. Documentation

Release 2020.2.0.post0.dev6

Author

Jan 13, 2021

ROBOT PROGRAMMING

1 Utilities API **3**

1.1 commandbased module . . . 3
1.2 magicbot module . . . 6
1.3 robotpy_ext.autonomous package . . . 17
1.4 robotpy_ext.common_drivers package . . . 22
1.5 robotpy_ext.control package . . . 24
1.6 robotpy_ext.misc package . . . 25

2 Indices and tables **31**

Python Module Index **33**

Index **35**

This project is intended to be a common place for high quality code to live for "things that should be in WPILib, but aren't". The python implementation of WPILib is intended to be very close to the implementations in the other languages, which is where packages like this come in.

- Most anything will be accepted, but ideally full frameworks will be separate packages and don't belong here
- Ideally, contributions will have unit tests
- Ideally, contributions will not have external python dependencies other than WPILib – though, this may change.
- Contributions will work (or at least, not break) on all supported RobotPy platforms (Windows/Linux/OSX, RoboRio)
- All pull requests will be tested using Travis-CI

CHAPTER

ONE

UTILITIES API

1.1 commandbased module

class commandbased.commandbasedrobot.**CommandBasedRobot**(*self: wpilib._wpilib.TimedRobot*, *period: seconds = 0.02*) → None

Bases: wpilib._wpilib.TimedRobot

The base class for a Command-Based Robot. To use, instantiate commands and trigger them.

Constructor for TimedRobot.

Parameters **period** – Period.

commandPeriodic()
Run the scheduler regularly. If an error occurs during a competition, prevent it from crashing the program.

handleCrash(*error*)
Called if an exception is raised in the Scheduler during a competition. Writes an error message to the driver station by default. If you want more complex behavior, override this method in your robot class.

class commandbased.cancelcommand.**CancelCommand**(*command*)
When this command is run, it cancels the command it was passed.

Parameters **command** (Command) – The command to cancel.

initialize(*self: wpilib.command._commands_v1.Command*) → None
The initialize method is called the first time this Command is run after being started.

isFinished(*self: wpilib.command._commands_v1.Command*) → bool
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.

Returning false will result in the command never ending automatically. It may still be cancelled manually or interrupted by another command. Returning true will result in the command executing once and finishing immediately. We recommend using InstantCommand for this.

Returns Whether this command is finished. @see IsTimedOut()

These functions can be used to make programming CommandGroups much more intuitive. For more information, check each method's docstring.

commandbased.flowcontrol.**BREAK**(*steps=1*)
Calling this function will end the loop that contains it. Pass an integer to break out of that number of nested loops.

class commandbased.flowcontrol.**CommandFlow**(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: wpilib.command._commands_v1.CommandGroup) -> None
2. __init__(self: wpilib.command._commands_v1.CommandGroup, name: str) -> None

Creates a new CommandGroup with the given name.

Parameters **name** – The name for this command group

addParallel(**args*, ***kwargs*)

Overloaded function.

1. addParallel(self: wpilib.command._commands_v1.CommandGroup, command: wpilib.command._commands_v1.Command) -> None

Adds a new child Command to the group. The Command will be started after all the previously added Commands.

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, 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

2. addParallel(self: wpilib.command._commands_v1.CommandGroup, command: wpilib.command._commands_v1.Command, timeout: float) -> None

Adds a new child Command to the group with the given 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, 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)

addSequential(**args*, ***kwargs*)

Overloaded function.

1. addSequential(self: wpilib.command._commands_v1.CommandGroup, command: wpilib.command._commands_v1.Command) -> None

Adds a new Command to the group. The Command will be started after all the previously added Commands.

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

2. addSequential(self: wpilib.command._commands_v1.CommandGroup, command: wpilib.command._commands_v1.Command, timeout: float) -> None

Adds a new Command to the group with a given 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. 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)

setParent(*self: wpilib.command._commands_v1.Command*, *parent: wpilib.command._commands_v1.CommandGroup*) → None
Sets the parent of this command. No actual change is made to the group.

Parameters **parent** – the parent

start(*self: wpilib.command._commands_v1.Command*) → None
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.

commandbased.flowcontrol.**ELIF**(*condition*)
Use as a decorator for a function. That function will be placed into a CommandGroup which will be triggered by a ConditionalCommand that uses the passed condition. That ConditionalCommand will then be added as the onFalse for the ConditionalCommand created by a previous IF or ELIF.

commandbased.flowcontrol.**ELSE**(*func*)
Use as a decorator for a function. That function will be placed into a CommandGroup which will be added as the onFalse for the ConditionalCommand created by a previous IF or ELIF.

commandbased.flowcontrol.**IF**(*condition*)
Use as a decorator for a function. That function will be placed into a CommandGroup and run inside a ConditionalCommand with the given condition. The decorated function must accept one positional argument that will be used as its 'self'.

commandbased.flowcontrol.**RETURN**()
Calling this function will end the source CommandGroup immediately.

commandbased.flowcontrol.**WHILE**(*condition*)
Use as a decorator for a function. That function will be placed into a CommandGroup, which will be added to a ConditionalCommand. It will be modified to restart itself automatically.

1.2 magicbot module

class magicbot.magicrobot.**MagicRobot**
Bases: wpilib._wpilib.RobotBase

Robots that use the MagicBot framework should use this as their base robot class. If you use this as your base, you must implement the following methods:

- *createObjects()*
- *teleopPeriodic()*

MagicRobot uses the *AutonomousModeSelector* to allow you to define multiple autonomous modes and to select one of them via the SmartDashboard/Shuffleboard.

MagicRobot will set the following NetworkTables variables automatically:

- /robot/mode: one of 'disabled', 'auto', 'teleop', or 'test'
- /robot/is_simulation: True/False
- /robot/is_ds_attached: True/False

Constructor for a generic robot program.

User code should be placed in the constructor that runs before the Autonomous or Operator Control period starts. The constructor will run to completion before Autonomous is entered.

This must be used to ensure that the communications code starts. In the future it would be nice to put this code into it's own task that loads on boot so ensure that it runs.

autonomousInit()
Initialization code for autonomous mode may go here.

Users may override this method for initialization code which will be called each time the robot enters autonomous mode, regardless of the selected autonomous mode.

This can be useful for code that must be run at the beginning of a match.

Note: This method is called after every component's on_enable method, but before the selected autonomous mode's on_enable method.

Return type None

consumeExceptions(*forceReport=False*)
This returns a context manager which will consume any uncaught exceptions that might otherwise crash the robot.

Example usage:

```
def teleopPeriodic(self):
    with self.consumeExceptions():
        if self.joystick.getTrigger():
            self.shooter.shoot()

    with self.consumeExceptions():
        if self.joystick.getRawButton(2):
            self.ball_intake.run()

    # and so on...
```

Parameters **forceReport** (bool) – Always report the exception to the DS. Don't set this to True

See also:

onException() for more details

control_loop_wait_time = 0.02
Amount of time each loop takes (default is 20ms)

createObjects()
You should override this and initialize all of your wpilib objects here (and not in your components, for example). This serves two purposes:

- It puts all of your motor/sensor initialization in the same place, so that if you need to change a port/pin number it makes it really easy to find it. Additionally, if you want to create a simplified robot program to test a specific thing, it makes it really easy to copy/paste it elsewhere
- It allows you to use the magic injection mechanism to share variables between components

---

Note: Do not access your magic components in this function, as their instances have not been created yet. Do not create them either.

