.

DriverStation.**isBrownedOut**()
Check if the system is browned out.

Returns True if the system is browned out.

DriverStation.**isDSAttached**()
Is the driver station attached to the robot?

Returns True if the robot is being controlled by a driver station.

DriverStation.**isDisabled**()
Gets a value indicating whether the Driver Station requires the robot to be disabled.

Returns True if the robot should be disabled, False otherwise.

DriverStation.**isEnabled**()
Gets a value indicating whether the Driver Station requires the robot to be enabled.

Returns True if the robot is enabled, False otherwise.

DriverStation.**isFMSAttached**()
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

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

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

DriverStation.**isOperatorControl**()
Gets a value indicating whether the Driver Station requires the robot to be running in operator-controlled mode.

Returns True if operator-controlled mode should be enabled, False otherwise.

DriverStation.**isSysActive**()
Gets a value indicating whether the FPGA outputs are enabled. The outputs may be disabled if the robot is disabled or e-stopped, the watdhog has expired, or if the roboRIO browns out.

Returns True if the FPGA outputs are enabled.

DriverStation.**isTest**()
Gets a value indicating whether the Driver Station requires the robot to be running in test mode.

Returns True if test mode should be enabled, False otherwise.

DriverStation.**kJoystickPorts = 6**
The number of joystick ports

DriverStation.**release**()
Kill the thread

static DriverStation.**reportError**(*error*, *printTrace*)
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

static DriverStation.**reportWarning**(*error*, *printTrace*)
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 warning string

DriverStation.**waitForData**(*timeout=None*)
Wait for new data or for timeout, which ever comes first. If timeout is None, wait for new data only.

Parameters **timeout** – The maximum time in seconds to wait.

Returns True if there is new data, otherwise False

Encoder

class wpilib.**Encoder**(**args*, ***kwargs*)
Bases: *wpilib.SensorBase*

Class to read quadrature encoders.

Quadrature encoders are devices that count shaft rotation and can sense direction. The output of the Encoder class is an integer that can count either up or down, and can go negative for reverse direction counting. When creating Encoders, a direction can be supplied that inverts 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.

Instance variables:

•aSource: The A phase of the quad encoder

•bSource: The B phase of the quad encoder

•indexSource: The index source (available on some encoders)

Encoder constructor. Construct a Encoder given a and b channels and optionally an index channel.

The encoder will start counting immediately.

The a, b, and optional index channel arguments may be either channel numbers or *DigitalSource* sources. There may also be a boolean reverseDirection, and an encodingType according to the following list.

•aSource, bSource

•aSource, bSource, reverseDirection

•aSource, bSource, reverseDirection, encodingType

•aSource, bSource, indexSource, reverseDirection

•aSource, bSource, indexSource

•aChannel, bChannel

•aChannel, bChannel, reverseDirection

•aChannel, bChannel, reverseDirection, encodingType

•aChannel, bChannel, indexChannel, reverseDirection

•aChannel, bChannel, indexChannel

For positional arguments, if the passed object has a *getPortHandleForRouting* function, it is assumed to be a DigitalSource.

Alternatively, sources and/or channels may be passed as keyword arguments. The behavior of specifying both a source and a number for the same channel is undefined, as is passing both a positional and a keyword argument for the same channel.

In addition, keyword parameters may be provided for reverseDirection and inputType.

Parameters

- **`aSource`** – The source that should be used for the a channel.
- **`bSource`** – The source that should be used for the b channel.
- **`indexSource`** – The source that should be used for the index channel.
- **`aChannel`** – The digital input index that should be used for the a channel.
- **`bChannel`** – The digital input index that should be used for the b channel.
- **`indexChannel`** – The digital input index that should be used for the index channel.

- **reverseDirection** – Represents the orientation of the encoder and inverts the output values if necessary so forward represents positive values. Defaults to False if unspecified.
- **encodingType** (*Encoder.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. Defaults to k4X if unspecified.

class EncodingType
Bases: object

The number of edges for the counterbase to increment or decrement on

k1X = 0

k2X = 1

k4X = 2

class Encoder.**IndexingType**
Bases: object

kResetOnFallingEdge = 2

kResetOnRisingEdge = 3

kResetWhileHigh = 0

kResetWhileLow = 1

class Encoder.**PIDSourceType**
Bases: object

A description for the type of output value to provide to a *PIDController*

kDisplacement = 0

kRate = 1

Encoder.**encoder**

Encoder.**free**()

Encoder.**get**()
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.

Encoder.**getDirection**()
The last direction the encoder value changed.

Returns The last direction the encoder value changed.

Encoder.**getDistance**()
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()*.

Encoder.**getEncodingScale**()

Returns The encoding scale factor 1x, 2x, or 4x, per the requested encodingType. Used to divide raw edge counts down to spec'd counts.

Encoder.**getFPGAIndex**()

Returns The Encoder's FPGA index

Encoder.**getPIDSourceType**()

Encoder.**getPeriod**()
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.

Deprecated since version Use: *getRate()* in favor of this method. This returns unscaled periods and *getRate()* scales using value from getDistancePerPulse().

Returns Period in seconds of the most recent pulse.

Encoder.**getRate**()
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.

Encoder.**getRaw**()
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

Encoder.**getSamplesToAverage**()
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)

Encoder.**getStopped**()
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.

Encoder.**pidGet**()
Implement the PIDSource interface.

Returns The current value of the selected source parameter.

Encoder.**reset**()
Reset the Encoder distance to zero. Resets the current count to zero on the encoder.

Encoder.**setDistancePerPulse**(*distancePerPulse*)
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.

Encoder.**setIndexSource**(*source*, *indexing_type=3*)
Set the index source for the encoder. When this source rises, the encoder count automatically resets.

Parameters

- **source** – Either an initialized DigitalSource or a DIO channel number
- **indexing_type** – The state that will cause the encoder to reset

Type Either a `DigitalInput` or number

Type A value from `wpilib.IndexingType`

`Encoder.`**setMaxPeriod**(*maxPeriod*)

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.

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

`Encoder.`**setMinRate**(*minRate*)

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()*.

`Encoder.`**setPIDSourceType**(*pidSource*)

Set which parameter of the encoder you are using as a process control variable. The encoder class supports the rate and distance parameters.

Parameters **pidSource** – An enum to select the parameter.

`Encoder.`**setReverseDirection**(*reverseDirection*)

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

`Encoder.`**setSamplesToAverage**(*samplesToAverage*)

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.

TODO: Should this raise an exception, so that the user has to deal with giving an incorrect value?

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

Filter

class `wpilib.`**Filter**(*source*)

Bases: `object`

Superclass for filters

Constructor.

Parameters **source** (*PIDSource*, callable) –

get()

Returns the current filter estimate without also inserting new data as *pidGet()* would do.

Returns The current filter estimate

getPIDSourceType()

pidGet()

pidGetSource()
Calls PIDGet() of source

Returns Current value of source

reset()
Reset the filter state

setPIDSourceType(*pidSourceType*)

GearTooth

class wpilib.**GearTooth**(*channel*, *directionSensitive=False*)
Bases: *wpilib.Counter*

Interface to the gear tooth sensor supplied by FIRST

Currently there is no reverse sensing on the gear tooth sensor, but in future versions we might implement the necessary timing in the FPGA to sense direction.

Construct a GearTooth sensor.

Parameters

- **channel** (*int*) – The DIO channel index or DigitalSource that the sensor is connected to.
- **directionSensitive** (*bool*) – True to enable the pulse length decoding in hardware to specify count direction. Defaults to False.

enableDirectionSensing(*directionSensitive*)

free()

kGearToothThreshold = 5.5e-05

GyroBase

class wpilib.**GyroBase**
Bases: *wpilib.SensorBase*

GyroBase is the common base class for Gyro implementations such as *AnalogGyro*.

class PIDSourceType
Bases: object

A description for the type of output value to provide to a *PIDController*

kDisplacement = 0

kRate = 1

GyroBase.**calibrate**()

GyroBase.**getAngle**()

GyroBase.**getPIDSourceType**()

GyroBase.**getRate**()

GyroBase.**pidGet**()
Get the output of the gyro for use with PIDControllers. May be the angle or rate depending on the set *PIDSourceType*

Returns the current angle according to the gyro

Return type float

GyroBase.**reset**()

GyroBase.**setPIDSourceType**(*pidSource*)
Set which parameter of the gyro you are using as a process control variable. The Gyro class supports the rate and angle parameters.

Parameters **pidSource** (*PIDSource.PIDSourceType*) – An enum to select the parameter.

I2C

class wpilib.**I2C**(*port*, *deviceAddress*, *simPort=None*)
Bases: object

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.

Example usage:

```
i2c = wpilib.I2C(wpilib.I2C.Port.kOnboard, 4)

# Write bytes 'text', and receive 4 bytes in data
data = i2c.transaction(b'text', 4)
```

Constructor.

Parameters

- **port** (*I2C.Port*) – The I2C port the device is connected to.
- **deviceAddress** – The address of the device on the I2C bus.
- **simPort** – This must be an object that implements all of the i2c* functions from hal_impl that you use. See test_i2c.py for an example.

class **Port**
Bases: object

kMXP = 1

kOnboard = 0

I2C.**addressOnly**()
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.

I2C.**free**()

I2C.**port**

I2C.**read**(*registerAddress*, *count*)
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.

Returns The data read from the device.

Return type iterable of bytes

I2C.**readOnly**(*count*)
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 **count** – The number of bytes to read in the transaction.

Returns The data read from the device.

Return type iterable of bytes

I2C.**transaction**(*dataToSend*, *receiveSize*)
Generic transaction.

This is a lower-level interface to the I2C hardware giving you more control over each transaction.

Parameters

- **dataToSend** (*iterable of bytes*) – Buffer of data to send as part of the transaction.
- **receiveSize** (*int*) – Number of bytes to read from the device.

Returns Data received from the device.

Return type iterable of bytes

I2C.**verifySensor**(*registerAddress*, *expected*)
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.

The device must support and be configured to use register auto-increment.

Parameters

- **registerAddress** – The base register to start reading from the device.
- **expected** – The values expected from the device.

Returns True if the sensor was verified to be connected

I2C.**write**(*registerAddress*, *data*)
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.

I2C.**writeBulk**(*data*)
Execute a write transaction with the device.

Write multiple bytes to a register on a device and wait until the transaction is complete.

Parameters **data** (*iterable of bytes*) – The data to write to the device.

Returns Transfer Aborted... False for success, True for aborted.

Usage:

```
# send byte string
failed = spi.writeBulk(b'stuff')

# send list of integers
failed = spi.write([0x01, 0x02])
```

GamepadBase

class wpilib.interfaces.**GamepadBase**(*port*)

Bases: *wpilib.interfaces.GenericHID*

GamepadBase Interface.

getBumper(*hand*)

Is the bumper pressed.

Parameters **hand** – which hand

Returns true if the bumper is pressed

getName()

getPOV(*pov=0*)

getPOVCount()

getRawAxis(*axis*)

getRawButton(*button*)

getStickButton(*hand=None*)

getType()

setOutput(*outputNumber*, *value*)

setOutputs(*value*)

setRumble(*type*, *value*)

GenericHID

class wpilib.interfaces.**GenericHID**(*port*)

Bases: object

GenericHID Interface.

class HIDType(*value*)

Bases: object

kHID1stPerson = 24

kHIDDriving = 22

kHIDFlight = 23

kHIDGamepad = 21

kHIDJoystick = 20

kUnknown = -1

kXInputArcadePad = 19

kXInputArcadeStick = 3

kXInputDancePad = 5

kXInputDrumKit = 8

kXInputFlightStick = 4

kXInputGamepad = 1

kXInputGuitar = 6

kXInputGuitar2 = 7

kXInputGuitar3 = 11

kXInputUnknown = 0

kXInputWheel = 2

class GenericHID.**Hand**
Bases: object

Which hand the Human Interface Device is associated with.

kLeft = 0
Left Hand

kRight = 1
Right Hand

class GenericHID.**RumbleType**
Bases: object

Represents a rumble output on the JoyStick.

kLeftRumble = 0
Left Hand

kRightRumble = 1
Right Hand

GenericHID.**getName**()
Get the name of the HID.

Returns the name of the HID.

GenericHID.**getPOV**(*pov=0*)
Get the angle in degrees of a POV on the HID.

The POV angles start at 0 in the up direction, and increase clockwise (eg 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.

GenericHID.**getPOVCount**()
For the current HID, return the number of POVs.

GenericHID.**getPort**()
Get the port number of the HID.

Returns The port number of the HID.

GenericHID.**getRawAxis**(*which*)
Get the raw axis.

Parameters **which** – index of the axis

Returns the raw value of the selected axis

GenericHID.**getRawButton**(*button*)
Is the given button pressed.

Parameters **button** – which button number

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

GenericHID.**getType**()
Get the type of the HID.

Returns the type of the HID.

GenericHID.**getX**(*hand=None*)
Get the x position of HID.

Parameters **hand** – which hand, left or right

Returns the x position

GenericHID.**getY**(*hand=None*)
Get the y position of the HID.

Parameters **hand** – which hand, left or right

Returns the y position

GenericHID.**setOutput**(*outputNumber*, *value*)
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

GenericHID.**setOutputs**(*value*)
Set all HID output values for the HID.

Parameters **value** – The 32 bit output value (1 bit for each output)

GenericHID.**setRumble**(*type*, *value*)
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

InterruptableSensorBase

class wpilib.**InterruptableSensorBase**
Bases: *wpilib.SensorBase*

Base for sensors to be used with interrupts

Create a new InterrupatableSensorBase

allocateInterrupts(*watcher*)
Allocate the interrupt

Parameters **watcher** – True if the interrupt should be in synchronous mode where the user program will have to explicitly wait for the interrupt to occur.

cancelInterrupts()
Cancel interrupts on this device. This deallocates all the chipobject structures and disables any interrupts.

disableInterrupts()
Disable Interrupts without without deallocating structures.

enableInterrupts()
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()

getPortHandleForRouting()

interrupt

readFallingTimestamp()
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 setUpSourceEdge.

Returns Timestamp in seconds since boot.

readRisingTimestamp()
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 setUpSourceEdge.

Returns Timestamp in seconds since boot.

requestInterrupts(*handler=None*)
Request one of the 8 interrupts asynchronously on this digital input.

Parameters **handler** – (optional) The function that will be called whenever there is an interrupt on this device. Request interrupts in synchronous mode where the user program interrupt handler will be called when an interrupt occurs. The default is interrupt on rising edges only. If not specified, the user program will have to explicitly wait for the interrupt to occur using waitForInterrupt.

setUpSourceEdge(*risingEdge*, *fallingEdge*)
Set which edge to trigger interrupts on

Parameters

- **risingEdge** – True to interrupt on rising edge
- **fallingEdge** – True to interrupt on falling edge

waitForInterrupt(*timeout*, *ignorePrevious=True*)
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 (default), ignore interrupts that happened before waitForInterrupt was called.

IterativeRobot

class wpilib.**IterativeRobot**
Bases: *wpilib.RobotBase*

IterativeRobot implements a specific type of Robot Program framework, extending the *RobotBase* class.

The IterativeRobot class is intended to be subclassed by a user creating a robot program.

This class is intended to implement the "old style" default code, by providing 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 only when first disabled
- *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 mode is entered from another mode

Periodic() functions – each of these functions is called iteratively at the appropriate periodic rate (aka the "slow loop"). The period of the iterative robot is synced to the driver station control packets, giving a periodic frequency of about 50Hz (50 times per second).

- *disabledPeriodic()*
- *autonomousPeriodic()*
- *teleopPeriodic()*
- *testPeriodic()*

Constructor for RobotIterativeBase.

The constructor initializes the instance variables for the robot to indicate the status of initialization for disabled, autonomous, and teleop code.

> **Warning:** If you override `__init__` in your robot class, you must call the base class constructor. This must be used to ensure that the communications code starts.

autonomousInit()
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()
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.

Packets are received approximately every 20ms. Fixed loop timing is not guaranteed due to network timing variability and the function may not be called at all if the Driver Station is disconnected. For most use cases the variable timing will not be an issue. If your code does require guaranteed fixed periodic timing, consider using Notifier or PIDController instead.

disabledInit()
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()
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.

Packets are received approximately every 20ms. Fixed loop timing is not guaranteed due to network timing variability and the function may not be called at all if the Driver Station is disconnected. For most use cases the variable timing will not be an issue. If your code does require guaranteed fixed periodic timing, consider using Notifier or PIDController instead.

logger = <logging.Logger object>
A python logging object that you can use to send messages to the log. It is recommended to use this instead of print statements.

robotInit()
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 1 time.

Note: It is simpler to override this function instead of defining a constructor for your robot class

---

robotPeriodic()
Periodic code for all robot modes should go here.

This function is called each time a new packet is received from the driver station.

Packets are received approximately every 20ms. Fixed loop timing is not guaranteed due to network timing variability and the function may not be called at all if the Driver Station is disconnected. For most use cases the variable timing will not be an issue. If your code does require guaranteed fixed periodic timing, consider using Notifier or PIDController instead.

startCompetition()
Provide an alternate "main loop" via startCompetition().

teleopInit()
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()
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.

Packets are received approximately every 20ms. Fixed loop timing is not guaranteed due to network timing variability and the function may not be called at all if the Driver Station is disconnected. For most use cases the variable timing will not be an issue. If your code does require guaranteed fixed periodic timing, consider using Notifier or PIDController instead.

testInit()
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()
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.

Packets are received approximately every 20ms. Fixed loop timing is not guaranteed due to network timing variability and the function may not be called at all if the Driver Station is disconnected. For most use cases the variable timing will not be an issue. If your code does require guaranteed fixed periodic timing, consider using Notifier or PIDController instead.

Jaguar

class wpilib.**Jaguar**(*channel*)
Bases: *wpilib.PWMSpeedController*

Texas Instruments / Vex Robotics Jaguar Speed Controller as a PWM device.

Constructor.

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

Joystick

class wpilib.**Joystick**(*port*, *numAxisTypes=None*, *numButtonTypes=None*)
Bases: wpilib.interfaces.JoystickBase

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.

This constructor is intended for use by subclasses to configure the number of constants for axes and buttons.

Parameters

- **port** (*int*) – The port on the Driver Station that the joystick is plugged into.
- **numAxisTypes** (*int*) – The number of axis types.
- **numButtonTypes** (*int*) – The number of button types.

class AxisType
Bases: object

Represents an analog axis on a joystick.

kNumAxis = 5

kThrottle = 4

kTwist = 3

kX = 0

kY = 1

kZ = 2

class Joystick.**ButtonType**
Bases: object

Represents a digital button on the Joystick

kNumButton = 2

kTop = 1

kTrigger = 0

Joystick.**flush_outputs**()
Flush all joystick HID & rumble output values to the HAL

Joystick.**getAxis**(*axis*)
For the current joystick, return the axis determined by the argument.

This is for cases where the joystick axis is returned programmatically, otherwise one of the previous functions would be preferable (for example *getX()*).

Parameters **axis** (*Joystick.AxisType*) – The axis to read.

Returns The value of the axis.

Return type float

Joystick.**getAxisChannel**(*axis*)
Get the channel currently associated with the specified axis.

Parameters **axis** (*int*) – The axis to look up the channel for.

Returns The channel for the axis.

Return type int

Joystick.**getAxisCount**()
For the current joystick, return the number of axis

Joystick.**getAxisType**(*axis*)
Get the axis type of a joystick axis.

Returns the axis type of a joystick axis.

Joystick.**getBumper**(*hand=None*)
This is not supported for the Joystick.

This method is only here to complete the GenericHID interface.

Parameters **hand** – This parameter is ignored for the Joystick class and is only here to complete the GenericHID interface.

Returns The state of the bumper (always False)

Return type bool

Joystick.**getButton**(*button*)
Get buttons based on an enumerated type.

The button type will be looked up in the list of buttons and then read.

Parameters **button** (*Joystick.ButtonType*) – The type of button to read.

Returns The state of the button.

Return type bool

Joystick.**getButtonCount**()
For the current joystick, return the number of buttons

:rtype int

Joystick.**getDirectionDegrees**()
Get the direction of the vector formed by the joystick and its origin in degrees.

Returns The direction of the vector in degrees

Return type float

Joystick.**getDirectionRadians**()
Get the direction of the vector formed by the joystick and its origin in radians.

Returns The direction of the vector in radians

Return type float

Joystick.**getIsXbox**()
Get the value of isXbox for the current joystick.

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

Joystick.**getMagnitude**()
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

Return type float

Joystick.**getName**()
Get the name of the HID.

Returns The name of the HID.

Joystick.**getPOV**(*pov=0*)

Joystick.**getPOVCount**()

Joystick.**getRawAxis**(*axis*)
Get the value of the axis.

Parameters **axis** (*int*) – The axis to read, starting at 0.

Returns The value of the axis.

Return type float

Joystick.**getRawButton**(*button*)
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** (*int*) – The button number to be read (starting at 1).

Returns The state of the button.

Return type bool

Joystick.**getThrottle**()
Get the throttle value of the current joystick.

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

Returns The Throttle value of the joystick.

Return type float

Joystick.**getTop**(*hand=None*)
Read the state of the top button on the joystick.

Look up which button has been assigned to the top and read its state.

Parameters **hand** – This parameter is ignored for the Joystick class and is only here to complete the GenericHID interface.

Returns The state of the top button.

Return type bool

Joystick.**getTrigger**(*hand=None*)
Read the state of the trigger on the joystick.

Look up which button has been assigned to the trigger and read its state.

Parameters **hand** – This parameter is ignored for the Joystick class and is only here to complete the GenericHID interface.

Returns The state of the trigger.

Return type bool

Joystick.**getTwist**()
Get the twist value of the current joystick.

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

Returns The Twist value of the joystick.

Return type float

Joystick.**getType**()
Get the type of the HID.

Returns the type of the HID.

Joystick.**getX**(*hand=None*)
Get the X value of the joystick.

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

Parameters **hand** – Unused

Returns The X value of the joystick.

Return type float

Joystick.**getY**(*hand=None*)
Get the Y value of the joystick.

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

Parameters **hand** – Unused

Returns The Y value of the joystick.

Return type float

Joystick.**getZ**(*hand=None*)

Joystick.**kDefaultThrottleAxis** = 3

Joystick.**kDefaultTopButton** = 2

Joystick.**kDefaultTriggerButton** = 1

Joystick.**kDefaultTwistAxis** = 2

Joystick.**kDefaultXAxis** = 0

Joystick.**kDefaultYAxis** = 1

Joystick.**kDefaultZAxis** = 2

Joystick.**setAxisChannel**(*axis*, *channel*)
Set the channel associated with a specified axis.

Parameters

- **axis** (*int*) – The axis to set the channel for.
- **channel** (*int*) – The channel to set the axis to.

Joystick.**setOutput**(*outputNumber*, *value*)

Joystick.**setOutputs**(*value*)

Joystick.**setRumble**(*type*, *value*)
Set the rumble output for the joystick. The DS currently supports 2 rumble values, left rumble and right rumble

Parameters

- **type** (Joystick.RumbleType) – Which rumble value to set
- **value** (*float*) – The normalized value (0 to 1) to set the rumble to

LinearDigitalFilter

class wpilib.**LinearDigitalFilter**(*source*, *ffGains*, *fbGains*)
Bases: *wpilib.Filter*

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

Note: 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 smoothes 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, I highly recommend the following articles:

- http://en.wikipedia.org/wiki/Linear_filter
- http://en.wikipedia.org/wiki/Iir_filter
- http://en.wikipedia.org/wiki/Fir_filter

Note: *pidGet()* should be called by the user on a known, regular period. You can set up a Notifier to do this (look at the *PIDController* class), or do it "inline" with code in a periodic function.

Note: 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 *pidGet()* gets called at the desired, constant frequency!

There are static methods you can use to build common filters:

- *highPass()*
- *movingAverage()*
- *singlePoleIIR()*

Constructor. Create a linear FIR or IIR filter

Parameters

- **source** (*PIDSource*, callable) – The PIDSource object that is used to get values
- **ffGains** (*list, tuple*) – The "feed forward" or FIR gains
- **fbGains** (*list, tuple*) – The "feed back" or IIR gains

get()

Returns the current filter estimate without also inserting new data as *pidGet()* would do.

Returns The current filter estimate

static highPass(*source*, *timeConstant*, *period*)

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

- **source** (*PIDSource*, callable) – The PIDSource object that is used to get values
- **timeConstant** (*float*) – The discrete-time time constant in seconds
- **period** (*float*) – The period in seconds between samples taken by the user

Returns *LinearDigitalFilter*

static movingAverage(*source*, *taps*)
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

- **source** (*PIDSource*, callable) – The PIDSource object that is used to get values
- **taps** – The number of samples to average over. Higher = smoother but slower

Raises `ValueError` if number of taps is less than 1

Returns *LinearDigitalFilter*

pidGet()
Calculates the next value of the filter

Returns The filtered value at this step

reset()
Reset the filter state

static singlePoleIIR(*source*, *timeConstant*, *period*)
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

- **source** (*PIDSource*, callable) – The PIDSource object that is used to get values
- **timeConstant** (*float*) – The discrete-time time constant in seconds
- **period** (*float*) – The period in seconds between samples taken by the user

Returns *LinearDigitalFilter*

LiveWindow

class wpilib.**LiveWindow**
Bases: `object`

The public interface for putting sensors and actuators on the LiveWindow.

static addActuator(*subsystem*, *name*, *component*)
Add an Actuator associated with the subsystem and with call it by the given name.

Parameters

- **subsystem** – The subsystem this component is part of.

- **name** – The name of this component.
- **component** – A LiveWindowSendable component that represents a actuator.

static addActuatorChannel(*moduleType*, *channel*, *component*)
Add Actuator to LiveWindow. The components are shown with the module type, slot and channel like this: Servo[0,2] for a servo object connected to the first digital module and PWM port 2.

Parameters

- **moduleType** – A string that defines the module name in the label for the value
- **channel** – The channel number the device is plugged into (usually PWM)
- **component** – The reference to the object being added

static addActuatorModuleChannel(*moduleType*, *moduleNumber*, *channel*, *component*)
Add Actuator to LiveWindow. The components are shown with the module type, slot and channel like this: Servo[0,2] for a servo object connected to the first digital module and PWM port 2.

Parameters

- **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 (usually PWM)
- **component** – The reference to the object being added

static addSensor(*subsystem*, *name*, *component*)
Add a Sensor associated with the subsystem and with call it by the given name.

Parameters

- **subsystem** – The subsystem this component is part of.
- **name** – The name of this component.
- **component** – A LiveWindowSendable component that represents a sensor.

static addSensorChannel(*moduleType*, *channel*, *component*)
Add Sensor to LiveWindow. The components are shown with the type and channel like this: Gyro[0] for a gyro object connected to the first analog channel.

Parameters

- **moduleType** – A string indicating the type of the module used in the naming (above)
- **channel** – The channel number the device is connected to
- **component** – A reference to the object being added

components = {}

firstTime = True

static initializeLiveWindowComponents()
Initialize all the LiveWindow elements the first time we enter LiveWindow mode. By holding off creating the NetworkTable entries, it allows them to be redefined before the first time in LiveWindow mode. This allows default sensor and actuator values to be created that are replaced with the custom names from users calling addActuator and addSensor.

liveWindowEnabled = False

livewindowTable = None

static removeComponent(*component*)
Removes a component from LiveWindow.

Parameters **component** – The reference to the object being removed.

static run()
The run method is called repeatedly to keep the values refreshed on the screen in test mode.

sensors = set()

static setEnabled(*enabled*)
Set the enabled state of LiveWindow. If it's being enabled, turn off the scheduler and remove all the commands from the queue and enable all the components registered for LiveWindow. If it's being disabled, stop all the registered components and reenable the scheduler.

TODO: add code to disable PID loops when enabling LiveWindow. The commands should reenable the PID loops themselves when they get rescheduled. This prevents arms from starting to move around, etc. after a period of adjusting them in LiveWindow mode.

statusTable = None

static updateValues()
Puts all sensor values on the live window.

LiveWindowSendable

class wpilib.**LiveWindowSendable**
Bases: *wpilib.Sendable*

A special type of object that can be displayed on the live window.

MotorSafety

class wpilib.**MotorSafety**
Bases: object

Provides mechanisms to safely shutdown motors if they aren't updated often enough.

The MotorSafety object is constructed for every object that wants to implement the Motor Safety protocol. The helper object has the code to actually do the timing and call the motors stop() method when the timeout expires. The motor object is expected to call the feed() method whenever the motors value is updated.

The constructor for a MotorSafety object. The helper object is constructed for every object that wants to implement the Motor Safety protocol. The helper object has the code to actually do the timing and call the motors stop() method when the timeout expires. The motor object is expected to call the feed() method whenever the motors value is updated.

DEFAULT_SAFETY_EXPIRATION = 0.1

check()
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()
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()
Feed the motor safety object. Resets the timer on this object that is used to do the timeouts.

getExpiration()
Retrieve the timeout value for the corresponding motor safety object.

Returns the timeout value in seconds.

Return type float

helpers = <_weakrefset.WeakSet object>

helpers_lock = <unlocked _thread.lock object>

isAlive()
Determine of the motor is still operating or has timed out.

Returns True if the motor is still operating normally and hasn't timed out.

Return type float

isSafetyEnabled()
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

Return type bool

setExpiration(*expirationTime*)
Set the expiration time for the corresponding motor safety object.

Parameters **expirationTime** (*float*) – The timeout value in seconds.

setSafetyEnabled(*enabled*)
Enable/disable motor safety for this device. Turn on and off the motor safety option for this PWM object.

Parameters **enabled** (*bool*) – True if motor safety is enforced for this object

PIDController

class wpilib.**PIDController**(**args*, ***kwargs*)
Bases: *wpilib.LiveWindowSendable*

Can be used to control devices via a PID Control Loop.

Creates a separate thread which reads the given *PIDSource* and takes care of the integral calculations, as well as writing the given *PIDOutput*.

This feedback controller runs in discrete time, so time deltas are not used in the integral and derivative calculations. Therefore, the sample rate affects the controller's behavior for a given set of PID constants.

Allocate a PID object with the given constants for P, I, D, and F

Arguments can be structured as follows:

•Kp, Ki, Kd, Kf, PIDSource, PIDOutput, period

•Kp, Ki, Kd, PIDSource, PIDOutput, period

•Kp, Ki, Kd, PIDSource, PIDOutput

•Kp, Ki, Kd, Kf, PIDSource, PIDOutput

Parameters

- **Kp** (*float or int*) – the proportional coefficient
- **Ki** (*float or int*) – the integral coefficient

- **Kd** (*float or int*) – the derivative coefficient
- **Kf** (*float or int*) – the feed forward term
- **source** (A function, or an object that implements *PIDSource*) – Called to get values
- **output** (A function, or an object that implements *PIDOutput*) – Receives the output percentage
- **period** (*float or int*) – the loop time for doing calculations. This particularly effects calculations of the integral and differential terms. The default is 50ms.