Return type None

disabledInit()
Initialization code for disabled mode may go here.

Users may override this method for initialization code which will be called each time the robot enters disabled mode.

Note: The on_disable functions of all components are called before this function is called.

Return type None

disabledPeriodic()
Periodic code for disabled mode should go here.

Users should override this method for code which will be called periodically at a regular rate while the robot is in disabled mode.

This code executes before the execute functions of all components are called.

endCompetition()

Return type None

error_report_interval = 0.5
Error report interval: when an FMS is attached, how often should uncaught exceptions be reported?

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

onException(*forceReport=False*)
This function must *only* be called when an unexpected exception has occurred that would otherwise crash the robot code. Use this inside your operatorActions() function.

If the FMS is attached (eg, during a real competition match), this function will return without raising an error. However, it will try to report one-off errors to the Driver Station so that it will be recorded in the Driver Station Log Viewer. Repeated errors may not get logged.

Example usage:

```
def teleopPeriodic(self):
    try:
        if self.joystick.getTrigger():
            self.shooter.shoot()
    except:
        self.onException()

    try:
        if self.joystick.getRawButton(2):
            self.ball_intake.run()
    except:
        self.onException()

    # and so on...
```

Parameters **forceReport** (bool) – Always report the exception to the DS. Don't set this to True

Return type None

robotPeriodic()

Periodic code for all modes should go here.

Users must override this method to utilize it but it is not required.

This function gets called last in each mode. You may use it for any code you need to run during all modes of the robot (e.g NetworkTables updates)

The default implementation will update SmartDashboard, LiveWindow and Shuffleboard.

Return type None

startCompetition()

This runs the mode-switching loop.

Warning: Internal API, don't override

Return type None

teleopInit()

Initialization code for teleop control code may go here.

Users may override this method for initialization code which will be called each time the robot enters teleop mode.

Note: The on_enable functions of all components are called before this function is called.

Return type None

teleopPeriodic()
Periodic code for teleop mode should go here.

Users should override this method for code which will be called periodically at a regular rate while the robot is in teleop mode.

This code executes before the `execute` functions of all components are called.

Note: If you want this function to be called in autonomous mode, set `use_teleop_in_autonomous` to True in your robot class.

---

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 disabled mode.

Return type `None`

testPeriodic()
Periodic code for test mode should go here.

Return type `None`

use_teleop_in_autonomous = False
If True, teleopPeriodic will be called in autonomous mode

1.2.1 Component

class magicbot.magiccomponent.**MagicComponent**
Bases: `object`

To automagically retrieve variables defined in your base robot object, you can add the following:

```
class MyComponent:

    # other variables 'imported' automatically from MagicRobot
    elevator_motor: Talon
    other_component: MyOtherComponent

    ...

    def execute(self):

        # This will be automatically set to the Talon
        # instance created in robot.py
        self.elevator_motor.set(self.value)
```

What this says is "find the variable in the robot class called 'elevator_motor', which is a Talon". If the name and type match, then the variable will automatically be injected into your component when it is created.

Note: You don't need to inherit from `MagicComponent`, it is only provided for documentation's sake

---

execute()
This function is called at the end of the control loop

logger: logging.Logger

on_disable()
Called when the robot leaves autonomous or teleoperated

on_enable()
Called when the robot enters autonomous or teleoperated mode. This function should initialize your component to a "safe" state so that unexpected things don't happen when enabling the robot.

Note: You'll note that there isn't a separate initialization function for autonomous and teleoperated modes. This is intentional, as they should be the same.

---

setup()
This function is called after `createObjects` has been called in the main robot class, and after all components have been created

The setup function is optional and components do not have to define one. `setup()` functions are called in order of component definition in the main robot class.

Note: For technical reasons, variables imported from MagicRobot are not initialized when your component's constructor is called. However, they will be initialized by the time this function is called.

1.2.2 Tunable

magicbot.magic_tunable.**collect_feedbacks**(*component*, *cname*, *prefix='components'*)
Finds all methods decorated with *feedback()* on an object and returns a list of 2-tuples (method, NetworkTables entry).

Note: This isn't useful for normal use.

---

magicbot.magic_tunable.**feedback**(*f=None*, ***, *key=None*)
This decorator allows you to create NetworkTables values that are automatically updated with the return value of a method.

`key` is an optional parameter, and if it is not supplied, the key will default to the method name with a leading `get_` removed. If the method does not start with `get_`, the key will be the full name of the method.

The key of the NetworkTables value will vary based on what kind of object the decorated method belongs to:

- A component: `/components/COMPONENTNAME/VARNAME`
- Your main robot class: `/robot/VARNAME`

The NetworkTables value will be auto-updated in all modes.

Warning: The function should only act as a getter, and must not take any arguments (other than self).

Example:

```
from magicbot import feedback

class MyComponent:
    navx: ...
```

(continues on next page)

(continued from previous page)

```
    @feedback
    def get_angle(self):
        return self.navx.getYaw()

class MyRobot(magicbot.MagicRobot):
    my_component: MyComponent

    ...
```

In this example, the NetworkTable key is stored at `/components/my_component/angle`.

See also:

`LiveWindow` may suit your needs, especially if you wish to monitor WPILib objects.

New in version 2018.1.0.

magicbot.magic_tunable.**setup_tunables**(*component*, *cname*, *prefix='components'*)

Connects the tunables on an object to NetworkTables.

Parameters

- **component** – Component object
- **cname** (str) – Name of component
- **prefix** (Optional[str]) – Prefix to use, or no prefix if None

Note: This is not needed in normal use, only useful for testing

Return type None

class magicbot.magic_tunable.**tunable**(*default*, ***, *writeDefault=True*, *subtable=None*, *doc=None*)

Bases: Generic[magicbot.magic_tunable.V]

This allows you to define simple properties that allow you to easily communicate with other programs via NetworkTables.

The following example will define a NetworkTable variable at `/components/my_component/foo`:

```
class MyRobot(magicbot.MagicRobot):

    my_component: MyComponent

...

from magicbot import tunable

class MyComponent:

    # define the tunable property
    foo = tunable(True)

    def execute(self):

        # set the variable
```

(continues on next page)

(continued from previous page)

```
        self.foo = True

        # get the variable
        foo = self.foo
```

The key of the NetworkTables variable will vary based on what kind of object the decorated method belongs to:

- A component: `/components/COMPONENTNAME/VARNAME`
- An autonomous mode: `/autonomous/MODENAME/VARNAME`
- Your main robot class: `/robot/VARNAME`

Note: When executing unit tests on objects that create tunables, you will want to use setup_tunables to set the object up. In normal usage, MagicRobot does this for you, so you don't have to do anything special.

1.2.3 Resettable

`magicbot.magic_reset.`**`collect_resets`**(*cls*)

Get all the `will_reset_to` variables and their values from a class.