AbsoluteTolerance_onTarget(*value*)

class PIDSourceType

Bases: object

A description for the type of output value to provide to a *PIDController*

kDisplacement = 0

kRate = 1

PIDController.**PercentageTolerance_onTarget**(*percentage*)

PIDController.**calculateFeedForward**()

Calculate the feed forward term

Both of the provided feed forward calculations are velocity feed forwards. If a different feed forward calculation is desired, the user can override this function and provide his or her own. This function does no synchronization because the PIDController class only calls it in synchronized code, so be careful if calling it oneself.

If a velocity PID controller is being used, the F term should be set to 1 over the maximum setpoint for the output. If a position PID controller is being used, the F term should be set to 1 over the maximum speed for the output measured in setpoint units per this controller's update period (see the default period in this class's constructor).

PIDController.**disable**()

Stop running the PIDController, this sets the output to zero before stopping.

PIDController.**enable**()

Begin running the PIDController.

PIDController.**free**()

Free the PID object

PIDController.**get**()

Return the current PID result. This is always centered on zero and constrained the the max and min outs.

Returns the latest calculated output

PIDController.**getAvgError**()

Returns the current difference of the error over the past few iterations. You can specify the number of iterations to average with *setToleranceBuffer()* (defaults to 1). getAvgError() is used for the onTarget() function.

Returns the current average of the error

PIDController.**getContinuousError**(*error*)

Wraps error around for continuous inputs. The original error is returned if continuous mode is disabled. This is an unsynchronized function.

Parameters **error** – The current error of the PID controller.

Returns Error for continuous inputs.

PIDController.**getD**()
Get the Differential coefficient.

Returns differential coefficient

PIDController.**getDeltaSetpoint**()
Returns the change in setpoint over time of the PIDController

Returns the change in setpoint over time

PIDController.**getError**()
Returns the current difference of the input from the setpoint.

Returns the current error

PIDController.**getF**()
Get the Feed forward coefficient.

Returns feed forward coefficient

PIDController.**getI**()
Get the Integral coefficient

Returns integral coefficient

PIDController.**getP**()
Get the Proportional coefficient.

Returns proportional coefficient

PIDController.**getPIDSourceType**(*pidSourceType*)
Returns the type of input the PID controller is using

Returns the PID controller input type

PIDController.**getSetpoint**()
Returns the current setpoint of the PIDController.

Returns the current setpoint

PIDController.**instances = 0**

PIDController.**isAvgErrorValid**()
Returns whether or not any values have been collected. If no values have been collected, getAvgError is 0, which is invalid.

Returns True if *getAvgError()* is currently valid.

PIDController.**isEnable**()
Return True if PIDController is enabled.

PIDController.**kDefaultPeriod = 0.05**

PIDController.**onTarget**()
Return True if the error is within the percentage of the total input range, determined by setTolerance. This assumes that the maximum and minimum input were set using `setInput()`.

Returns True if the error is less than the tolerance

PIDController.**reset**()
Reset the previous error, the integral term, and disable the controller.

PIDController.**setAbsoluteTolerance**(*absvalue*)
Set the absolute error which is considered tolerable for use with *onTarget()*.

Parameters **absvalue** – absolute error which is tolerable in the units of the input object

PIDController.**setContinuous**(*continuous=True*)
Set the PID controller to consider the input to be continuous. Rather then using the max and min in as constraints, it considers them to be the same point and automatically calculates the shortest route to the setpoint.

Parameters **continuous** – Set to True turns on continuous, False turns off continuous

PIDController.**setInputRange**(*minimumInput*, *maximumInput*)
Sets the maximum and minimum values expected from the input.

Parameters

- **minimumInput** – the minimum percentage expected from the input
- **maximumInput** – the maximum percentage expected from the output

PIDController.**setOutputRange**(*minimumOutput*, *maximumOutput*)
Sets the minimum and maximum values to write.

Parameters

- **minimumOutput** – the minimum percentage to write to the output
- **maximumOutput** – the maximum percentage to write to the output

PIDController.**setPID**(*p*, *i*, *d*, *f=0.0*)
Set the PID Controller gain parameters. Set the proportional, integral, and differential coefficients.

Parameters

- **p** – Proportional coefficient
- **i** – Integral coefficient
- **d** – Differential coefficient
- **f** – Feed forward coefficient (optional, default is 0.0)

PIDController.**setPIDSourceType**(*pidSourceType*)
Sets what type of input the PID controller will use

Parameters **pidSourceType** – the type of input

PIDController.**setPercentTolerance**(*percentage*)
Set the percentage error which is considered tolerable for use with *onTarget()*. (Input of 15.0 = 15 percent)

Parameters **percentage** – percent error which is tolerable

PIDController.**setSetpoint**(*setpoint*)
Set the setpoint for the PIDController Clears the queue for GetAvgError().

Parameters **setpoint** – the desired setpoint

PIDController.**setTolerance**(*percent*)
Set the percentage error which is considered tolerable for use with *onTarget()*. (Input of 15.0 = 15 percent)

Parameters **percent** – error which is tolerable

Deprecated since version 2015.1: Use *setPercentTolerance()* or *setAbsoluteTolerance()* instead.

PIDController.**setToleranceBuffer**(*bufLength*)
Set the number of previous error samples to average for tolerancing. When determining whether a mechanism is on target, the user may want to use a rolling average of previous measurements instead of a precise position or velocity. This is useful for noisy sensors which return a few erroneous measurements when the mechanism is on target. However, the mechanism will not register as on target for at least the specified bufLength cycles.

Parameters **bufLength** (*int*) – Number of previous cycles to average.

PowerDistributionPanel

class wpilib.**PowerDistributionPanel**(*module=0*)
Bases: *wpilib.SensorBase*

Use to obtain voltage, current, temperature, power, and energy from the Power Distribution Panel over CAN

Parameters **module** (*int*) – CAN ID of the PDP

clearStickyFaults()
Clear all pdp sticky faults

getCurrent(*channel*)
Query the current of a single channel of the PDP

Returns The current of one of the PDP channels (channels 0-15) in Amperes

Return type float

getTemperature()
Query the temperature of the PDP

Returns The temperature of the PDP in degrees Celsius

Return type float

getTotalCurrent()
Query the current of all monitored PDP channels (0-15)

Returns The total current drawn from the PDP channels in Amperes

Return type float

getTotalEnergy()
Query the total energy drawn from the monitored PDP channels

Returns The total energy drawn from the PDP channels in Joules

Return type float

getTotalPower()
Query the total power drawn from the monitored PDP channels

Returns The total power drawn from the PDP channels in Watts

Return type float

getVoltage()
Query the input voltage of the PDP

Returns The voltage of the PDP in volts

Return type float

resetTotalEnergy()
Reset the total energy to 0

Preferences

class wpilib.**Preferences**
Bases: object

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 can not access the file directly, but may modify values at specific fields which will then be saved to the file when `save()` is called.

This class is thread safe.

This will also interact with `networktables.NetworkTable` by creating a table called "Preferences" with all the key-value pairs. To save using *NetworkTable*, simply set the boolean at position ~S A V E~ to true. Also, if the value of any variable is " in the *NetworkTable*, then that represents non-existence in the *Preferences* table.

Creates a preference class that will automatically read the file in a different thread. Any call to its methods will be blocked until the thread is finished reading.

TABLE_NAME = 'Preferences'

containsKey(*key*)
Returns whether or not there is a key with the given name.

Parameters **key** – the key

Returns True if there is a value at the given key

getBoolean(*key*, *backup=None*)
Returns the boolean at the given key. If this table does not have a value for that position, then the given backup value will be returned.

Parameters

- **key** – the key
- **backup** – the value to return if none exists in the table

Returns either the value in the table, or the backup

getFloat(*key*, *backup=None*)
Returns the float at the given key. If this table does not have a value for that position, then the given backup value will be returned.

Parameters

- **key** – the key
- **backup** – the value to return if none exists in the table

Returns either the value in the table, or the backup

Raises TableKeyNotDefinedException if key cannot be found

static getInstance()
Returns the preferences instance.

Returns the preferences instance

getInt(*key*, *backup=None*)
Returns the int at the given key. If this table does not have a value for that position, then the given backup value will be returned.

Parameters

- **key** – the key
- **backup** – the value to return if none exists in the table

Returns either the value in the table, or the backup

Raises TableKeyNotDefinedException if key cannot be found

getKeys()

Returns a list of the keys

getString(*key*, *backup=None*)

Returns the string at the given key. If this table does not have a value for that position, then the given backup value will be returned.

Parameters

- **key** – the key
- **backup** – the value to return if none exists in the table

Returns either the value in the table, or the backup

keys()

Python style get list of keys.

putBoolean(*key*, *value*)

Puts the given float into the preferences table.

The key may not have any whitespace nor an equals sign.

This will NOT save the value to memory between power cycles, to do that you must call `save()` (which must be used with care) at some point after calling this.

Parameters

- **key** – the key
- **value** – the value

putFloat(*key*, *value*)

Puts the given float into the preferences table.

The key may not have any whitespace nor an equals sign.

This will NOT save the value to memory between power cycles, to do that you must call `save()` (which must be used with care) at some point after calling this.

Parameters

- **key** – the key
- **value** – the value

putInt(*key*, *value*)

Puts the given int into the preferences table.

The key may not have any whitespace nor an equals sign.

This will NOT save the value to memory between power cycles, to do that you must call `save()` (which must be used with care) at some point after calling this.

Parameters

- **key** – the key
- **value** – the value

putString(*key*, *value*)
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.

This will NOT save the value to memory between power cycles, to do that you must call `save()` (which must be used with care) at some point after calling this.

Parameters

- **key** – the key
- **value** – the value

remove(*key*)
Remove a preference

Parameters **key** – the key

valueChangedEx(*source*, *key*, *value*, *isNew*)

PWM

class wpilib.**PWM**(*channel*)
Bases: *wpilib.LiveWindowSendable*

Raw interface to 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. There is no delay.

As of revision 0.1.10 of the FPGA, the FPGA interprets the 0-2000 values as follows:

•2000 = full "forward"

•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 .5ms)

•0 = disabled (i.e. PWM output is held low)

kDefaultPwmPeriod is the 1x period (5.05 ms). In hardware, the period scaling is implemented as an output squelch to get longer periods for old devices.

•20ms periods (50 Hz) are the "safest" setting in that this works for all devices

•20ms periods seem to be desirable for Vex Motors

•20ms periods are the specified period for HS-322HD servos, but work reliably down to 10.0 ms; starting at about 8.5ms, the servo sometimes hums and get hot; by 5.0ms the hum is nearly continuous

•10ms periods work well for Victor 884

•5ms periods allows higher update rates for Luminary Micro Jaguar speed controllers. Due to the shipping firmware on the Jaguar, we can't run the update period less than 5.05 ms.

Allocate a PWM given a channel.

Parameters **channel** (*int*) – The PWM channel number. 0-9 are on-board, 10-19 are on the MXP port

class **PeriodMultiplier**
Bases: object

Represents the amount to multiply the minimum servo-pulse pwm period by.

k1X = 1
Period Multiplier: don't skip pulses.

k2X = 2
Period Multiplier: skip every other pulse.

k4X = 4
Period Multiplier: skip three out of four pulses.

PWM.**enableDeadbandElimination**(*eliminateDeadband*)
Optionally eliminate the deadband from a speed controller.

Parameters **eliminateDeadband** (*bool*) – 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.

PWM.**free**()
Free the PWM channel.

Free the resource associated with the PWM channel and set the value to 0.

PWM.**getChannel**()
Gets the channel number associated with the PWM Object.

Returns The channel number.

Return type int

PWM.**getPosition**()
Get the PWM value in terms of a position.

This is intended to be used by servos.

Note: *setBounds()* must be called first.

Returns The position the servo is set to between 0.0 and 1.0.

Return type float

PWM.**getRaw**()
Get the PWM value directly from the hardware.

Read a raw value from a PWM channel.

Returns Raw PWM control value. Range: 0 - 255.

Return type int

PWM.**getRawBounds**()
Gets the bounds on the PWM pulse widths. This Gets 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.

Returns tuple of (max, deadbandMax, center, deadbandMin, min)

PWM.**getSpeed**()
Get the PWM value in terms of speed.

This is intended to be used by speed controllers.

Note: *setBounds()* must be called first.

Returns The most recently set speed between -1.0 and 1.0.

Return type float

PWM.**handle**

PWM.**setBounds**(*max*, *deadbandMax*, *center*, *deadbandMin*, *min*)
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** (*float*) – The max PWM pulse width in ms
- **deadbandMax** (*float*) – The high end of the deadband range pulse width in ms
- **center** (*float*) – The center (off) pulse width in ms
- **deadbandMin** (*float*) – The low end of the deadband pulse width in ms
- **min** (*float*) – The minimum pulse width in ms

PWM.**setDisabled**()
Temporarily disables the PWM output. The next set call will reenable the output.

PWM.**setPeriodMultiplier**(*mult*)
Slow down the PWM signal for old devices.

Parameters **mult** (PWM.PeriodMultiplier) – The period multiplier to apply to this channel

PWM.**setPosition**(*pos*)
Set the PWM value based on a position.

This is intended to be used by servos.

Note: *setBounds()* must be called first.

Parameters **pos** (*float*) – The position to set the servo between 0.0 and 1.0.

PWM.**setRaw**(*value*)
Set the PWM value directly to the hardware.

Write a raw value to a PWM channel.

Parameters **value** (*int*) – Raw PWM value. Range 0 - 255.

PWM.**setRawBounds**(*max*, *deadbandMax*, *center*, *deadbandMin*, *min*)
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** (*int*) – The Minimum pwm value
- **deadbandMax** (*int*) – The high end of the deadband range

- **center** (*int*) – The center speed (off)
- **deadbandMin** (*int*) – The low end of the deadband range
- **min** (*int*) – The minimum pwm value

Deprecated since version 2017.0.0: Recommended to set bounds in ms using *setBounds()* instead

PWM.**setSpeed**(*speed*)
Set the PWM value based on a speed.

This is intended to be used by speed controllers.

Note: *setBounds()* must be called first.

Parameters **speed** (*float*) – The speed to set the speed controller between -1.0 and 1.0.

PWM.**setZeroLatch**()

PWMSpeedController

class wpilib.**PWMSpeedController**(*channel*)
Bases: *wpilib.SafePWM*

Common base class for all PWM Speed Controllers.

free()

get()
Get the recently set value of the PWM.

Returns The most recently set value for the PWM between -1.0 and 1.0.

Return type float

getInverted()
Common interface for inverting the direction of a speed controller.

Returns The state of inversion (True is inverted)

pidWrite(*output*)
Write out the PID value as seen in the PIDOutput base object.

Parameters **output** (*float*) – Write out the PWM value as was found in the PIDController.

set(*speed*)
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** (*float*) – The speed to set. Value should be between -1.0 and 1.0.

setInverted(*isInverted*)
Common interface for inverting the direction of a speed controller.

Parameters **isInverted** – The state of inversion (True is inverted).

Relay

class wpilib.**Relay**(*channel*, *direction=None*)
Bases: *wpilib.SensorBase*, *wpilib.LiveWindowSendable*, *wpilib.MotorSafety*

Controls VEX Robotics Spike style relay outputs.

Relays are intended to be connected to Spikes or similar relays. The relay channels controls a pair of channels 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.

Initially the relay is set to both lines at 0v.

Parameters

- **channel** (*int*) – The channel number for this relay (0-3)
- **direction** (*Relay.Direction*) – The direction that the Relay object will control. If not specified, defaults to allowing both directions.

class **Direction**
Bases: object

The Direction(s) that a relay is configured to operate in.

kBoth = 0
Both directions are valid

kForward = 1
Only forward is valid

kReverse = 2
Only reverse is valid

class Relay.**Value**
Bases: object

The state to drive a Relay to.

kForward = 2
Forward

kOff = 0
Off

kOn = 1
On for relays with defined direction

kReverse = 3
Reverse

Relay.**forwardHandle**

Relay.**free**()

Relay.**get**()
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

Return type *Relay.Value*

`Relay.`**`getChannel`**`()`
Get the channel number.

Returns The channel number.

`Relay.`**`getDescription`**`()`

`Relay.`**`relayChannels = <wpilib.resource.Resource object>`**

`Relay.`**`reverseHandle`**

`Relay.`**`set`**`(`*value*`)`
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 set to any of the four states: 0v-0v, 12v-0v, 0v-12v, 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`** (*Relay.Value*) – The state to set the relay.

`Relay.`**`setDirection`**`(`*direction*`)`
Set the Relay Direction.

Changes which values the relay can be set to depending on which direction is used.

Valid inputs are kBothDirections, kForwardOnly, and kReverseOnly.

Parameters **`direction`** (*Relay.Direction*) – The direction for the relay to operate in

`Relay.`**`stopMotor`**`()`

Resource

class `wpilib.`**`Resource`**`(`*size*`)`
Bases: `object`

Tracks resources in the program.

The Resource class is a convenient way of keeping track of allocated arbitrary resources in the program. Resources are just indices that have an lower and upper bound that are tracked by this class. In the library they are used for tracking allocation of hardware channels but this is purely arbitrary. The resource class does not do any actual allocation, but simply tracks if a given index is currently in use.

Allocate storage for a new instance of Resource. Allocate a bool array of values that will get initialized to indicate that no resources have been allocated yet. The indicies of the resources are 0..size-1.

Parameters **`size`** – The number of blocks to allocate

`allocate``(`*obj*, *index=None*`)`
Allocate a resource.

When index is None or unspecified, a free resource value within the range is located and returned after it is marked allocated. Otherwise, it is verified unallocated, then returned.

Parameters

- **obj** – The object requesting the resource.
- **index** – The resource to allocate

Returns The index of the allocated block.

Raises **IndexError** – If there are no resources available to be allocated or the specified index is already used.

free(*index*)
Force-free an allocated resource. After a resource is no longer needed, for example a destructor is called for a channel assignment class, free will release the resource value so it can be reused somewhere else in the program.

Parameters **index** – The index of the resource to free.

RobotBase

class wpilib.**RobotBase**
Bases: 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.

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.

Warning: If you override __init__ in your robot class, you must call the base class constructor. This must be used to ensure that the communications code starts.

free()
Free the resources for a RobotBase class.

static **initializeHardwareConfiguration**()
Common initialization for all robot programs.

isAutonomous()
Determine if the robot is currently in Autonomous mode as determined by the field controls.

Returns True if the robot is currently operating Autonomously

Return type bool

isDisabled()
Determine if the Robot is currently disabled.

Returns True if the Robot is currently disabled by the field controls.

Return type bool

isEnabled()
Determine if the Robot is currently enabled.

Returns True if the Robot is currently enabled by the field controls.

Return type bool

isNewDataAvailable()
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?

Return type bool

isOperatorControl()
Determine if the robot is currently in Operator Control mode as determined by the field controls.

Returns True if the robot is currently operating in Tele-Op mode

Return type bool

static isReal()

Returns If the robot is running in the real world.

Return type bool

static isSimulation()

Returns If the robot is running in simulation.

Return type bool

isTest()
Determine if the robot is currently in Test mode as determined by the driver station.

Returns True if the robot is currently operating in Test mode.

Return type bool

static main(*robot_cls*)
Starting point for the applications.

startCompetition()
Provide an alternate "main loop" via startCompetition().

RobotDrive

class wpilib.**RobotDrive**(**args*, ***kwargs*)
Bases: *wpilib.MotorSafety*

Operations on a robot drivetrain based on a definition of the motor configuration.

The robot drive class handles basic driving for a robot. Currently, 2 and 4 motor tank and mecanum drive trains are supported. In the future other drive types like swerve might be implemented. Motor channel numbers are passed supplied on creation of the class. Those are used for either the drive function (intended for hand created drive code, such as autonomous) or with the Tank/Arcade functions intended to be used for Operator Control driving.

Constructor for RobotDrive.

Either 2 or 4 motors can be passed to the constructor to implement a two or four wheel drive system, respectively.

When positional arguments are used, these are the two accepted orders:

•leftMotor, rightMotor

•frontLeftMotor, rearLeftMotor, frontRightMotor, rearRightMotor

Alternatively, the above names can be used as keyword arguments.

Either channel numbers or motor controllers can be passed (determined by whether the passed object has a *set* function). If channel numbers are passed, the motorController keyword argument, if present, is the motor controller class to use; if unspecified, `Talon` is used.

class MotorType
Bases: `object`

The location of a motor on the robot for the purpose of driving.

kFrontLeft = 0
Front left

kFrontRight = 1
Front right

kRearLeft = 2
Rear left

kRearRight = 3
Rear right

`RobotDrive.`**arcadeDrive**(**args*, ***kwargs*)
Provide tank steering using the stored robot configuration.

Either one or two joysticks (with optional specified axis) or two raw values may be passed positionally, along with an optional squaredInputs boolean. The valid positional combinations are:

•stick

•stick, squaredInputs

•moveStick, moveAxis, rotateStick, rotateAxis

•moveStick, moveAxis, rotateStick, rotateAxis, squaredInputs

•moveValue, rotateValue

•moveValue, rotateValue, squaredInputs

Alternatively, the above names can be used as keyword arguments. The behavior of mixes of keyword arguments in other than the combinations above is undefined.

If specified positionally, the value and joystick versions are disambiguated by looking for a *getY* function on the stick.

Parameters

- **stick** – The joystick to use for Arcade single-stick driving. The Y-axis will be selected for forwards/backwards and the X-axis will be selected for rotation rate.
- **moveStick** – The Joystick object that represents the forward/backward direction.
- **moveAxis** – The axis on the moveStick object to use for forwards/backwards (typically Y_AXIS).
- **rotateStick** – The Joystick object that represents the rotation value.
- **rotateAxis** – The axis on the rotation object to use for the rotate right/left (typically X_AXIS).
- **moveValue** – The value to use for forwards/backwards.
- **rotateValue** – The value to use for the rotate right/left.
- **squaredInputs** – Setting this parameter to True decreases the sensitivity at lower speeds. Defaults to True if unspecified.

RobotDrive.**drive**(*outputMagnitude*, *curve*)
Drive the motors at "outputMagnitude" and "curve".

Both outputMagnitude and curve are -1.0 to +1.0 values, where 0.0 represents stopped and not turning. `curve < 0` will turn left and `curve > 0` will turn right.

The algorithm for steering provides a constant turn radius for any normal speed range, both forward and backward. Increasing m_sensitivity causes sharper turns for fixed values of curve.

This function will most likely be used in an autonomous routine.

Parameters

- **outputMagnitude** – The speed setting for the outside wheel in a turn, forward or backwards, +1 to -1.
- **curve** – The rate of turn, constant for different forward speeds. Set `curve < 0` for left turn or `curve > 0` for right turn.

Set `curve = e^(-r/w)` to get a turn radius r for wheelbase w of your robot. Conversely, turn radius r = -ln(curve)*w for a given value of curve and wheelbase w.

RobotDrive.**free**()

RobotDrive.**getDescription**()

RobotDrive.**getNumMotors**()

RobotDrive.**holonomicDrive**(*magnitude*, *direction*, *rotation*)
Holonomic Drive method for Mecanum wheeled robots.

This is an alias to *mecanumDrive_Polar()* for backward compatibility.

Parameters

- **magnitude** – The speed that the robot should drive in a given direction. [-1.0..1.0]
- **direction** – The direction the robot should drive. The direction and magnitude are independent of the rotation rate.
- **rotation** – The rate of rotation for the robot that is completely independent of the magnitude or direction. [-1.0..1.0]

RobotDrive.**kArcadeRatioCurve_Reported = False**

RobotDrive.**kArcadeStandard_Reported = False**

RobotDrive.**kDefaultExpirationTime = 0.1**

RobotDrive.**kDefaultMaxOutput = 1.0**

RobotDrive.**kDefaultSensitivity = 0.5**

RobotDrive.**kMaxNumberOfMotors = 4**

RobotDrive.**kMecanumCartesian_Reported = False**

RobotDrive.**kMecanumPolar_Reported = False**

RobotDrive.**kTank_Reported = False**

static RobotDrive.**limit**(*num*)
Limit motor values to the -1.0 to +1.0 range.

RobotDrive.**mecanumDrive_Cartesian**(*x*, *y*, *rotation*, *gyroAngle*)
Drive method for Mecanum wheeled robots.

A method for driving with Mecanum wheeled robots. There are 4 wheels on the robot, arranged so that the front and back wheels are toed in 45 degrees. When looking at the wheels from the top, the roller axles should form an X across the robot.

This is designed to be directly driven by joystick axes.

Parameters

- **x** – The speed that the robot should drive in the X direction. [-1.0..1.0]
- **y** – The speed that the robot should drive in the Y direction. This input is inverted to match the forward == -1.0 that joysticks produce. [-1.0..1.0]
- **rotation** – The rate of rotation for the robot that is completely independent of the translation. [-1.0..1.0]
- **gyroAngle** – The current angle reading from the gyro. Use this to implement field-oriented controls.

RobotDrive.**mecanumDrive_Polar**(*magnitude*, *direction*, *rotation*)

Drive method for Mecanum wheeled robots.

A method for driving with Mecanum wheeled robots. There are 4 wheels on the robot, arranged so that the front and back wheels are toed in 45 degrees. When looking at the wheels from the top, the roller axles should form an X across the robot.

Parameters

- **magnitude** – The speed that the robot should drive in a given direction.
- **direction** – The direction the robot should drive in degrees. The direction and maginitute are independent of the rotation rate.
- **rotation** – The rate of rotation for the robot that is completely independent of the magnitute or direction. [-1.0..1.0]

static RobotDrive.**normalize**(*wheelSpeeds*)

Normalize all wheel speeds if the magnitude of any wheel is greater than 1.0.

static RobotDrive.**rotateVector**(*x*, *y*, *angle*)

Rotate a vector in Cartesian space.

RobotDrive.**setInvertedMotor**(*motor*, *isInverted*)

Invert a motor direction.

This is used when a motor should run in the opposite direction as the drive code would normally run it. Motors that are direct drive would be inverted, the drive code assumes that the motors are geared with one reversal.

Parameters

- **motor** – The motor index to invert.
- **isInverted** – True if the motor should be inverted when operated.

RobotDrive.**setLeftRightMotorOutputs**(*leftOutput*, *rightOutput*)

Set the speed of the right and left motors.

This is used once an appropriate drive setup function is called such as twoWheelDrive(). The motors are set to “leftSpeed” and “rightSpeed” and includes flipping the direction of one side for opposing motors.

Parameters

- **leftOutput** – The speed to send to the left side of the robot.
- **rightOutput** – The speed to send to the right side of the robot.

RobotDrive.**setMaxOutput**(*maxOutput*)
Configure the scaling factor for using RobotDrive with motor controllers in a mode other than PercentVbus.

Parameters **maxOutput** – Multiplied with the output percentage computed by the drive functions.

RobotDrive.**setSensitivity**(*sensitivity*)
Set the turning sensitivity.

This only impacts the drive() entry-point.

Parameters **sensitivity** – Effectively sets the turning sensitivity (or turn radius for a given value)

RobotDrive.**stopMotor**()

RobotDrive.**tankDrive**(**args*, ***kwargs*)
Provide tank steering using the stored robot configuration.

Either two joysticks (with optional specified axis) or two raw values may be passed positionally, along with an optional squaredInputs boolean. The valid positional combinations are:

•leftStick, rightStick

•leftStick, rightStick, squaredInputs

•leftStick, leftAxis, rightStick, rightAxis

•leftStick, leftAxis, rightStick, rightAxis, squaredInputs

•leftValue, rightValue

•leftValue, rightValue, squaredInputs

Alternatively, the above names can be used as keyword arguments. The behavior of mixes of keyword arguments in other than the combinations above is undefined.

If specified positionally, the value and joystick versions are disambiguated by looking for a *getY* function.

Parameters

- **leftStick** – The joystick to control the left side of the robot.
- **leftAxis** – The axis to select on the left side Joystick object (defaults to the Y axis if unspecified).
- **rightStick** – The joystick to control the right side of the robot.
- **rightAxis** – The axis to select on the right side Joystick object (defaults to the Y axis if unspecified).
- **leftValue** – The value to control the left side of the robot.
- **rightValue** – The value to control the right side of the robot.
- **squaredInputs** – Setting this parameter to True decreases the sensitivity at lower speeds. Defaults to True if unspecified.

RobotState

class wpilib.**RobotState**
Bases: object

Provides an interface to determine the current operating state of the robot code.

impl = None

static isAutonomous()

static isDisabled()

static isEnabled()

static isOperatorControl()

static isTest()

SafePWM

class wpilib.**SafePWM**(*channel*)

Bases: *wpilib.PWM*, *wpilib.MotorSafety*

A raw PWM interface that implements the *MotorSafety* interface

Constructor for a SafePWM object taking a channel number.

Parameters **channel** (*int*) – The channel number to be used for the underlying PWM object. 0-9 are on-board, 10-19 are on the MXP port.

disable()

getDescription()

stopMotor()

Stop the motor associated with this PWM object. This is called by the MotorSafety object when it has a timeout for this PWM and needs to stop it from running.

SampleRobot

class wpilib.**SampleRobot**

Bases: *wpilib.RobotBase*

A simple robot base class that knows the standard FRC competition states (disabled, autonomous, or operator controlled).

You can build a simple robot program off of this by overriding the robotinit(), *disabled()*, *autonomous()* and *operatorControl()* methods. The *startCompetition()* method will call these methods (sometimes repeatedly) depending on the state of the competition.

Alternatively you can override the *robotMain()* method and manage all aspects of the robot yourself (not recommended).

Warning: While it may look like a good choice to use for your code if you're inexperienced, don't. Unless you know what you are doing, complex code will be much more difficult under this system. Use *IterativeRobot* or command based instead if you're new.

autonomous()

Autonomous should go here. Users should add autonomous code to this method that should run while the field is in the autonomous period.

Called once each time the robot enters the autonomous state.

disabled()

Disabled should go here. Users should overload this method to run code that should run while the field is disabled.

Called once each time the robot enters the disabled state.

logger = <logging.Logger object>
A python logging object that you can use to send messages to the log. It is recommended to use this instead of print statements.

operatorControl()
Operator control (tele-operated) code should go here. Users should add Operator Control code to this method that should run while the field is in the Operator Control (tele-operated) period.

Called once each time the robot enters the operator-controlled state.

robotInit()
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 1 time.

Note: It is simpler to override this function instead of defining a constructor for your robot class

---

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.

robotMain()
Robot main program for free-form programs.

This should be overridden by user subclasses if the intent is to not use the autonomous() and operatorControl() methods. In that case, the program is responsible for sensing when to run the autonomous and operator control functions in their program.

This method will be called immediately after the constructor is called. If it has not been overridden by a user subclass (i.e. the default version runs), then the robotInit(), disabled(), autonomous() and operatorControl() methods will be called.

If you override this function, you must call `hal.HALNetworkCommunicationObserveUserProgramStarting()` to indicate that your robot is ready to be enabled, as it will not be called for you.

Warning: Nobody actually wants to override this function. Neither do you.

startCompetition()
Start a competition. This code tracks the order of the field starting to ensure that everything happens in the right order. Repeatedly run the correct method, either Autonomous or OperatorControl when the robot is enabled. After running the correct method, wait for some state to change, either the other mode starts or the robot is disabled. Then go back and wait for the robot to be enabled again.

test()
Test code should go here. Users should add test code to this method that should run while the robot is in test mode.