Note: This isn't useful for normal use.

Return type `Dict[str, Any]`

`class` `magicbot.magic_reset.`**`will_reset_to`**(*default*)

Bases: `object`

This marker indicates that this variable on a component will be reset to a default value after each time that execute is called.

Example usage:

```
class Component:

    foo = will_reset_to(False)

    def control_fn(self):
        self.foo = True

    def execute(self):
        if self.foo:
            # ...

        # after this function is executed, foo is reset
        # back to the default value (False)
```

Note: This will only work for MagicRobot components

> **Warning:** This will not work on classes that set `__slots__`.

default

1.2.4 State machines

class magicbot.state_machine.**AutonomousStateMachine**
Bases: *magicbot.state_machine.StateMachine*

This is a specialized version of the StateMachine that is designed to be used as an autonomous mode. There are a few key differences:

- The `engage()` function is always called, so the state machine will always run to completion unless done() is called
- VERBOSE_LOGGING is set to True, so a log message will be printed out upon each state transition

VERBOSE_LOGGING = True

done()
Call this function to end execution of the state machine.

This function will always be called when a state machine ends. Even if the engage function is called repeatedly, done() will be called.

> **Note:** If you wish to do something each time execution ceases, override this function (but be sure to call `super().done()`!)

on_enable()
magicbot component API: called when autonomous/teleop is enabled

on_iteration(*tm*)

exception magicbot.state_machine.**IllegalCallError**
Bases: Exception

exception magicbot.state_machine.**InvalidStateName**
Bases: Exception

exception magicbot.state_machine.**InvalidWrapperError**
Bases: Exception

exception magicbot.state_machine.**MultipleDefaultStatesError**
Bases: Exception

exception magicbot.state_machine.**MultipleFirstStatesError**
Bases: Exception

exception magicbot.state_machine.**NoFirstStateError**
Bases: Exception

class magicbot.state_machine.**StateMachine**
Bases: object

The StateMachine class is used to implement magicbot components that allow one to easily define a finite state machine (FSM) that can be executed via the magicbot framework.

You create a component class that inherits from `StateMachine`. Each state is represented as a single function, and you indicate that a function is a particular state by decorating it with one of the following decorators:

- *@default_state*
- *@state*
- *@timed_state*

As the state machine executes, the decorated function representing the current state will be called. Decorated state functions can receive the following parameters (all of which are optional):

- `tm` - The number of seconds since autonomous has started
- `state_tm` - The number of seconds since this state has been active (note: it may not start at zero!)
- `initial_call` - Set to True when the state is initially called, False otherwise. If the state is switched to multiple times, this will be set to True at the start of each state.

To be consistent with the magicbot philosophy, in order for the state machine to execute its states you must call the *engage()* function upon each execution of the main robot control loop. If you do not call this function, then execution of the FSM will cease.

Note: If you wish for the FSM to continue executing state functions regardless whether `engage()` is called, you must set the `must_finish` parameter in your state decorator to be True.

When execution ceases (because `engage()` was not called), the *done()* function will be called and the FSM will be reset to the starting state. The state functions will not be called again unless `engage` is called.

As a magicbot component, StateMachine contains an `execute` function that will be called on each control loop. All state execution occurs from within that function call. If you call other components from a StateMachine component, you should ensure that your component is defined *before* the other components in your Robot class.

Warning: As StateMachine already contains an execute function, there is no need to define your own `execute` function for a state machine component – if you override `execute`, then the state machine may not work correctly. Instead, use the *@default_state* decorator.

Here's a very simple example of how you might implement a shooter automation component that moves a ball into a shooter when the shooter is ready:

```
class ShooterAutomation:

    # Some other component
    shooter = Shooter
    ball_pusher = BallPusher

    def fire(self):
        """This is called from the main loop"""
        self.engage()

    @state(first=True)
    def begin_firing(self):
        """This function will only be called IFF fire is called and
           the FSM isn't currently in the 'firing' state. If fire
           was not called, this function will not execute."""
        self.shooter.enable()
        if self.shooter.ready():
            self.next_state('firing')

    @timed_state(duration=1.0, must_finish=True)
```

(continues on next page)

(continued from previous page)

```
    def firing(self):
        """Because must_finish=True, once the FSM has reached this
           state, this state will continue executing even if engage
           isn't called"""
        self.shooter.enable()
        self.ball_pusher.push()

    #
    # Note that there is no execute function defined as part of
    # this component
    #
...

class MyRobot(magicbot.MagicRobot):

    ...

    def teleopPeriodic(self):

        if self.joystick.getTrigger():
            self.shooter_automation.fire()
```

This object has a lot of really useful NetworkTables integration as well:

- tunables are created in /components/NAME/state - state durations can be tuned here - The 'current state' is output as it happens - Descriptions and names of the states are here (for dashboard use)

Warning: This object is not intended to be threadsafe and should not be accessed from multiple threads

VERBOSE_LOGGING = False

current_state
NT variable that indicates which state will be executed next (though, does not guarantee that it will be executed). Will return an empty string if the state machine is not currently engaged.

done()
Call this function to end execution of the state machine.

This function will always be called when a state machine ends. Even if the engage function is called repeatedly, done() will be called.

Note: If you wish to do something each time execution ceases, override this function (but be sure to call `super().done()`!)

---

engage(*initial_state=None*, *force=False*)
This signals that you want the state machine to execute its states.

Parameters

- **initial_state** – If specified and execution is not currently occurring, start in this state instead of in the 'first' state
- **force** – If True, will transition even if the state machine is currently active.

execute()
magicbot component API: This is called on each iteration of the control loop. Most of the time, you

will not want to override this function. If you find you want to, you may want to use the @default_state mechanism instead.

property is_executing

Returns True if the state machine is executing states

next_state(*name*)
Call this function to transition to the next state

Parameters **name** – Name of the state to transition to

Note: This should only be called from one of the state functions

---

next_state_now(*name*)
Call this function to transition to the next state, and call the next state function immediately. Prefer to use *next_state()* instead.

Parameters **name** – Name of the state to transition to

Note: This should only be called from one of the state functions

---

on_disable()
magicbot component API: called when autonomous/teleop is disabled

on_enable()
magicbot component API: called when autonomous/teleop is enabled

magicbot.state_machine.**default_state**(*f=None*)
If this decorator is applied to a method in an object that inherits from *StateMachine*, it indicates that the method is a default state; that is, if no other states are executing, this state will execute. If the state machine is always executing, the default state will never execute.

There can only be a single default state in a StateMachine object.

The decorated function can have the following arguments in any order:

- `tm` - The number of seconds since the state machine has started
- `state_tm` - The number of seconds since this state has been active (note: it may not start at zero!)
- `initial_call` - Set to True when the state is initially called, False otherwise. If the state is switched to multiple times, this will be set to True at the start of each state execution.

magicbot.state_machine.**state**(*f=None*, ***, *first=False*, *must_finish=False*)
If this decorator is applied to a function in an object that inherits from *StateMachine*, it indicates that the function is a state. The state will continue to be executed until the `next_state` function is executed.

The decorated function can have the following arguments in any order:

- `tm` - The number of seconds since the state machine has started
- `state_tm` - The number of seconds since this state has been active (note: it may not start at zero!)
- `initial_call` - Set to True when the state is initially called, False otherwise. If the state is switched to multiple times, this will be set to True at the start of each state execution.

Parameters

- **first** (*bool*) – If True, this state will be ran first

- **must_finish** (*bool*) – If True, then this state will continue executing even if `engage()` is not called. However, if `done()` is called, execution will stop regardless of whether this is set.

`magicbot.state_machine.`**`timed_state`**(*f=None*, ***, *duration=None*, *next_state=None*, *first=False*, *must_finish=False*)

If this decorator is applied to a function in an object that inherits from *StateMachine*, it indicates that the function is a state that will run for a set amount of time unless interrupted.

It is guaranteed that a timed_state will execute at least once, even if it expires prior to being executed.

The decorated function can have the following arguments in any order:

- `tm` - The number of seconds since the state machine has started
- `state_tm` - The number of seconds since this state has been active (note: it may not start at zero!)
- `initial_call` - Set to True when the state is initially called, False otherwise. If the state is switched to multiple times, this will be set to True at the start of each state execution.

Parameters

- **duration** (*float*) – The length of time to run the state before progressing to the next state
- **next_state** (*str*) – The name of the next state. If not specified, then this will be the last state executed if time expires
- **first** (*bool*) – If True, this state will be ran first
- **must_finish** (*bool*) – If True, then this state will continue executing even if `engage()` is not called. However, if `done()` is called, execution will stop regardless of whether this is set.

1.3 robotpy_ext.autonomous package

1.3.1 robotpy_ext.autonomous.selector module

class `robotpy_ext.autonomous.selector.`**`AutonomousModeSelector`**(*autonomous_pkgname*, **args*, ***kwargs*)

Bases: `object`

This object loads all modules in a specified python package, and tries to automatically discover autonomous modes from them. Each module is added to a `SendableChooser` object, which allows the user to select one of them via SmartDashboard.

Autonomous mode objects must implement the following functions:

- `on_enable` - Called when autonomous mode is initially enabled
- `on_disable` - Called when autonomous mode is no longer active
- `on_iteration` - Called for each iteration of the autonomous control loop

Your autonomous object may have the following attributes:

- `MODE_NAME` - The name of the autonomous mode to display to users
- `DISABLED` - If True, don't allow this mode to be selected
- `DEFAULT` - If True, this is the default autonomous mode selected

Here is an example of using `AutonomousModeSelector` in `TimedRobot`:

```
class MyRobot(wpilib.TimedRobot):

    def robotInit(self):
        self.automodes = AutonomousModeSelector('autonomous')

    def autonomousInit(self):
        self.automodes.start()

    def autonomousPeriodic(self):
        self.automodes.periodic()

    def disabledInit(self):
        self.automodes.disable()
```

If you use AutonomousModeSelector, you may also be interested in the autonomous state machine helper (*StatefulAutonomous*).

Check out the samples in our github repository that show some basic usage of `AutonomousModeSelector`.

Note: If you use AutonomousModeSelector, then you should add `robotpy_ext.autonomous.selector_tests` to your pyfrc unit tests like so:

```
from robotpy_ext.autonomous.selector_tests import *
```

Note: For your autonomous mode's `on_disable` method to be called, you must call *disable()* in `disabledInit`.

It is okay to not call *disable()* if you do not need `on_disable`.

Parameters

- **autonomous_pkgname** – Module to load autonomous modes from
- **args** – Args to pass to created autonomous modes
- **kwargs** – Keyword args to pass to created autonomous modes

disable()

Disables the active autonomous mode.

You can call this from your `disabledInit` method to call your autonomous mode's `on_disable` method.

New in version 2020.1.5.

Return type `None`

periodic()

Execute one control loop iteration of the active autonomous mode.

Call this from your `autonomousPeriodic` method.

New in version 2020.1.5.

Return type `None`

run(*control_loop_wait_time=0.02*, *iter_fn=None*, *on_exception=None*, *watchdog=None*)
This method implements the entire autonomous loop.

Do not call this from `TimedRobot` as this will break the timing of your control loop when your robot switches to teleop.

This function will NOT exit until autonomous mode has ended. If you need to execute code in all autonomous modes, pass a function or list of functions as the `iter_fn` parameter, and they will be called once per autonomous mode iteration.

Parameters

- **control_loop_wait_time** – Amount of time between iterations
- **iter_fn** – Called at the end of every iteration while autonomous mode is executing
- **on_exception** – Called when an uncaught exception is raised, must take a single keyword arg "forceReport"
- **watchdog** (`Union[Watchdog, SimpleWatchdog, None]`) – a WPILib Watchdog to feed every iteration

start()
Start autonomous mode.

This initialises the selected autonomous mode. Call this from your `autonomousInit` method.

New in version 2020.1.5.

Return type `None`

1.3.2 robotpy_ext.autonomous.selector_tests module

1.3.3 robotpy_ext.autonomous.stateful_autonomous module

exception robotpy_ext.autonomous.stateful_autonomous.**InvalidWrapperError**
Bases: `Exception`

class robotpy_ext.autonomous.stateful_autonomous.**StatefulAutonomous**(*components=None*)
Bases: `object`

This object is designed to be used to implement autonomous modes that can be used with the *AutonomousModeSelector* object to select an appropriate autonomous mode. However, you don't have to.

This object is designed to meet the following goals:

- Supports simple built-in tuning of autonomous mode parameters via SmartDashboard
- Easy to create autonomous modes that support state machine or time-based operation
- Autonomous modes that are easy to read and understand

You use this by defining a class that inherits from `StatefulAutonomous`. To define each state, you use the *timed_state()* decorator on a function. When each state is run, the decorated function will be called. Decorated functions can receive the following parameters:

- `tm` - The number of seconds since autonomous has started
- `state_tm` - The number of seconds since this state has been active (note: it may not start at zero!)
- `initial_call` - Set to True when the state is initially called, False otherwise. If the state is switched to multiple times, this will be set to True at the start of each state.

An example autonomous mode that drives the robot forward for 5 seconds might look something like this:

```
from robotpy_ext.autonomous import StatefulAutonomous

class DriveForward(StatefulAutonomous):

    MODE_NAME = 'Drive Forward'

    def initialize(self):
        pass

    @timed_state(duration=0.5, next_state='drive_forward', first=True)
    def drive_wait(self):
        pass

    @timed_state(duration=5)
    def drive_forward(self):
        self.drive.move(0, 1, 0)
```

Note that in this example, it is assumed that the DriveForward object is initialized with a dictionary with a value 'drive' that contains an object that has a move function:

```
components = {'drive': SomeObject() }
mode = DriveForward(components)
```

If you use this object with *AutonomousModeSelector*, make sure to initialize it with the dictionary, and it will be passed to this autonomous mode object when initialized.

See also:

Check out the samples in our github repository that show some basic usage of `AutonomousModeSelector`.