SD540

class wpilib.**SD540**(*channel*)
Bases: *wpilib.PWMSpeedController*

Mindsensors SD540 Speed Controller

Constructor.

Parameters **channel** – The PWM channel that the SD540 is attached to. 0-9 are on-board, 10-19 are on the MXP port

---

Note: 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
- .94ms = full "reverse"

Sendable

class wpilib.**Sendable**
Bases: object

The base interface for objects that can be sent over the network through network tables

SendableChooser

class wpilib.**SendableChooser**
Bases: *wpilib.Sendable*

A useful tool for presenting a selection of options to be displayed on 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.

Example:

```
# This shows the user two options on the SmartDashboard
chooser = wpilib.SendableChooser()
chooser.addObject('option1', '1')
chooser.addObject('option2', '2')

wpilib.SmartDashboard.putData('Choice', chooser)

# .. later, ask to see what the user selected?
value = chooser.getSelected()
```

Instantiates a SendableChooser.

DEFAULT = 'default'

OPTIONS = 'options'

SELECTED = 'selected'

addDefault(*name*, *object*)
Add the given object to the list of options and marks it as the default. Functionally, this is very close to addObject(...) except that it will use this as the default option if none other is explicitly selected.

Parameters

- **name** – the name of the option
- **object** – the option

addObject(*name*, *object*)
Adds the given object to the list of options. On the SmartDashboard on the desktop, the object will appear as the given name.

Parameters

- **name** – the name of the option
- **object** – the option

getSelected()
Returns the object associated with the selected option. If there is none selected, it will return the default. If there is none selected and no default, then it will return None.

Returns the object associated with the selected option

SensorBase

class wpilib.**SensorBase**
Bases: *wpilib.LiveWindowSendable*

Base class for all sensors

Stores most recent status information as well as containing utility functions for checking channels and error processing.

static checkAnalogInputChannel(*channel*)
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.

Parameters **channel** – The channel number to check.

static checkAnalogOutputChannel(*channel*)
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.

Parameters **channel** – The channel number to check.

static checkDigitalChannel(*channel*)
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.

Parameters **channel** – The channel number to check.

static checkPDPChannel(*channel*)
Verify that the power distribution channel number is within limits. Channel numbers are 0-based.

Parameters **channel** – The channel number to check.

static checkPDPModule(*module*)
Verify that the power distribution module number is within limits. Module numbers are 0-based.

Parameters **module** – The module number to check.

static checkPWMChannel(*channel*)
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.

Parameters **channel** – The channel number to check.

static checkRelayChannel(*channel*)
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.

Parameters **channel** – The channel number to check.

static checkSolenoidChannel(*channel*)
Verify that the solenoid channel number is within limits. Channel numbers are 0-based.

Parameters **channel** – The channel number to check.

static checkSolenoidModule(*moduleNumber*)
Verify that the solenoid module is correct.

Parameters **moduleNumber** – The solenoid module module number to check.

defaultSolenoidModule = 0
Default solenoid module

free()
Free the resources used by this object

static getDefaultSolenoidModule()
Get the number of the default solenoid module.

Returns The number of the default solenoid module.

kAnalogInputChannels = 8
Number of analog input channels per roboRIO

kAnalogOutputChannels = 2
Number of analog output channels per roboRIO

kDigitalChannels = 31
Number of digital channels per roboRIO

kPCMModules = 63
Number of PCM modules

kPDPChannels = 16
Number of power distribution channels per PDP

kPDPModules = 63
Number of power distribution channels per PDP

kPwmChannels = 20
Number of PWM channels per roboRIO

kRelayChannels = 4
Number of relay channels per roboRIO

kSolenoidChannels = 8
Number of solenoid channels per module

kSystemClockTicksPerMicrosecond = 40
Ticks per microsecond

static setDefaultSolenoidModule(*moduleNumber*)
Set the default location for the Solenoid module.

Parameters **moduleNumber** – The number of the solenoid module to use.

Servo

class wpilib.**Servo**(*channel*)

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.

Constructor.

- By default *kDefaultMaxServoPWM* ms is used as the maxPWM value
- By default *kDefaultMinServoPWM* ms is used as the minPWM value

Parameters **channel** (*int*) – The PWM channel to which the servo is attached. 0-9 are on-board, 10-19 are on the MXP port.

free()

get()

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.

Return type float

getAngle()

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.

Return type float

getServoAngleRange()

kDefaultMaxServoPWM = 2.4

kDefaultMinServoPWM = 0.6

kMaxServoAngle = 180.0

kMinServoAngle = 0.0

set(*value*)

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** (*float*) – Position from 0.0 to 1.0.

setAngle(*degrees*)

Set the servo angle.

Assumes 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** (*float*) – The angle in degrees to set the servo.

SmartDashboard

class wpilib.**SmartDashboard**

Bases: object

The bridge between robot programs and the SmartDashboard on the laptop

When a value is put into the SmartDashboard, it pops up on the SmartDashboard on the remote host. Users can put values into and get values from the SmartDashboard.

These values can also be accessed by a NetworkTables client via the 'SmartDashboard' table:

```
from networktables import NetworkTable
sd = NetworkTable.getTable('SmartDashboard')

# sd.putXXX and sd.getXXX work as expected here
```

classmethod **clearFlags**(*key*, *flags*)

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)

classmethod **clearPersistent**(*key*)

Stop making a key's value persistent through program restarts. The key cannot be null.

Parameters **key** – the key name

classmethod **containsKey**(*key*)

Checks the table and tells if it contains the specified key.

Parameters **key** – key the key to search for

Returns true if the table as a value assigned to the given key

classmethod **delete**(*key*)

Deletes the specified key in this table. The key can not be null.

Parameters **key** – the key name

classmethod **getBoolean**(*key*, *defaultValue=<class 'wpilib.smartdashboard.SmartDashboard._defaultValueSentry'>*)

Returns the boolean the key maps to. If the key does not exist or is of different type, it will return the default value; if that is not provided, it will throw a KeyError.

Calling this method without passing defaultValue is deprecated.

Parameters

- **key** (*str*) – the key to look up
- **defaultValue** – returned if the key doesn't exist

Returns the value associated with the given key or the given default value if there is no value associated with the key

Raises `KeyError` if the key doesn't exist and defaultValue is not provided.

classmethod getBooleanArray(*key*, *defaultValue=<class 'wpilib.smartdashboard.SmartDashboard._defaultValueSentry'>*)
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; if that is not provided, it will throw a `KeyError`.

Calling this method without passing defaultValue is deprecated.

Parameters

- **key** (*str*) – the key to look up
- **defaultValue** – returned if the key doesn't exist

Returns the value associated with the given key or the given default value if there is no value associated with the key

Raises `KeyError` if the key doesn't exist and defaultValue is not provided.

classmethod getData(*key*)
Returns the value at the specified key.

Parameters **key** (*str*) – the key

Returns the value

Raises `KeyError` if the key doesn't exist

classmethod getDouble(*key*, *defaultValue=<class 'wpilib.smartdashboard.SmartDashboard._defaultValueSentry'>*)
Returns the number the key maps to. If the key does not exist or is of different type, it will return the default value; if that is not provided, it will throw a `KeyError`.

Calling this method without passing defaultValue is deprecated.

Parameters

- **key** (*str*) – the key to look up
- **defaultValue** – returned if the key doesn't exist

Returns the value associated with the given key or the given default value if there is no value associated with the key

Raises `KeyError` if the key doesn't exist and defaultValue is not provided.

classmethod getFlags(*key*)
Returns the flags for the specified key.

Parameters **key** – the key name

Returns the flags, or 0 if the key is not defined

classmethod getInt(*key*, *defaultValue=<class 'wpilib.smartdashboard.SmartDashboard._defaultValueSentry'>*)
Returns the number the key maps to. If the key does not exist or is of different type, it will return the default value; if that is not provided, it will throw a `KeyError`.

Calling this method without passing defaultValue is deprecated.

Parameters

- **key** (*str*) – the key to look up
- **defaultValue** – returned if the key doesn't exist

Returns the value associated with the given key or the given default value if there is no value associated with the key

Raises `KeyError` if the key doesn't exist and defaultValue is not provided.

classmethod **getKeys**(*types=0*)
Get array of keys in the table.

Parameters **types** – bitmask of types; 0 is treated as a "don't care".

Returns keys currently in the table

classmethod **getNumber**(*key*, *defaultValue=<class 'wpilib.smartdashboard.SmartDashboard._defaultValueSentry'>*)
Returns the number the key maps to. If the key does not exist or is of different type, it will return the default value; if that is not provided, it will throw a `KeyError`.

Calling this method without passing defaultValue is deprecated.

Parameters

- **key** (*str*) – the key to look up
- **defaultValue** – returned if the key doesn't exist

Returns the value associated with the given key or the given default value if there is no value associated with the key

Raises `KeyError` if the key doesn't exist and defaultValue is not provided.

classmethod **getNumberArray**(*key*, *defaultValue=<class 'wpilib.smartdashboard.SmartDashboard._defaultValueSentry'>*)
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; if that is not provided, it will throw a `KeyError`.

Calling this method without passing defaultValue is deprecated.

Parameters

- **key** (*str*) – the key to look up
- **defaultValue** – returned if the key doesn't exist

Returns the value associated with the given key or the given default value if there is no value associated with the key

Raises `KeyError` if the key doesn't exist and defaultValue is not provided.

classmethod **getRaw**(*key*, *defaultValue=<class 'wpilib.smartdashboard.SmartDashboard._defaultValueSentry'>*)
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; if that is not provided, it will throw a `KeyError`.

Calling this method without passing defaultValue is deprecated.

Parameters

- **key** (*str*) – the key to look up
- **defaultValue** – returned if the key doesn't exist

Returns the value associated with the given key or the given default value if there is no value associated with the key

Raises `KeyError` if the key doesn't exist and defaultValue is not provided.

classmethod **getString**(*key*, *defaultValue=<class 'wpilib.smartdashboard.SmartDashboard._defaultValueSentry'>*)
Returns the string the key maps to. If the key does not exist or is of different type, it will return the default value; if that is not provided, it will throw a `KeyError`.

Calling this method without passing defaultValue is deprecated.

Parameters

- **key** (*str*) – the key to look up

- **defaultValue** – returned if the key doesn't exist

Returns the value associated with the given key or the given default value if there is no value associated with the key

Raises `KeyError` if the key doesn't exist and defaultValue is not provided.

classmethod getStringArray(*key*, *defaultValue=<class 'wpilib.smartdashboard.SmartDashboard._defaultValueSentry'>*)
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** (*str*) – the key to look up
- **defaultValue** (*list(str)*) – 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

Return type list(str)

Raises **KeyError** – If the value doesn't exist and no default is provided, or if it is the wrong type

classmethod isPersistent(*key*)
Returns whether the value is persistent through program restarts. The key cannot be null.

Parameters **key** – the key name

Returns True if the value is persistent.

classmethod putBoolean(*key*, *value*)
Put a boolean in the table.

Parameters

- **key** – the key to be assigned to
- **value** – the value that will be assigned

:return False if the table key already exists with a different type

classmethod putBooleanArray(*key*, *value*)
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

classmethod putData(**args*, ***kwargs*)
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.

Two argument formats are supported: key, data:

Parameters

- **key** (*str*) – the key (cannot be None)
- **data** – the value

Or the single argument "value":

Parameters **value** – the named value (getName is called to retrieve the value)

classmethod putDouble(*key*, *value*)
Put a number 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

classmethod putInt(*key*, *value*)
Put a number 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

classmethod putNumber(*key*, *value*)
Put a number 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

classmethod putNumberArray(*key*, *value*)
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

classmethod putRaw(*key*, *value*)
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

classmethod putString(*key*, *value*)
Put a string 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

classmethod putStringArray(*key*, *value*)
Put a string array in the table

Parameters

- **key** (*str*) – the key to be assigned to
- **value** (*list(str)*) – the value that will be assigned

Returns False if the table key already exists with a different type

Return type bool

classmethod setDefaultBoolean(*key*, *defaultValue*)
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 doens't exist.

Returns False if the table key exists with a different type

classmethod setDefaultBooleanArray(*key*, *defaultValue*)
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 doens't exist.

Returns False if the table key exists with a different type

classmethod setDefaultNumber(*key*, *defaultValue*)
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 doens't exist.

Returns False if the table key exists with a different type

classmethod setDefaultNumberArray(*key*, *defaultValue*)
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 doens't exist.

Returns False if the table key exists with a different type

classmethod setDefaultRaw(*key*, *defaultValue*)
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 doens't exist.

Returns False if the table key exists with a different type

classmethod setDefaultString(*key*, *defaultValue*)
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 doens't exist.

Returns False if the table key exists with a different type

classmethod setDefaultStringArray(*key*, *defaultValue*)
If the key doesn't currently exist, then the specified value will be assigned to the key.

Parameters

- **key** (*str*) – the key to be assigned to
- **defaultValue** (*list(str)*) – the default value to set if key doesn't exist.

Returns False if the table key exists with a different type

Return type bool

classmethod setFlags(*key*, *flags*)
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)

classmethod setPersistent(*key*)
Makes a key's value persistent through program restarts. The key cannot be null.

Parameters **key** – the key name

table = None

tablesToData = {}

Solenoid

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

Solenoid class for running high voltage Digital Output.

The Solenoid class is typically used for pneumatic solenoids, but could be used for any device within the current spec of the PCM.

Constructor.

Arguments can be supplied as positional or keyword. Acceptable positional argument combinations are:

•channel

•moduleNumber, channel

Alternatively, the above names can be used as keyword arguments.

Parameters

- **moduleNumber** (*int*) – The CAN ID of the PCM the solenoid is attached to
- **channel** (*int*) – The channel on the PCM to control (0..7)

free()
Mark the solenoid as freed.

get()
Read the current value of the solenoid.

Returns True if the solenoid output is on or false if the solenoid output is off.

Return type bool

isBlackListed()

Check if the 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*)
Set the value of a solenoid.

Parameters **on** (*bool*) – True will turn the solenoid output on. False will turn the solenoid output off.

solenoidHandle

SolenoidBase

class wpilib.**SolenoidBase**(*moduleNumber*)
Bases: *wpilib.SensorBase*

SolenoidBase class is the common base class for the Solenoid and DoubleSolenoid classes.

Constructor.

Parameters **moduleNumber** – The PCM CAN ID

clearAllPCMStickyFaults()
Clear ALL sticky faults inside the PCM that Solenoid is wired to.

If a sticky fault is set, then it will be persistently cleared. Compressor drive maybe momentarily disable while flages 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.

getAll()
Read all 8 solenoids from the module used by this solenoid as a single byte.

Returns The current value of all 8 solenoids on this module.

getPCMSolenoidBlackList()

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.

getPCMSolenoidVoltageFault()

Returns True if PCM is in fault state : The common highside solenoid voltage rail is too low, most likely a solenoid channel has been shorted.

getPCMSolenoidVoltageStickyFault()

Returns True if PCM Sticky fault is set : The common highside solenoid voltage rail is too low, most likely a solenoid channel has been shorted.

Spark

class wpilib.**Spark**(*channel*)
Bases: *wpilib.PWMSpeedController*

REV Robotics SPARK Speed Controller

Constructor.

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

---

Note: 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.

- 2.003ms = full "forward"
- 1.55ms = the "high end" of the deadband range
- 1.50ms = center of the deadband range (off)
- 1.46ms = the "low end" of the deadband range
- .999ms = full "reverse"

SPI

class wpilib.**SPI**(*port*, *simPort=None*)
Bases: object

Represents a SPI bus port

Example usage:

```
spi = wpilib.SPI(wpilib.SPI.Port.kOnboardCS0)

# Write bytes 'text', and receive something
data = spi.transaction(b'text')
```

Constructor

Parameters

- **port** (*SPI.Port*) – the physical SPI port
- **simPort** – This must be an object that implements all of the spi* functions from hal_impl that you use. See test_spi.py for an example.

class Port
Bases: object

kMXP = 4

kOnboardCS0 = 0

kOnboardCS1 = 1

kOnboardCS2 = 2

kOnboardCS3 = 3

SPI.**devices = 0**

SPI.**free**()

SPI.**freeAccumulator**()
Frees the accumulator.

SPI.**getAccumulatorAverage**()
Read the average of the accumulated value.

Returns The accumulated average value (value / count).

SPI.**getAccumulatorCount**()
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.

SPI.**getAccumulatorLastValue**()
Read the last value read by the accumulator engine.

SPI.**getAccumulatorOutput**()
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.

Returns tuple of (value, count)

SPI.**getAccumulatorValue**()
Read the accumulated value.

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

SPI.**initAccumulator**(*period*, *cmd*, *xfer_size*, *valid_mask*, *valid_value*, *data_shift*, *data_size*, *is_signed*, *big_endian*)
Initialize the accumulator.

Parameters

- **period** – Time between reads
- **cmd** – SPI command to send to request data
- **xfer_size** – SPI transfer size, in bytes
- **valid_mask** – Mask to apply to received data for validity checking
- **valid_data** – After valid_mask is applied, required matching value for validity checking
- **data_shift** – Bit shift to apply to received data to get actual data value
- **data_size** – Size (in bits) of data field
- **is_signed** – Is data field signed?
- **big_endian** – Is device big endian?

SPI.**port**

SPI.**read**(*initiate*, *size*)
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.
- **size** – Number of bytes to read.

Returns received data bytes

SPI.**resetAccumulator**()
Resets the accumulator to zero.

SPI.**setAccumulatorCenter**(*center*)
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.

SPI.**setAccumulatorDeadband**(*deadband*)
Set the accumulator's deadband.

SPI.**setChipSelectActiveHigh**()
Configure the chip select line to be active high.

SPI.**setChipSelectActiveLow**()
Configure the chip select line to be active low.

SPI.**setClockActiveHigh**()
Configure the clock output line to be active high. This is sometimes called clock polarity low or clock idle low.

SPI.**setClockActiveLow**()
Configure the clock output line to be active low. This is sometimes called clock polarity high or clock idle high.

SPI.**setClockRate**(*hz*)
Configure the rate of the generated clock signal. The default value is 500,000 Hz. The maximum value is 4,000,000 Hz.

Parameters **hz** – The clock rate in Hertz.

SPI.**setLSBFirst**()
Configure the order that bits are sent and received on the wire to be least significant bit first.

SPI.**setMSBFirst**()
Configure the order that bits are sent and received on the wire to be most significant bit first.

SPI.**setSampleDataOnFalling**()
Configure that the data is stable on the falling edge and the data changes on the rising edge.

SPI.**setSampleDataOnRising**()
Configure that the data is stable on the rising edge and the data changes on the falling edge.

SPI.**transaction**(*dataToSend*)
Perform a simultaneous read/write transaction with the device

Parameters **dataToSend** (*iterable of bytes*) – The data to be written out to the device

Returns data received from the device

Usage:

```
# send byte string
data = spi.transaction(b'stuff')

# send list of integers
data = spi.transaction([0x01, 0x02])
```

SPI.**write**(*dataToSend*)

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.

Parameters **dataToSend** (*iterable of bytes*) – Data to send

Returns Number of bytes written

Usage:

```
# send byte string
writeCount = spi.write(b'stuff')

# send list of integers
writeCount = spi.write([0x01, 0x02])
```

Talon

class wpilib.**Talon**(*channel*)

Bases: *wpilib.PWMSpeedController*

Cross the Road Electronics (CTRE) Talon and Talon SR Speed Controller via PWM

Constructor for a Talon (original or Talon SR)

Parameters **channel** (*int*) – The PWM channel that the Talon is attached to. 0-9 are on-board, 10-19 are on the MXP port

---

Note: 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"

TalonSRX

class wpilib.**TalonSRX**(*channel*)
Bases: *wpilib.PWMSpeedController*

Cross the Road Electronics (CTRE) Talon SRX Speed Controller via PWM

See also:

See *CANTalon* for CAN control of Talon SRX.

Constructor for a TalonSRX connected via PWM.

Parameters **channel** (*int*) – The PWM channel that the TalonSRX is attached to. 0-9 are on-board, 10-19 are on the MXP port.

Note: The TalonSRX 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 TalonSRX User Manual available from CTRE.

- 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"

Timer

class wpilib.**Timer**
Bases: object

Provides time-related functionality for the robot

Note: Prefer to use this module for time functions, instead of the time module in the standard library. This will make it easier for your code to work properly in simulation.

---

static **delay**(*seconds*)
Pause the thread for a specified time. Pause the execution of the thread 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 thread containing the wait will pause until the wait time is expired.

Parameters **seconds** (*float*) – Length of time to pause

Warning: If you're tempted to use this function for autonomous mode to time transitions between actions, don't do it!
Delaying the main robot thread for more than a few milliseconds is generally discouraged, and will cause problems and possibly leave the robot unresponsive.

get()
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

Return type float

static getFPGATimestamp()
Return the system clock time in seconds. Return the time from the FPGA hardware clock in seconds since the FPGA started.

Returns Robot running time in seconds.

Return type float

static getMatchTime()
Return the approximate match time. The FMS does not currently send the official match time to the robots. This returns the time since the enable signal sent from the Driver Station. At the beginning of autonomous, the time is reset to 0.0 seconds. At the beginning of teleop, the time is reset to +15.0 seconds. If the robot is disabled, this returns 0.0 seconds.

Warning: This is not an official time (so it cannot be used to argue with referees).

Returns Match time in seconds since the beginning of autonomous

Return type float

getMsClock()

Returns the system clock time in milliseconds.

Return type int

hasPeriodPassed(*period*)
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 (in seconds).

Returns If the period has passed.

Return type bool

reset()
Reset the timer by setting the time to 0. Make the timer startTime the current time so new requests will be relative now.

start()
Start the timer running. Just set the running flag to true indicating that all time requests should be relative to the system clock.

stop()
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.

Ultrasonic

class wpilib.**Ultrasonic**(*pingChannel*, *echoChannel*, *units=0*)
Bases: *wpilib.SensorBase*

Ultrasonic rangefinder control

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 emmitted. 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).

Create an instance of the Ultrasonic Sensor. This is designed to supchannel 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

class PIDSourceType
Bases: object

A description for the type of output value to provide to a *PIDController*

kDisplacement = 0

kRate = 1

class Ultrasonic.**Unit**
Bases: object

The units to return when PIDGet is called

kInches = 0

kMillimeters = 1

Ultrasonic.**automaticEnabled = False**
Automatic round robin mode

Ultrasonic.**free**()

Ultrasonic.**getDistanceUnits**()
Get the current DistanceUnit that is used for the PIDSource interface.

Returns The type of DistanceUnit that is being used.

Ultrasonic.**getPIDSourceType**()

Ultrasonic.**getRangeInches**()
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.

Return type float

Ultrasonic.**getRangeMM**()
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 complted, then return 0.

Return type float

Ultrasonic.**instances = 0**

static Ultrasonic.**isAutomaticMode**()

Ultrasonic.**isEnabled**()
Is the ultrasonic enabled.

Returns True if the ultrasonic is enabled

Ultrasonic.**isRangeValid**()
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.

Returns True if the range is valid

Return type bool

Ultrasonic.**kMaxUltrasonicTime = 0.1**
Max time (ms) between readings.

Ultrasonic.**kPingTime = 9.999999999999999e-06**
Time (sec) for the ping trigger pulse.

Ultrasonic.**kPriority = 90**
Priority that the ultrasonic round robin task runs.

Ultrasonic.**kSpeedOfSoundInchesPerSec = 13560.0**

Ultrasonic.**pidGet**()
Get the range in the current DistanceUnit (PIDSource interface).

Returns The range in DistanceUnit

Return type float

Ultrasonic.**ping**()
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.

Ultrasonic.**sensors = <_weakrefset.WeakSet object>**
ultrasonic sensor list

Ultrasonic.**setAutomaticMode**(*enabling*)
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** (*bool*) – 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 preferred, it can be implemented by pinging the sensors manually and waiting for the results to come back.

Ultrasonic.**setDistanceUnits**(*units*)
Set the current DistanceUnit that should be used for the PIDSource interface.

Parameters **units** – The DistanceUnit that should be used.

Ultrasonic.**setEnabled**(*enable*)
Set if the ultrasonic is enabled.

Parameters **enable** (*bool*) – set to True to enable the ultrasonic

Ultrasonic.**setPIDSourceType**(*pidSource*)
Set which parameter you are using as a process control variable.

Parameters **pidSource** (*PIDSource.PIDSourceType*) – An enum to select the parameter.

static Ultrasonic.**ultrasonicChecker**()
Background task that goes through the list of ultrasonic sensors and pings each one in turn. The counter is configured to read the timing of the returned echo pulse.

> **Warning:** DANGER WILL ROBINSON, DANGER WILL ROBINSON: This code runs as a task and assumes that none of the ultrasonic sensors will change while it's running. If one does, then this will certainly break. Make sure to disable automatic mode before changing anything with the sensors!!

Utility

class wpilib.**Utility**
Bases: object

Contains global utility functions

static **getFPGARevision**()
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.

Return type int

static **getFPGATime**()
Read the microsecond timer from the FPGA.

Returns The current time in microseconds according to the FPGA.

Return type int

static **getFPGAVersion**()
Return the FPGA Version number.

Returns FPGA Version number.

Return type int

static **getUserButton**()
Get the state of the "USER" button on the roboRIO.

Returns True if the button is currently pressed down

Return type bool

Victor

class wpilib.**Victor**(*channel*)
Bases: *wpilib.PWMSpeedController*

VEX Robotics Victor 888 Speed Controller via PWM

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: 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 also but if users experience issues such as asymmetric behaviour 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 Robotics: http://content.vexrobotics.com/docs/ifi-v884-users-manual-9-25-06.pdf

- 2.027ms = full "forward"
- 1.525ms = the "high end" of the deadband range
- 1.507ms = center of the deadband range (off)
- 1.49ms = the "low end" of the deadband range
- 1.026ms = full "reverse"

Constructor.

Parameters **channel** (*int*) – The PWM channel that the Victor is attached to. 0-9 are on-board, 10-19 are on the MXP port

VictorSP

class wpilib.**VictorSP**(*channel*)
Bases: *wpilib.PWMSpeedController*

VEX Robotics Victor SP Speed Controller via PWM

Constructor.

Parameters **channel** (*int*) – The PWM channel that the VictorSP is attached to. 0-9 are on-board, 10-19 are on the MXP port.

Note: 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 VictorSP User Manual.

- 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"

XboxController

class wpilib.**XboxController**(*port*)
Bases: *wpilib.interfaces.GamepadBase*

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 XBoxController. The XBoxController index is the USB port on the Driver Station.

Parameters **port** – The port on the Driver Station that the joystick is plugged into

getAButton()
Read the value of the A button on the controller

Returns The state of the A button

Return type boolean

getBButton()
Read the value of the B button on the controller

Returns The state of the B button

Return type boolean

getBackButton()
Read the value of the back button on the controller

Returns The state of the back button

Return type boolean

getBumper(*hand*)
Read the values of the bumper button on the controller.

Parameters **hand** – Side of controller whose value should be returned.

Returns The state of the button

Return type boolean

getName()

getPOV(*pov*)

getPOVCount()

getRawAxis(*axis*)
Get the value of the axis

Parameters **axis** – The axis to read, starting at 0

Returns The value of the axis

Return type float

getRawButton(*button*)
Get the buttom value (starting at button 1)

Parameters **button** – The button number to be read (starting at 1)

Returns The state of the button

Return type boolean

getStartButton()
Read the value of the start button on the controller

Returns The state of the start button

Return type boolean

getStickButton(*hand*)
Read the values of the stick button on the controller

Parameters **hand** – Side of the controller whose value should be returned

Returns The state of the button

Return type boolean

getTriggerAxis(*hand*)
Get the trigger axis value of the controller.

Parameters **hand** – Side of controller whose value should be returned

Returns The trigger axis value of the controller

Return type float

getType()

getX(*hand*)
Get the X axis value of the controller.

Parameters **hand** – Side of controller whose value should be returned

Returns The X axis value of the controller

Return type float

getXButton()
Read the value of the X button on the controller

Returns The state of the X button

Return type boolean

getY(*hand*)
Get the Y axis value of the controller.

Parameters **hand** – Side of controller whose value should be returned

Returns The Y axis value of the controller

Return type float

getYButton()
Read the value of the Y button on the controller

Returns The state of the Y button

Return type boolean

setOutput(*outputNumber*, *value*)

setOutputs(*value*)

setRumble(*type_*, *value*)

wpilib.buttons Package

Classes in this package are used to interface various types of buttons to a command-based robot.

If you are not using the Command framework, you can ignore these classes.

Continued on next page

Table 1.2 – continued from previous page

wpilib.buttons.Button	This class provides an easy way to link commands to OI inputs.
wpilib.buttons.InternalButton([...])	This class is intended to be used within a program.
wpilib.buttons.JoystickButton(...)	Create a joystick button for triggering commands.
wpilib.buttons.NetworkButton(...)	
wpilib.buttons.Trigger	This class provides an easy way to link commands to inputs.

Button

class wpilib.buttons.**Button**
Bases: *wpilib.buttons.Trigger*

This class provides an easy way to link commands to OI inputs.

It is very easy to link a button to a command. For instance, you could link the trigger button of a joystick to a "score" command.

This class represents a subclass of *Trigger* that is specifically aimed at buttons on an operator interface as a common use case of the more generalized Trigger objects. This is a simple wrapper around Trigger with the method names renamed to fit the Button object use.

cancelWhenPressed(*command*)
Cancel the command when the button is pressed.

Parameters **command** –

toggleWhenPressed(*command*)
Toggles the command whenever the button is pressed (on then off then on).

Parameters **command** –

whenPressed(*command*)
Starts the given command whenever the button is newly pressed.

Parameters **command** – the command to start

whenReleased(*command*)
Starts the command when the button is released.

Parameters **command** – the command to start

whileHeld(*command*)
Constantly starts the given command while the button is held.

Command.start() will be called repeatedly while the button is held, and will be canceled when the button is released.

Parameters **command** – the command to start

InternalButton

class wpilib.buttons.**InternalButton**(*inverted=False*)
Bases: *wpilib.buttons.Button*

This class is intended to be used within a program. The programmer can manually set its value. Includes a setting for whether or not it should invert its value.

Creates an InternalButton which is inverted depending on the input.

Parameters **inverted** – If False, then this button is pressed when set to True, otherwise it is pressed when set to False.

get()

setInverted(*inverted*)

setPressed(*pressed*)

JoystickButton

class wpilib.buttons.**JoystickButton**(*joystick*, *buttonNumber*)
Bases: *wpilib.buttons.Button*

Create a joystick button for triggering commands.

Parameters

- **joystick** – The GenericHID object that has the button (e.g. *Joystick*, KinectStick, etc)
- **buttonNumber** – The button number (see GenericHID.getRawButton())

get()
Gets the value of the joystick button.

Returns The value of the joystick button

NetworkButton

class wpilib.buttons.**NetworkButton**(*table*, *field*)
Bases: *wpilib.buttons.Button*

get()

Trigger

class wpilib.buttons.**Trigger**
Bases: object

This class provides an easy way to link commands to inputs.