Parameters **components** (*dict*) – A dictionary of values that will be assigned as attributes to this object, using the key names in the dictionary

done()
Call this function to indicate that no more states should be called

next_state(*name*)
Call this function to transition to the next state

Parameters **name** – Name of the state to transition to

on_disable()
Called when the autonomous mode is disabled

on_enable()
Called when autonomous mode is enabled, and initializes the state machine internals.

If you override this function, be sure to call it from your customized `on_enable` function:

```
super().on_enable()
```

on_iteration(*tm*)
This function is called by the autonomous mode switcher, should not be called by enduser code. It is called once per control loop iteration.

register_sd_var(*name*, *default*, *add_prefix=True*, *vmin=- 1*, *vmax=1*)
Register a variable that is tunable via NetworkTables/SmartDashboard

When this autonomous mode is enabled, all of the SmartDashboard settings will be read and stored as attributes of this object. For example, to register a variable 'foo' with a default value of 1:

```
self.register_sd_var('foo', 1)
```

This value will show up on NetworkTables as the key `MODE_NAME\foo` if add_prefix is specified, otherwise as `foo`.

Parameters

- **name** – Name of variable to display to user, cannot have a space in it.
- **default** – Default value of variable
- **add_prefix** (*bool*) – Prefix this setting with the mode name
- **vmin** – For tuning: minimum value of this variable
- **vmax** – For tuning: maximum value of this variable

robotpy_ext.autonomous.stateful_autonomous.**state**(*f=None*, *first=False*)

If this decorator is applied to a function in an object that inherits from *StatefulAutonomous*, it indicates that the function is a state. The state will continue to be executed until the `next_state` function is executed.

The decorated function can have the following arguments in any order:

- `tm` - The number of seconds since autonomous has started
- `state_tm` - The number of seconds since this state has been active (note: it may not start at zero!)
- `initial_call` - Set to True when the state is initially called, False otherwise. If the state is switched to multiple times, this will be set to True at the start of each state.

Parameters **first** (*bool*) – If True, this state will be ran first

robotpy_ext.autonomous.stateful_autonomous.**timed_state**(*f=None*, *duration=None*, *next_state=None*, *first=False*)

If this decorator is applied to a function in an object that inherits from *StatefulAutonomous*, it indicates that the function is a state that will run for a set amount of time unless interrupted

The decorated function can have the following arguments in any order:

- `tm` - The number of seconds since autonomous has started
- `state_tm` - The number of seconds since this state has been active (note: it may not start at zero!)
- `initial_call` - Set to True when the state is initially called, False otherwise. If the state is switched to multiple times, this will be set to True at the start of each state.

Parameters

- **duration** (*float*) – The length of time to run the state before progressing to the next state
- **next_state** (*str*) – The name of the next state. If not specified, then this will be the last state executed if time expires
- **first** (*bool*) – If True, this state will be ran first

1.4 robotpy_ext.common_drivers package

1.4.1 robotpy_ext.common_drivers.distance_sensors module

class robotpy_ext.common_drivers.distance_sensors.**SharpIR2Y0A02**(*port*)

Sharp GP2Y0A02YK0F is an analog IR sensor capable of measuring distances from 20cm to 150cm. Output distance is measured in centimeters.

Distance is calculated using the following equation derived from the graph provided in the datasheet:

```
62.28*x ^ -1.092
```

Warning: FRC Teams: the case on these sensors is conductive and grounded, and should not be mounted on a metallic surface!

Parameters **port** – Analog port number

getDistance()

Returns distance in centimeters. The output is constrained to be between 22.5 and 145

class robotpy_ext.common_drivers.distance_sensors.**SharpIR2Y0A21**(*port*)

Sharp GP2Y0A21YK0F is an analog IR sensor capable of measuring distances from 10cm to 80cm. Output distance is measured in centimeters.

Distance is calculated using the following equation derived from the graph provided in the datasheet:

```
26.449*x ^ -1.226
```

Warning: FRC Teams: the case on these sensors is conductive and grounded, and should not be mounted on a metallic surface!

Parameters **port** – Analog port number

getDistance()

Returns distance in centimeters. The output is constrained to be between 10 and 80

class robotpy_ext.common_drivers.distance_sensors.**SharpIRGP2Y0A41SK0F**(*port*)

Sharp GP2Y0A41SK0F is an analog IR sensor capable of measuring distances from 4cm to 40cm. Output distance is measured in centimeters.

Distance is calculated using the following equation derived from the graph provided in the datasheet:

```
12.84*x ^ -0.9824
```

Warning: FRC Teams: the case on these sensors is conductive and grounded, and should not be mounted on a metallic surface!

Parameters **port** – Analog port number

getDistance()

Returns distance in centimeters. The output is constrained to be between 4.5 and 35

1.4.2 robotpy_ext.common_drivers.driver_base module

class robotpy_ext.common_drivers.driver_base.**DriverBase**
This should be the base class for all drivers in the cdl, currently all it does is spit out a warning message if the driver has not been verified.

Constructor for DriverBase, all this does is print a message to console if the driver has not been verified yet.

verified = False

1.4.3 robotpy_ext.common_drivers.units module

class robotpy_ext.common_drivers.units.**Unit**(*base_unit*, *base_to_unit*, *unit_to_base*)
Bases: object

The class for all of the units here

Unit constructor, used as a mechanism to convert between various measurements :param base_unit: The instance of Unit to base conversions from. If None, then assume it is the ultimate base unit :param base_to_unit: A callable to convert measurements between this unit and the base unit :param unit_to_base: A callable to convert measurements between the base unit and this unit

robotpy_ext.common_drivers.units.**convert**(*source_unit*, *target_unit*, *value*)
Convert between units, returns value in target_unit :param source_unit: The unit of value :param target_unit: The desired output unit :param value: The value, in source_unit, to convert

1.4.4 robotpy_ext.common_drivers.xl_max_sonar_ez module

These are a set of drivers for the XL-MaxSonar EZ series of sonar modules. The devices have a few different ways of reading from them, and the these drivers attempt to cover some of the methods

class robotpy_ext.common_drivers.xl_max_sonar_ez.**MaxSonarEZAnalog**(*channel*, *output_units=<robotpy_ext.common_d*
object>)
Bases: *robotpy_ext.common_drivers.driver_base.DriverBase*

This is a driver for the MaxSonar EZ series of sonar sensors, using the analog output of the sensor.

To use this driver, pin 3 on the sensor must be mapped to an analog pin, and the sensor must be on a 5v supply.

Sonar sensor constructor

Parameters

- **channel** – The analog input index which is wired to the analog output pin (pin 3) on the sensor.
- **output_units** – The Unit instance specifying the format of value to return

get()
Return the current sonar sensor reading, in the units specified from the constructor

verified = False

class robotpy_ext.common_drivers.xl_max_sonar_ez.**MaxSonarEZPulseWidth**(*channel*, *output_units=<robotpy_ext.common_drivers.xl_max_sonar_ez.Unit object>*)

Bases: *robotpy_ext.common_drivers.driver_base.DriverBase*

This is a driver for the MaxSonar EZ series of sonar sensors, using the pulse-width output of the sensor.

To use this driver, pin 2 on the sensor must be mapped to a dio pin.

Sonar sensor constructor

Parameters

- **channel** – The digital input index which is wired to the pulse-width output pin (pin 2) on the sensor.
- **output_units** – The Unit instance specifying the format of value to return

get()
Return the current sonar sensor reading, in the units specified from the constructor

verified = True

1.5 robotpy_ext.control package

1.5.1 robotpy_ext.control.button_debouncer module

class robotpy_ext.control.button_debouncer.**ButtonDebouncer**(*joystick*, *buttonnum*, *period=0.5*)

Useful utility class for debouncing buttons

Parameters

- **joystick** (wpilib.Joystick) – Joystick object
- **buttonnum** (*int*) – Number of button to retrieve
- **period** (*float*) – Period of time (in seconds) to wait before allowing new button presses. Defaults to 0.5 seconds.

get()
Returns the value of the joystick button. If the button is held down, then True will only be returned once every debounce_period seconds

set_debounce_period(*period*)
Set number of seconds to wait before returning True for the button again

1.5.2 robotpy_ext.control.toggle module

class robotpy_ext.control.toggle.**Toggle**(*joystick*, *button*, *debounce_period=None*)
Utility class for joystick button toggle

Usage:

```
foo = Toggle(joystick, 3)

if foo:
    toggleFunction()

if foo.on:
    onToggle()

if foo.off:
    offToggle()
```

Parameters

- **joystick** (Joystick) – wpilib.Joystick that contains the button to toggle
- **button** (int) – Number of button that will act as toggle. Same value used in *getRawButton()*
- **debounce_period** (Optional[float]) – Period in seconds to wait before registering a new button press.

get()

Returns State of toggle

Return type bool

property off
Equates to true if toggle is in the 'off' state

property on
Equates to true if toggle is in the 'on' state

1.6 robotpy_ext.misc package

1.6.1 robotpy_ext.misc.asyncio_policy module

This is a replacement event loop and policy for asyncio that uses FPGA time, rather than native python time.

class robotpy_ext.misc.asyncio_policy.**FPGATimedEventLoop**(*selector=None*)
Bases: asyncio.unix_events._UnixSelectorEventLoop

An asyncio event loop that uses wpilib time rather than python time

time()
Return the time according to the event loop's clock.

This is a float expressed in seconds since an epoch, but the epoch, precision, accuracy and drift are unspecified and may differ per event loop.

class robotpy_ext.misc.asyncio_policy.**FPGATimedEventLoopPolicy**
Bases: asyncio.events.AbstractEventLoopPolicy

An asyncio event loop policy that uses FPGATimedEventLoop

robotpy_ext.misc.asyncio_policy.**patch_asyncio_policy**()
Sets an instance of FPGATimedEventLoopPolicy as the default asyncio event loop policy

1.6.2 robotpy_ext.misc.looptimer module

class robotpy_ext.misc.looptimer.**LoopTimer**(*logger*)
Bases: object

A utility class that measures the number of loops that a robot program executes, and computes the min/max/average period for loops in the last second.

Example usage:

```
class Robot(wpilib.IterativeRobot):

    def teleopInit(self):
        self.loop_timer = LoopTimer(self.logger)

    def teleopPeriodic(self):
        self.loop_timer.measure()
```

Mainly intended for debugging purposes to measure how much lag.

measure()
Computes loop performance information and periodically dumps it to the info logger.

reset()

1.6.3 robotpy_ext.misc.precise_delay module

class robotpy_ext.misc.precise_delay.**NotifierDelay**(*delay_period*)
Bases: object

Synchronizes a timing loop against interrupts from the FPGA.

This will delay so that the next invocation of your loop happens at precisely the same period, assuming that your loop does not take longer than the specified period.

Example:

```
with NotifierDelay(0.02) as delay:
    while something:
        # do things here
        delay.wait()
```

Parameters **delay_period** (float) – The period's amount of time (in seconds).

free()
Clean up the notifier.

Do not use this object after this method is called.

Return type None

wait()
Wait until the delay period has passed.

Return type None

class robotpy_ext.misc.precise_delay.**PreciseDelay**(*delay_period*)
Bases: object

Used to synchronize a timing loop. Will delay precisely so that the next invocation of your loop happens at the same period, as long as your code does not run longer than the length of the delay.

Our experience has shown that 25ms is a good loop period.

Usage:

```
delay = PreciseDelay(time_to_delay)

while something:
    # do things here
    delay.wait()
```

Deprecated since version 2019: PreciseDelay is terribly inefficient. Use *NotifierDelay* instead.

Parameters **delay_period** (*float*) – The amount of time (in seconds) to do a delay

wait()
Waits until the delay period has passed

1.6.4 robotpy_ext.misc.periodic_filter module

class robotpy_ext.misc.periodic_filter.**PeriodicFilter**(*period*, *bypass_level=30*)
Bases: object

Periodic Filter to help keep down clutter in the console. Simply add this filter to your logger and the logger will only print periodically.

The logger will always print logging levels of WARNING or higher, unless given a different bypass level

Example:

```
class Component1:

    def setup(self):
        # Set period to 3 seconds, set bypass_level to WARN
        self.logger.addFilter(PeriodicFilter(3, bypass_level=logging.WARN))

    def execute(self):
        # This message will be printed once every three seconds
        self.logger.info('Component1 Executing')

        # This message will be printed out every loop
        self.logger.warn("Uh oh, this shouldn't have happened...")
```

Parameters

- **period** – Wait period (in seconds) between logs
- **bypass_level** – Lowest logging level that the filter should not catch

filter(*record*)
Performs filtering action for logger

1.6.5 robotpy_ext.misc.simple_watchdog module

class robotpy_ext.misc.simple_watchdog.**SimpleWatchdog**(*timeout*)
Bases: object

A class that's a wrapper around a watchdog timer.

When the timer expires, a message is printed to the console and an optional user-provided callback is invoked.

The watchdog is initialized disabled, so the user needs to call enable() before use.

Note: This is a simpler replacement for the wpilib.Watchdog, and should function mostly the same (except that this watchdog will not detect infinite loops).

Warning: This watchdog is not threadsafe

Watchdog constructor.

Parameters **timeout** (float) – The watchdog's timeout in seconds with microsecond resolution.

addEpoch(*epochName*)
Adds time since last epoch to the list printed by printEpochs().

Epochs are a way to partition the time elapsed so that when overruns occur, one can determine which parts of an operation consumed the most time.

Parameters **epochName** (str) – The name to associate with the epoch.

Return type None

disable()
Disables the watchdog timer.

Return type None

enable()
Enables the watchdog timer.

Return type None

getTime()
Returns the time in seconds since the watchdog was last fed.

Return type float

getTimeout()
Returns the watchdog's timeout in seconds.

Return type float

isExpired()
Returns true if the watchdog timer has expired.

Return type bool

kMinPrintPeriod = 1000000

printIfExpired()
Prints list of epochs added so far and their times.

Return type None

reset()
Resets the watchdog timer.

This also enables the timer if it was previously disabled.

Return type None

setTimeout(*timeout*)
Sets the watchdog's timeout.

Parameters **timeout** (float) – The watchdog's timeout in seconds with microsecond resolution.