It is very easy to link a button to a command. For instance, you could link the trigger button of a joystick to a "score" command.

It is encouraged that teams write a subclass of Trigger if they want to have something unusual (for instance, if they want to react to the user holding a button while the robot is reading a certain sensor input). For this, they only have to write the *get()* method to get the full functionality of the Trigger class.

cancelWhenActive(*command*)
Cancels a command when the trigger becomes active.

Parameters **command** – the command to cancel

get()
Returns whether or not the trigger is active

This method will be called repeatedly a command is linked to the Trigger.

Returns whether or not the trigger condition is active.

grab()
Returns whether *get()* returns True or the internal table for *SmartDashboard* use is pressed.

toggleWhenActive(*command*)
Toggles a command when the trigger becomes active.

Parameters **command** – the command to toggle

whenActive(*command*)
Starts the given command whenever the trigger just becomes active.

Parameters **command** – the command to start

whenInactive(*command*)
Starts the command when the trigger becomes inactive.

Parameters **command** – the command to start

whileActive(*command*)
Constantly starts the given command while the button is held.

`Command.start()` will be called repeatedly while the trigger is active, and will be canceled when the trigger becomes inactive.

Parameters **command** – the command to start

wpilib.command Package

Objects in this package allow you to implement a robot using Command-based programming. Command based programming is a design pattern to help you organize your robot programs, by organizing your robot program into components based on *Command* and *Subsystem*

The python implementation of the Command framework closely follows the Java language implementation. RobotPy has several examples of command based robots available.

Each one of the objects in the Command framework has detailed documentation available. If you need more information, for examples, tutorials, and other detailed information on programming your robot using this pattern, we recommend that you consult the Java version of the FRC Control System documentation

wpilib.command.Command([name, timeout])	The Command class is at the very core of the entire command framework.
wpilib.command.CommandGroup([name])	A CommandGroup is a list of commands which are executed in sequence.
wpilib.command.ConditionalCommand(name)	A ConditionalCommand is a *Command* that starts one of two commands.
wpilib.command.InstantCommand([name])	A command that has no duration.
wpilib.command.PIDCommand(p, i, d)	This class defines a Command which interacts heavily with a PID loop.
wpilib.command.PIDSubsystem(p, i, d)	This class is designed to handle the case where there is a Subsystem which
wpilib.command.PrintCommand(message)	A PrintCommand is a command which prints out a string when it is initiali
wpilib.command.Scheduler()	The Scheduler is a singleton which holds the top-level running commands.
wpilib.command.StartCommand(...)	A StartCommand will call the start() method of another command when it
wpilib.command.Subsystem([name])	This class defines a major component of the robot.
wpilib.command.TimedCommand(...)	A command that runs for a set period of time.
wpilib.command.WaitCommand(timeout)	A WaitCommand will wait for a certain amount of time before finishing.
wpilib.command.WaitForChildren([...])	This command will only finish if whatever *CommandGroup* it is in has no
wpilib.command.WaitUntilCommand(time)	This will wait until the game clock reaches some value, then continue to th

Command

class wpilib.command.**Command**(*name=None*, *timeout=None*)
Bases: *wpilib.Sendable*

The Command class is at the very core of the entire command framework. Every command can be started with a call to start(). Once a command is started it will call *initialize()*, and then will repeatedly call *execute()* until *isFinished()* returns True. Once it does, *end()* will be called.

However, if at any point while it is running *cancel()* is called, then the command will be stopped and *interrupted()* will be called.

If a command uses a *Subsystem*, then it should specify that it does so by calling the *requires()* method in its constructor. Note that a Command may have multiple requirements, and *requires()* should be called for each one.

If a command is running and a new command with shared requirements is started, then one of two things will happen. If the active command is interruptible, then *cancel()* will be called and the command will be removed to make way for the new one. If the active command is not interruptible, the other one will not even be started, and the active one will continue functioning.

See also:

Subsystem, *CommandGroup*

Creates a new command.

Parameters

- **name** – The name for this command; if unspecified or None, The name of this command will be set to its class name.
- **timeout** – The time (in seconds) before this command "times out". Default is no timeout. See isTimedOut().

cancel()
This will cancel the current command.

This will cancel the current command eventually. It can be called multiple times. And it can be called when the command is not running. If the command is running though, then the command will be marked as canceled and eventually removed.

> **Warning:** A command can not be canceled if it is a part of a *CommandGroup*, you must cancel the CommandGroup instead.

clearRequirements()
Clears list of subsystem requirements. This is only used by *ConditionalCommand* so cancelling the chosen command works properly in *CommandGroup*.

doesRequire(*system*)
Checks if the command requires the given *Subsystem*.

Parameters **system** – the system

Returns whether or not the subsystem is required, or False if given None.

end()
Called when the command ended peacefully. This is where you may want to wrap up loose ends, like shutting off a motor that was being used in the command.

execute()
The execute method is called repeatedly until this Command either finishes or is canceled.

getGroup()
Returns the *CommandGroup* that this command is a part of. Will return None if this Command is not in a group.

Returns the *CommandGroup* that this command is a part of (or None if not in group)

getName()
Returns the name of this command. If no name was specified in the constructor, then the default is the name of the class.

Returns the name of this command

getRequirements()
Returns the requirements (as a set of Subsystems) of this command

initialize()
The initialize method is called the first time this Command is run after being started.

interrupted()
Called when the command ends because somebody called cancel() or another command shared the same requirements as this one, and booted it out.

This is where you may want to wrap up loose ends, like shutting off a motor that was being used in the command.

Generally, it is useful to simply call the end() method within this method, as done here.

isCanceled()
Returns whether or not this has been canceled.

Returns whether or not this has been canceled

isFinished()
Returns whether this command is finished. If it is, then the command will be removed and end() will be called.

It may be useful for a team to reference the isTimedOut() method for time-sensitive commands, or override TimedCommand.

If you do not specify isFinished in your command, the command will only end if interrupted or canceled. If you want a command that executes only once and then ends, override InstantCommand.

Returns whether this command is finished.

See *isTimedOut()*

See

class *.TimedCommand*

See

class *.InstantCommand*

isInterruptible()
Returns whether or not this command can be interrupted.

Returns whether or not this command can be interrupted

isRunning()
Returns whether or not the command is running. This may return true even if the command has just been canceled, as it may not have yet called *interrupted()*.

Returns whether or not the command is running

isTimedOut()
Returns whether or not the *timeSinceInitialized()* method returns a number which is greater than or equal to the timeout for the command. If there is no timeout, this will always return false.

Returns whether the time has expired

lockChanges()
Prevents further changes from being made

removed()
Called when the command has been removed. This will call *interrupted()* or *end()*.

requires(*subsystem*)
This method specifies that the given Subsystem is used by this command. This method is crucial to the functioning of the Command System in general.

Note that the recommended way to call this method is in the constructor.

Parameters **subsystem** – the *Subsystem* required

run()
The run method is used internally to actually run the commands.

Returns whether or not the command should stay within the Scheduler.

setInterruptible(*interruptible*)
Sets whether or not this command can be interrupted.

Parameters **interruptible** – whether or not this command can be interrupted

setParent(*parent*)
Sets the parent of this command. No actual change is made to the group.

Parameters **parent** – the parent

setRunWhenDisabled(*run*)
Sets whether or not this {@link Command} should run when the robot is disabled.

By default a command will not run when the robot is disabled, and will in fact be canceled.

Parameters **run** – whether or not this command should run when the robot is disabled

setTimeout(*seconds*)
Sets the timeout of this command.

Parameters **seconds** – the timeout (in seconds)

See *isTimedOut()*

start()
Starts up the command. Gets the command ready to start. Note that the command will eventually start, however it will not necessarily do so immediately, and may in fact be canceled before initialize is even called.

startRunning()
This is used internally to mark that the command has been started. The lifecycle of a command is:

- *startRunning()* is called.
- *run()* is called (multiple times potentially)
- *removed()* is called

It is very important that *startRunning()* and *removed()* be called in order or some assumptions of the code will be broken.

startTiming()
Called to indicate that the timer should start. This is called right before initialize() is, inside the run() method.

timeSinceInitialized()
Returns the time since this command was initialized (in seconds). This function will work even if there is no specified timeout.

Returns the time since this command was initialized (in seconds).

willRunWhenDisabled()
Returns whether or not this Command will run when the robot is disabled, or if it will cancel itself.

CommandGroup

class wpilib.command.**CommandGroup**(*name=None*)
Bases: *wpilib.command.Command*

A CommandGroup is a list of commands which are executed in sequence.

Commands in a CommandGroup are added using the *addSequential()* method and are called sequentially. CommandGroups are themselves Commands and can be given to other CommandGroups.

CommandGroups will carry all of the requirements of their subcommands. Additional requirements can be specified by calling requires() normally in the constructor.

CommandGroups can also execute commands in parallel, simply by adding them using addParallel(...).

See also:

Command, Subsystem

Creates a new CommandGroup with the given name.

Parameters **name** – the name for this command group (optional). If None, the name of this command will be set to its class name.

class Entry(*command*, *state*, *timeout*)
Bases: object

BRANCH_CHILD = 2

BRANCH_PEER = 1

IN_SEQUENCE = 0

isTimedOut()

CommandGroup.**addParallel**(*command*, *timeout=None*)
Adds a new child Command to the group (with an optional timeout). The Command will be started after all the previously added Commands.

Once the Command is started, it will run until it finishes, is interrupted, or the time expires (if a timeout is provided), whichever is sooner. Note that the given Command will have no knowledge that it is on a timer.

Instead of waiting for the child to finish, a CommandGroup will have it run at the same time as the subsequent Commands. The child will run until either it finishes, the timeout expires, a new child with conflicting requirements is started, or the main sequence runs a Command with conflicting requirements. In the latter two cases, the child will be canceled even if it says it can't be interrupted.

Note that any requirements the given Command has will be added to the group. For this reason, a Command's requirements can not be changed after being added to a group.

It is recommended that this method be called in the constructor.

Parameters

- **command** – The command to be added
- **timeout** – The timeout (in seconds) (optional)

CommandGroup.**addSequential**(*command*, *timeout=None*)

Adds a new Command to the group (with an optional timeout). The Command will be started after all the previously added Commands.

Once the Command is started, it will be run until it finishes or the time expires, whichever is sooner (if a timeout is provided). Note that the given Command will have no knowledge that it is on a timer.

Note that any requirements the given Command has will be added to the group. For this reason, a Command's requirements can not be changed after being added to a group.

It is recommended that this method be called in the constructor.

Parameters

- **command** – The Command to be added
- **timeout** – The timeout (in seconds) (optional)

CommandGroup.**cancelConflicts**(*command*)

CommandGroup.**end**()

CommandGroup.**execute**()

CommandGroup.**initialize**()

CommandGroup.**interrupted**()

CommandGroup.**isFinished**()

Returns True if all the Commands in this group have been started and have finished.

Teams may override this method, although they should probably reference super().isFinished() if they do.

Returns whether this CommandGroup is finished

CommandGroup.**isInterruptible**()

Returns whether or not this group is interruptible. A command group will be uninterruptible if setInterruptable(False) was called or if it is currently running an uninterruptible command or child.

Returns whether or not this CommandGroup is interruptible.

ConditionalCommand

class wpilib.command.**ConditionalCommand**(*name*, *onTrue=None*, *onFalse=None*)

Bases: *wpilib.command.Command*

A ConditionalCommand is a *Command* that starts one of two commands.

A ConditionalCommand uses m_condition to determine whether it should run m_onTrue or m_onFalse.

A ConditionalCommand adds the proper *Command* to the *Scheduler* during `initialize()` and then *isFinished()* will return true once that *Command* has finished executing.

If no *Command* is specified for m_onFalse, the occurrence of that condition will be a no-op.

@see *Command* @see *Scheduler*

Creates a new ConditionalCommand with given name and onTrue and onFalse Commands.

Users of this constructor should also override condition().

Parameters

- **name** – the name for this command group
- **onTrue** – The Command to execute if {@link ConditionalCommand#condition()} returns true
- **onFalse** – The Command to execute if {@link ConditionalCommand#condition()} returns false

condition()
The Condition to test to determine which Command to run.

Returns true if m_onTrue should be run or false if m_onFalse should be run.

interrupted()

isFinished()

InstantCommand

class wpilib.command.**InstantCommand**(*name=None*)
Bases: *wpilib.command.Command*

A command that has no duration. Subclasses should implement the initialize() method to carry out desired actions.

isFinished()

PIDCommand

class wpilib.command.**PIDCommand**(*p*, *i*, *d*, *period=None*, *f=0.0*, *name=None*)
Bases: *wpilib.command.Command*

This class defines a Command which interacts heavily with a PID loop.

It provides some convenience methods to run an internal PIDController. It will also start and stop said PIDController when the PIDCommand is first initialized and ended/interrupted.

Instantiates a PIDCommand that will use the given p, i and d values. It will use the class name as its name unless otherwise specified. It will also space the time between PID loop calculations to be equal to the given period.

Parameters

- **p** – the proportional value
- **i** – the integral value
- **d** – the derivative value
- **period** – the time (in seconds) between calculations (optional)
- **f** – the feed forward value
- **name** – the name (optional)

getPIDController()
Returns the PIDController used by this PIDCommand. Use this if you would like to fine tune the pid loop.

Notice that calling setSetpoint(...) on the controller will not result in the setpoint being trimmed to be in the range defined by setSetpointRange(...).

Returns the PIDController used by this PIDCommand

getPosition()
Returns the current position

Returns the current position

getSetpoint()
Returns the setpoint.

Returns the setpoint

returnPIDInput()
Returns the input for the pid loop.

It returns the input for the pid loop, so if this command was based off of a gyro, then it should return the angle of the gyro

All subclasses of PIDCommand must override this method.

This method will be called in a different thread then the *Scheduler* thread.

Returns the value the pid loop should use as input

setSetpoint(*setpoint*)
Sets the setpoint to the given value. If `setRange()` was called, then the given setpoint will be trimmed to fit within the range.

Parameters **setpoint** – the new setpoint

setSetpointRelative(*deltaSetpoint*)
Adds the given value to the setpoint. If `setRange()` was used, then the bounds will still be honored by this method.

Parameters **deltaSetpoint** – the change in the setpoint

usePIDOutput(*output*)
Uses the value that the pid loop calculated. The calculated value is the "output" parameter. This method is a good time to set motor values, maybe something along the lines of *driveline.tankDrive(output, -output)*.

All subclasses of PIDCommand should override this method.

This method will be called in a different thread then the Scheduler thread.

Parameters **output** – the value the pid loop calculated

PIDSubsystem

class wpilib.command.**PIDSubsystem**(*p*, *i*, *d*, *period=None*, *f=0.0*, *name=None*)
Bases: *wpilib.command.Subsystem*

This class is designed to handle the case where there is a Subsystem which uses a single {@link PIDController} almost constantly (for instance, an elevator which attempts to stay at a constant height).

It provides some convenience methods to run an internal PIDController. It also allows access to the internal PIDController in order to give total control to the programmer.

Instantiates a PIDSubsystem that will use the given p, i and d values. It will use the class name as its name unless otherwise specified. It will also space the time between PID loop calculations to be equal to the given period.

Parameters

- **p** – the proportional value
- **i** – the integral value

- **d** – the derivative value
- **period** – the time (in seconds) between calculations (optional)
- **f** – the feed forward value
- **name** – the name (optional)

disable()
Disables the internal *PIDController*

enable()
Enables the internal *PIDController*

getPIDController()
Returns the PIDController used by this PIDSubsystem. Use this if you would like to fine tune the pid loop.