Return type None

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

c

commandbased.cancelcommand, 3
commandbased.commandbasedrobot, 3
commandbased.flowcontrol, 3

m

magicbot.magic_reset, 12
magicbot.magic_tunable, 10
magicbot.magiccomponent, 9
magicbot.magicrobot, 6
magicbot.state_machine, 13

r

robotpy_ext.autonomous.selector, 17
robotpy_ext.autonomous.stateful_autonomous, 19
robotpy_ext.common_drivers.distance_sensors, 22
robotpy_ext.common_drivers.driver_base, 23
robotpy_ext.common_drivers.units, 23
robotpy_ext.common_drivers.xl_max_sonar_ez, 23
robotpy_ext.control.button_debouncer, 24
robotpy_ext.control.toggle, 25
robotpy_ext.misc.asyncio_policy, 25
robotpy_ext.misc.looptimer, 26
robotpy_ext.misc.periodic_filter, 27
robotpy_ext.misc.precise_delay, 26
robotpy_ext.misc.simple_watchdog, 28

INDEX

A

addEpoch() (*robotpy_ext.misc.simple_watchdog.SimpleWatchdog method*), 28
addParallel() (*commandbased.flowcontrol.CommandFlow method*), 4
addSequential() (*commandbased.flowcontrol.CommandFlow method*), 4
autonomousInit() (*magicbot.magicrobot.MagicRobot method*), 6
AutonomousModeSelector (*class in robotpy_ext.autonomous.selector*), 17
AutonomousStateMachine (*class in magicbot.state_machine*), 13

B

BREAK() (*in module commandbased.flowcontrol*), 3
ButtonDebouncer (*class in robotpy_ext.control.button_debouncer*), 24

C

CancelCommand (*class in commandbased.cancelcommand*), 3
collect_feedbacks() (*in module magicbot.magic_tunable*), 10
collect_resets() (*in module magicbot.magic_reset*), 12
commandbased.cancelcommand module, 3
commandbased.commandbasedrobot module, 3
commandbased.flowcontrol module, 3
CommandBasedRobot (*class in commandbased.commandbasedrobot*), 3
CommandFlow (*class in commandbased.flowcontrol*), 3
commandPeriodic() (*commandbased.commandbasedrobot.CommandBasedRobot method*), 3
consumeExceptions() (*magicbot.magicrobot.MagicRobot method*), 6
control_loop_wait_time (*magicbot.magicrobot.MagicRobot attribute*), 7
convert() (*in module robotpy_ext.common_drivers.units*), 23
createObjects() (*magicbot.magicrobot.MagicRobot method*), 7
current_state (*magicbot.state_machine.StateMachine attribute*), 15

D

default (*magicbot.magic_reset.will_reset_to attribute*), 13
default_state() (*in module magicbot.state_machine*), 16
disable() (*robotpy_ext.autonomous.selector.AutonomousModeSelector method*), 18
disable() (*robotpy_ext.misc.simple_watchdog.SimpleWatchdog method*), 28
disabledInit() (*magicbot.magicrobot.MagicRobot method*), 7
disabledPeriodic() (*magicbot.magicrobot.MagicRobot method*), 7
done() (*magicbot.state_machine.AutonomousStateMachine method*), 13
done() (*magicbot.state_machine.StateMachine method*), 15
done() (*robotpy_ext.autonomous.stateful_autonomous.StatefulAutonomou method*), 20
DriverBase (*class in robotpy_ext.common_drivers.driver_base*), 23

E

ELIF() (*in module commandbased.flowcontrol*), 5
ELSE() (*in module commandbased.flowcontrol*), 5
enable() (*robotpy_ext.misc.simple_watchdog.SimpleWatchdog method*), 28
endCompetition() (*magicbot.magicrobot.MagicRobot method*), 7
engage() (*magicbot.state_machine.StateMachine method*), 15

error_report_interval (*magicbot.magicrobot.MagicRobot attribute*), 7
execute() (*magicbot.magiccomponent.MagicComponent method*), 9
execute() (*magicbot.state_machine.StateMachine method*), 15

F

feedback() (*in module magicbot.magic_tunable*), 10
filter() (*robotpy_ext.misc.periodic_filter.PeriodicFilter method*), 27
FPGATimedEventLoop (*class in robotpy_ext.misc.asyncio_policy*), 25
FPGATimedEventLoopPolicy (*class in robotpy_ext.misc.asyncio_policy*), 25
free() (*robotpy_ext.misc.precise_delay.NotifierDelay method*), 26

G

get() (*robotpy_ext.common_drivers.xl_max_sonar_ez.MaxSonarEZAnalog method*), 23
get() (*robotpy_ext.common_drivers.xl_max_sonar_ez.MaxSonarEZPulseWidth method*), 24
get() (*robotpy_ext.control.button_debouncer.ButtonDebouncer method*), 24
get() (*robotpy_ext.control.toggle.Toggle method*), 25
getDistance() (*robotpy_ext.common_drivers.distance_sensors.SharpIR2Y0A02 method*), 22
getDistance() (*robotpy_ext.common_drivers.distance_sensors.SharpIR2Y0A21 method*), 22
getDistance() (*robotpy_ext.common_drivers.distance_sensors.SharpIRGP2Y0A41SK0F method*), 22
getTime() (*robotpy_ext.misc.simple_watchdog.SimpleWatchdog method*), 28
getTimeout() (*robotpy_ext.misc.simple_watchdog.SimpleWatchdog method*), 28

H

handleCrash() (*commandbased.commandbasedrobot.CommandBasedRobot method*), 3

I

IF() (*in module commandbased.flowcontrol*), 5
IllegalCallError, 13
initialize() (*commandbased.cancelcommand.CancelCommand method*), 3
InvalidStateName, 13
InvalidWrapperError, 13, 19
is_executing() (*magicbot.state_machine.StateMachine property*), 16
isExpired() (*robotpy_ext.misc.simple_watchdog.SimpleWatchdog method*), 28
isFinished() (*commandbased.cancelcommand.CancelCommand method*), 3

K

kMinPrintPeriod (*robotpy_ext.misc.simple_watchdog.SimpleWatchdog attribute*), 28

L

logger (*magicbot.magiccomponent.MagicComponent attribute*), 9
logger (*magicbot.magicrobot.MagicRobot attribute*), 7
LoopTimer (*class in robotpy_ext.misc.looptimer*), 26

M

magicbot.magic_reset
    module, 12
magicbot.magic_tunable
    module, 10
magicbot.magiccomponent
    module, 9
magicbot.magicrobot
    module, 6
magicbot.state_machine
    module, 13
MagicComponent (*class in magicbot.magiccomponent*), 9
MagicRobot (*class in magicbot.magicrobot*), 6
MaxSonarEZAnalog (*class in robotpy_ext.common_drivers.xl_max_sonar_ez*), 23
MaxSonarEZPulseWidth (*class in robotpy_ext.common_drivers.xl_max_sonar_ez*), 23
measure() (*robotpy_ext.misc.looptimer.LoopTimer method*), 26
module
    commandbased.cancelcommand, 3
    commandbased.commandbasedrobot, 3
    commandbased.flowcontrol, 3
    magicbot.magic_reset, 12
    magicbot.magic_tunable, 10
    magicbot.magiccomponent, 9
    magicbot.magicrobot, 6
    magicbot.state_machine, 13
    robotpy_ext.autonomous.selector, 17
    robotpy_ext.autonomous.stateful_autonomous, 19
    robotpy_ext.common_drivers.distance_sensors, 22
    robotpy_ext.common_drivers.driver_base, 23

robotpy_ext.common_drivers.units, 23
robotpy_ext.common_drivers.xl_max_sonar_ez, 23
robotpy_ext.control.button_debouncer, 24
robotpy_ext.control.toggle, 25
robotpy_ext.misc.asyncio_policy, 25
robotpy_ext.misc.looptimer, 26
robotpy_ext.misc.periodic_filter, 27
robotpy_ext.misc.precise_delay, 26
robotpy_ext.misc.simple_watchdog, 28
MultipleDefaultStatesError, 13
MultipleFirstStatesError, 13

N

next_state() (*magicbot.state_machine.StateMachine method*), 16
next_state() (*robotpy_ext.autonomous.stateful_autonomous.StatefulAutonomous method*), 20
next_state_now() (*magicbot.state_machine.StateMachine method*), 16
NoFirstStateError, 13
NotifierDelay (*class in robotpy_ext.misc.precise_delay*), 26

O

off() (*robotpy_ext.control.toggle.Toggle property*), 25
on() (*robotpy_ext.control.toggle.Toggle property*), 25
on_disable() (*magicbot.magiccomponent.MagicComponent method*), 9
on_disable() (*magicbot.state_machine.StateMachine method*), 16
on_disable() (*robotpy_ext.autonomous.stateful_autonomous.StatefulAutonomous method*), 20
on_enable() (*magicbot.magiccomponent.MagicComponent method*), 10
on_enable() (*magicbot.state_machine.AutonomousStateMachine method*), 13
on_enable() (*magicbot.state_machine.StateMachine method*), 16
on_enable() (*robotpy_ext.autonomous.stateful_autonomous.StatefulAutonomous method*), 20
on_iteration() (*magicbot.state_machine.AutonomousStateMachine method*), 13
on_iteration() (*robotpy_ext.autonomous.stateful_autonomous.StatefulAutonomous method*), 20
onException() (*magicbot.magicrobot.MagicRobot method*), 7

P

patch_asyncio_policy() (*in module robotpy_ext.misc.asyncio_policy*), 26
periodic() (*robotpy_ext.autonomous.selector.AutonomousModeSelector method*), 18
PeriodicFilter (*class in robotpy_ext.misc.periodic_filter*), 27
PreciseDelay (*class in robotpy_ext.misc.precise_delay*), 27
printIfExpired() (*robotpy_ext.misc.simple_watchdog.SimpleWatchdog method*), 28

R

register_sd_var() (*robotpy_ext.autonomous.stateful_autonomous.StatefulAutonomous method*), 20
reset() (*robotpy_ext.misc.looptimer.LoopTimer method*), 26
reset() (*robotpy_ext.misc.simple_watchdog.SimpleWatchdog method*), 29
RETURN() (*in module commandbased.flowcontrol*), 5
robotPeriodic() (*magicbot.magicrobot.MagicRobot method*), 8
robotpy_ext.autonomous.selector module, 17
robotpy_ext.autonomous.stateful_autonomous module, 19
robotpy_ext.common_drivers.distance_sensors module, 22
robotpy_ext.common_drivers.driver_base module, 23
robotpy_ext.common_drivers.units module, 23
robotpy_ext.common_drivers.xl_max_sonar_ez module, 23
robotpy_ext.control.button_debouncer module, 24
robotpy_ext.control.toggle module, 25
robotpy_ext.misc.asyncio_policy module, 25
robotpy_ext.misc.looptimer module, 26
robotpy_ext.misc.periodic_filter module, 27
robotpy_ext.misc.precise_delay module, 26
robotpy_ext.misc.simple_watchdog module, 28
run() (*robotpy_ext.autonomous.selector.AutonomousModeSelector method*), 18

S

set_debounce_period()

(*robotpy_ext.control.button_debouncer.ButtonDebouncer method*), 24
setParent() (*commandbased.flowcontrol.CommandFlow method*), 5
setTimeout() (*robotpy_ext.misc.simple_watchdog.SimpleWatchdog method*), 29
setup() (*magicbot.magiccomponent.MagicComponent method*), 10
setup_tunables() (*in module magicbot.magic_tunable*), 11
SharpIR2Y0A02 (*class in robotpy_ext.common_drivers.distance_sensors*), 22
SharpIR2Y0A21 (*class in robotpy_ext.common_drivers.distance_sensors*), 22
SharpIRGP2Y0A41SK0F (*class in robotpy_ext.common_drivers.distance_sensors*), 22
SimpleWatchdog (*class in robotpy_ext.misc.simple_watchdog*), 28
start() (*commandbased.flowcontrol.CommandFlow method*), 5
start() (*robotpy_ext.autonomous.selector.AutonomousModeSelector method*), 19
startCompetition() (*magicbot.magicrobot.MagicRobot method*), 8
state() (*in module magicbot.state_machine*), 16
state() (*in module robotpy_ext.autonomous.stateful_autonomous*), 21
StatefulAutonomous (*class in robotpy_ext.autonomous.stateful_autonomous*), 19
StateMachine (*class in magicbot.state_machine*), 13

T

teleopInit() (*magicbot.magicrobot.MagicRobot method*), 8
teleopPeriodic() (*magicbot.magicrobot.MagicRobot method*), 8
testInit() (*magicbot.magicrobot.MagicRobot method*), 9
testPeriodic() (*magicbot.magicrobot.MagicRobot method*), 9
time() (*robotpy_ext.misc.asyncio_policy.FPGATimedEventLoop method*), 25
timed_state() (*in module magicbot.state_machine*), 17
timed_state() (*in module robotpy_ext.autonomous.stateful_autonomous*), 21
Toggle (*class in robotpy_ext.control.toggle*), 25
tunable (*class in magicbot.magic_tunable*), 11

U

Unit (*class in robotpy_ext.common_drivers.units*), 23
use_teleop_in_autonomous (*magicbot.magicrobot.MagicRobot attribute*), 9

V

VERBOSE_LOGGING (*magicbot.state_machine.AutonomousStateMachine attribute*), 13
VERBOSE_LOGGING (*magicbot.state_machine.StateMachine attribute*), 15
verified (*robotpy_ext.common_drivers.driver_base.DriverBase attribute*), 23
verified (*robotpy_ext.common_drivers.xl_max_sonar_ez.MaxSonarEZA attribute*), 23
verified (*robotpy_ext.common_drivers.xl_max_sonar_ez.MaxSonarEZH attribute*), 24

W

wait() (*robotpy_ext.misc.precise_delay.NotifierDelay method*), 26
wait() (*robotpy_ext.misc.precise_delay.PreciseDelay method*), 27
WHILE() (*in module commandbased.flowcontrol*), 5
will_reset_to (*class in magicbot.magic_reset*), 12