Notice that calling *setSetpoint()* on the controller will not result in the setpoint being trimmed to be in the range defined by `setSetpointRange()`.

Returns the *PIDController* used by this PIDSubsystem

getPosition()
Returns the current position

Returns the current position

getSetpoint()
Returns the setpoint.

Returns the setpoint

onTarget()
Return True if the error is within the percentage of the total input range, determined by setAbsoluteTolerance or setPercentTolerance. This assumes that the maximum and minimum input were set using setInput.

Returns True if the error is less than the tolerance

returnPIDInput()
Returns the input for the pid loop.

It returns the input for the pid loop, so if this command was based off of a gyro, then it should return the angle of the gyro

All subclasses of PIDSubsystem must override this method.

This method will be called in a different thread then the Scheduler thread.

Returns the value the pid loop should use as input

setAbsoluteTolerance(*t*)
Set the absolute error which is considered tolerable for use with OnTarget.

Parameters **t** – The absolute tolerance (same range as the PIDInput values)

setInputRange(*minimumInput*, *maximumInput*)
Sets the maximum and minimum values expected from the input.

Parameters

- **minimumInput** – the minimum value expected from the input
- **maximumInput** – the maximum value expected from the output

setOutputRange(*minimumOutput*, *maximumOutput*)
Sets the maximum and minimum values to write.

Parameters

- **minimumOutput** – the minimum value to write to the output
- **maximumOutput** – the maximum value to write to the output

setPercentTolerance(*p*)
Set the percentage error which is considered tolerable for use with OnTarget.

Parameters **p** – The percentage tolerance (value of 15.0 == 15 percent)

setSetpoint(*setpoint*)
Sets the setpoint to the given value. If `setRange()` was called, then the given setpoint will be trimmed to fit within the range.

Parameters **setpoint** – the new setpoint

setSetpointRelative(*deltaSetpoint*)
Adds the given value to the setpoint. If `setRange()` was used, then the bounds will still be honored by this method.

Parameters **deltaSetpoint** – the change in the setpoint

usePIDOutput(*output*)
Uses the value that the pid loop calculated. The calculated value is the "output" parameter. This method is a good time to set motor values, maybe something along the lines of *driveline.tankDrive(output, -output)*.

All subclasses of PIDSubsystem should override this method.

This method will be called in a different thread then the Scheduler thread.

Parameters **output** – the value the pid loop calculated

PrintCommand

class wpilib.command.**PrintCommand**(*message*)
Bases: *wpilib.command.InstantCommand*

A PrintCommand is a command which prints out a string when it is initialized, and then immediately finishes.

It is useful if you want a *CommandGroup* to print out a string when it reaches a certain point.

Instantiates a PrintCommand which will print the given message when it is run.

Parameters **message** – the message to print

initialize()

Scheduler

class wpilib.command.**Scheduler**
Bases: *wpilib.Sendable*

The Scheduler is a singleton which holds the top-level running commands. It is in charge of both calling the command's run() method and to make sure that there are no two commands with conflicting requirements running.

It is fine if teams wish to take control of the Scheduler themselves, all that needs to be done is to call Scheduler.getInstance().run() often to have Commands function correctly. However, this is already done for you if you use the CommandBased Robot template.

See also:

Command

Instantiates a Scheduler.

add(*command*)
Adds the command to the Scheduler. This will not add the *Command* immediately, but will instead wait for the proper time in the *run()* loop before doing so. The command returns immediately and does nothing if given null.

Adding a *Command* to the *Scheduler* involves the Scheduler removing any Command which has shared requirements.

Parameters **command** – the command to add

addButton(*button*)
Adds a button to the Scheduler. The Scheduler will poll the button during its *run()*.

Parameters **button** – the button to add

disable()
Disable the command scheduler.

enable()
Enable the command scheduler.

static getInstance()
Returns the Scheduler, creating it if one does not exist.

Returns the Scheduler

getName()

getType()

registerSubsystem(*system*)
Registers a *Subsystem* to this Scheduler, so that the Scheduler might know if a default Command needs to be run. All *Subsystem* objects should call this.

Parameters **system** – the system

remove(*command*)
Removes the *Command* from the Scheduler.

Parameters **command** – the command to remove

removeAll()
Removes all commands

run()
Runs a single iteration of the loop. This method should be called often in order to have a functioning Command system. The loop has five stages:

•Poll the Buttons

•Execute/Remove the Commands

•Send values to SmartDashboard

•Add Commands

•Add Defaults

StartCommand

class wpilib.command.**StartCommand**(*commandToStart*)
Bases: *wpilib.command.InstantCommand*

A StartCommand will call the start() method of another command when it is initialized and will finish immediately.

Instantiates a StartCommand which will start the given command whenever its initialize() is called.

Parameters **commandToStart** – the *Command* to start

initialize()

Subsystem

class wpilib.command.**Subsystem**(*name=None*)
Bases: *wpilib.Sendable*

This class defines a major component of the robot.

A good example of a subsystem is the driveline, or a claw if the robot has one.

All motors should be a part of a subsystem. For instance, all the wheel motors should be a part of some kind of "Driveline" subsystem.

Subsystems are used within the command system as requirements for Command. Only one command which requires a subsystem can run at a time. Also, subsystems can have default commands which are started if there is no command running which requires this subsystem.

See also:

Command

Creates a subsystem.

Parameters **name** – the name of the subsystem; if None, it will be set to the name to the name of the class.

confirmCommand()
Call this to alert Subsystem that the current command is actually the command. Sometimes, the Subsystem is told that it has no command while the Scheduler is going through the loop, only to be soon after given a new one. This will avoid that situation.

getCurrentCommand()
Returns the command which currently claims this subsystem.

Returns the command which currently claims this subsystem

getDefaultCommand()
Returns the default command (or None if there is none).

Returns the default command

getName()
Returns the name of this subsystem, which is by default the class name.

Returns the name of this subsystem

initDefaultCommand()
Initialize the default command for a subsystem By default subsystems have no default command, but if they do, the default command is set with this method. It is called on all Subsystems by CommandBase in the users program after all the Subsystems are created.

setCurrentCommand(*command*)
Sets the current command

Parameters **command** – the new current command

setDefaultCommand(*command*)
Sets the default command. If this is not called or is called with None, then there will be no default command for the subsystem.

Parameters **command** – the default command (or None if there should be none)

Warning: This should NOT be called in a constructor if the subsystem is a singleton.

TimedCommand

class wpilib.command.**TimedCommand**(*name*, *timeoutInSeconds*)
Bases: *wpilib.command.Command*

A command that runs for a set period of time.

isFinished()

WaitCommand

class wpilib.command.**WaitCommand**(*timeout*, *name=None*)
Bases: *wpilib.command.TimedCommand*

A WaitCommand will wait for a certain amount of time before finishing. It is useful if you want a *CommandGroup* to pause for a moment.

See also:

CommandGroup

Instantiates a WaitCommand with the given timeout.

Parameters

- **timeout** – the time the command takes to run
- **name** – the name of the command (optional)

WaitForChildren

class wpilib.command.**WaitForChildren**(*name=None*, *timeout=None*)
Bases: *wpilib.command.Command*

This command will only finish if whatever *CommandGroup* it is in has no active children. If it is not a part of a CommandGroup, then it will finish immediately. If it is itself an active child, then the CommandGroup will never end.

This class is useful for the situation where you want to allow anything running in parallel to finish, before continuing in the main CommandGroup sequence.

Creates a new command.

Parameters

- **name** – The name for this command; if unspecified or None, The name of this command will be set to its class name.
- **timeout** – The time (in seconds) before this command "times out". Default is no timeout. See isTimedOut().

isFinished()

WaitUntilCommand

class wpilib.command.**WaitUntilCommand**(*time*)

Bases: *wpilib.command.Command*

This will wait until the game clock reaches some value, then continue to the next command.

isFinished()

wpilib.interfaces Package

This package contains objects that can be used to determine the requirements of various interfaces used in WPILib.

Generally, the python version of WPILib does not require that you inherit from any of these interfaces, but instead will allow you to use custom objects as long as they have the same methods.

wpilib.interfaces.Accelerometer	Interface for 3-axis accelerometers
wpilib.interfaces.Controller	An interface for controllers.
wpilib.interfaces.CounterBase	Interface for counting the number of ticks on a digital input channel.
wpilib.interfaces.GenericHID(port)	GenericHID Interface.
wpilib.interfaces.Gyro	Interface for yaw rate gyros
wpilib.interfaces.NamedSendable	The interface for sendable objects that gives the sendable a default name in the Sr
wpilib.interfaces.PIDInterface	
wpilib.interfaces.PIDOutput	This interface allows *PIDController* to write its results to its output.
wpilib.interfaces.PIDSource	This interface allows for *PIDController* to automatically read from this objec
wpilib.interfaces.Potentiometer	
wpilib.interfaces.SpeedController	Interface for speed controlling devices.

Accelerometer

class wpilib.interfaces.**Accelerometer**

Bases: object

Interface for 3-axis accelerometers

class Range

Bases: object

k16G = 3

k2G = 0

k4G = 1

k8G = 2

Accelerometer.**getX**()

Common interface for getting the x axis acceleration

Returns The acceleration along the x axis in g-forces

`Accelerometer.`**`getY`**`()`
Common interface for getting the y axis acceleration

Returns The acceleration along the y axis in g-forces

`Accelerometer.`**`getZ`**`()`
Common interface for getting the z axis acceleration

Returns The acceleration along the z axis in g-forces

`Accelerometer.`**`setRange`**`(`*range*`)`
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.

Controller

class `wpilib.interfaces.`**`Controller`**
Bases: `object`

An interface for controllers. Controllers run control loops, the most command are PID controllers and there variants, but this includes anything that is controlling an actuator in a separate thread.

`disable``()`
Stops the control loop from running until explicitly re-enabled by calling *`enable()`*.

`enable``()`
Allows the control loop to run.

CounterBase

class `wpilib.interfaces.`**`CounterBase`**
Bases: `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`
Bases: `object`

The number of edges for the counterbase to increment or decrement on

`k1X = 0`
Count only the rising edge

`k2X = 1`
Count both the rising and falling edge

`k4X = 2`
Count rising and falling on both channels

`CounterBase.`**`get`**`()`
Get the count

Returns the count

CounterBase.**getDirection**()
Determine which direction the counter is going

Returns True for one direction, False for the other

CounterBase.**getPeriod**()
Get the time between the last two edges counted

Returns the time beteween the last two ticks in seconds

CounterBase.**getStopped**()
Determine if the counter is not moving

Returns True if the counter has not changed for the max period

CounterBase.**reset**()
Reset the count to zero

CounterBase.**setMaxPeriod**(*maxPeriod*)
Set the maximum time between edges to be considered stalled

Parameters **maxPeriod** – the maximum period in seconds

GenericHID

class wpilib.interfaces.**GenericHID**(*port*)
Bases: object

GenericHID Interface.

class **HIDType**(*value*)
Bases: object

kHID1stPerson = 24

kHIDDriving = 22

kHIDFlight = 23

kHIDGamepad = 21

kHIDJoystick = 20

kUnknown = -1

kXInputArcadePad = 19

kXInputArcadeStick = 3

kXInputDancePad = 5

kXInputDrumKit = 8

kXInputFlightStick = 4

kXInputGamepad = 1

kXInputGuitar = 6

kXInputGuitar2 = 7

kXInputGuitar3 = 11

kXInputUnknown = 0

kXInputWheel = 2

class GenericHID.**Hand**
Bases: object

Which hand the Human Interface Device is associated with.

kLeft = 0
Left Hand

kRight = 1
Right Hand

class GenericHID.**RumbleType**
Bases: object

Represents a rumble output on the JoyStick.

kLeftRumble = 0
Left Hand

kRightRumble = 1
Right Hand

GenericHID.**getName**()
Get the name of the HID.

Returns the name of the HID.

GenericHID.**getPOV**(*pov=0*)
Get the angle in degrees of a POV on the HID.

The POV angles start at 0 in the up direction, and increase clockwise (eg 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.

GenericHID.**getPOVCount**()
For the current HID, return the number of POVs.

GenericHID.**getPort**()
Get the port number of the HID.

Returns The port number of the HID.

GenericHID.**getRawAxis**(*which*)
Get the raw axis.

Parameters **which** – index of the axis

Returns the raw value of the selected axis

GenericHID.**getRawButton**(*button*)
Is the given button pressed.

Parameters **button** – which button number

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

GenericHID.**getType**()
Get the type of the HID.

Returns the type of the HID.

GenericHID.**getX**(*hand=None*)
Get the x position of HID.

Parameters **hand** – which hand, left or right

Returns the x position

GenericHID.**getY**(*hand=None*)
Get the y position of the HID.

Parameters **hand** – which hand, left or right

Returns the y position

GenericHID.**setOutput**(*outputNumber*, *value*)
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

GenericHID.**setOutputs**(*value*)
Set all HID output values for the HID.

Parameters **value** – The 32 bit output value (1 bit for each output)

GenericHID.**setRumble**(*type*, *value*)
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

Gyro

class wpilib.interfaces.**Gyro**
Bases: object

Interface for yaw rate gyros

calibrate()
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.

Note: Usually you don't need to call this, as it's called when the object is first created. If you do, it will freeze your robot for 5 seconds

---

free()
Free the resources used by the gyro

getAngle()
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.

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

getRate()
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()
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.

NamedSendable

class wpilib.interfaces.**NamedSendable**
Bases: *wpilib.Sendable*

The interface for sendable objects that gives the sendable a default name in the Smart Dashboard.

getName()

Returns The name of the subtable of SmartDashboard that the *Sendable* object will use

PIDInterface

class wpilib.interfaces.**PIDInterface**
Bases: *wpilib.interfaces.Controller*

disable()

enable()

getD()

getError()

getI()

getP()

getSetpoint()

isEnabled()

reset()

setPID(*p*, *i*, *d*)

setSetpoint(*setpoint*)

PIDOutput

class wpilib.interfaces.**PIDOutput**
Bases: object

This interface allows *PIDController* to write its results to its output.

pidWrite(*output*)
Set the output to the value calculated by *PIDController*.

Parameters **output** – the value calculated by PIDController

PIDSource

class wpilib.interfaces.**PIDSource**
Bases: object

This interface allows for *PIDController* to automatically read from this object.

class **PIDSourceType**
Bases: object

A description for the type of output value to provide to a *PIDController*

kDisplacement = 0

kRate = 1

static PIDSource.**from_obj_or_callable**(*objc*)
Utility method that gets a PIDSource object

Parameters **objc** – An object that implements the PIDSource interface, or a callable

Returns an object that implements the PIDSource interface

PIDSource.**getPIDSourceType**()

Get which parameter of the device you are using as a process control variable.

Returns the currently selected PID source parameter

PIDSource.**pidGet**()
Get the result to use in *PIDController*

Returns the result to use in PIDController

PIDSource.**setPIDSourceType**(*pidSource*)
Set which parameter of the device you are using as a process control variable.

Parameters **pidSource** (*PIDSourceType*) – An enum to select the parameter.

Potentiometer

class wpilib.interfaces.**Potentiometer**
Bases: *wpilib.interfaces.PIDSource*

get()

SpeedController

class wpilib.interfaces.**SpeedController**
Bases: *wpilib.interfaces.PIDOutput*

Interface for speed controlling devices.

disable()
Disable the speed controller.

get()
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()
Common interface for determining if a speed controller is in the inverted state or not.

Returns True if in inverted state

set(*speed*)
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*)
Common interface for inverting direction of a speed controller.

Parameters **isInverted** – The state of inversion

stopMotor()
Stops motor movement. Motor can be moved again by calling set without having to re-enable the motor.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

W

A

B

C

D

E

F

G

H

I

J

K

L

M

S

T

U

V

W

X