MPF Documentation Developer Documentation

Release 0.51.0-dev.16

The Mission Pinball Framework Team

Nov 18, 2018

DEVELOPER DOCUMENTATION

1 Understanding the MPF codebase **3**

2 Adding custom code to your machine **5**

3 Common functions to use in your code **7**

4 API Reference **9**

5 Writing Tests **11**

6 Extending, Adding to, and Enhancing MPF **13**

7 BCP Protocol **15**

8 Index **17**

8.1 Overview & Tour of MPF code . . . 17
8.2 Adding custom code to your game . . . 20
8.3 API Reference . . . 28
8.4 Common functions to use in your code . . . 224
8.5 Automated Testing . . . 225
8.6 Extending MPF . . . 230
8.7 BCP Protocol Specification . . . 232
8.8 Method & Class Index . . . 247

This is the developer documentation for the Mission Pinball Framework (MPF), version 0.51. Click the "Read the Docs" link in the lower left corner for other versions & downloads.

This documentation is for people who want to want to add custom Python code & game logic to their machine and for people who want to contribute to MPF itself.

Note: This is DEVELOPER documentation, not general USER documentation!

This documentation is for people writing custom Python code for MPF. If you're a general *user* of MPF, read the MPF User Documentation instead.

This developer documentation is broken into several sections:

CHAPTER 1

Understanding the MPF codebase

- *Overview & Tour of MPF code*
- *MPF Files & Modules*
- *How MPF installs itself*
- *Understanding the MPF boot up / start process*
- *MPF's divergence for pure YAML*

CHAPTER 2

Adding custom code to your machine

- *Adding custom code to your game*
- *How to add machine-wide custom code*
- *How to add custom Python code to a game mode*

CHAPTER 3

Common functions to use in your code

- *Common functions to use in your code*
- *Machine Variables in Code*
- *Player Variables in Code*

CHAPTER 4

API Reference

- *Core Components*
- *Devices*
- *Modes*
- *Config Players*
- *Hardware Platforms*
- *Miscellaneous Components*
- *Testing Class API*

CHAPTER 5

Writing Tests

- *Automated Testing*
- *How to run MPF unittests*
- *Writing Unit Tests for MPF*
- *Writing Custom Tests for your Machine*
- *Fuzz Testing*

CHAPTER 6

Extending, Adding to, and Enhancing MPF

- *Extending MPF*
- *Setting up your MPF Dev Environment*
- *Writing Plugins for MPF*
- *Developing your own hardware interface for MPF*

CHAPTER 7

BCP Protocol

- *BCP Protocol Specification*

CHAPTER 8

Index

- We have an *index* which lists all the classes, methods, and attributes in MPF across the board.

8.1 Overview & Tour of MPF code

This guide provides a general overview of the MPF and MPF-MC codebase.

8.1.1 MPF Files & Modules

The MPF packages contains the following folders:

- `/build_scripts`: Scripts which can be used to locally build & test MPF packages and wheels
- `/docs`: The Sphinx-based developer docs that you're reading now
- `/mpf`: The actual mpf package that's copied to your machine when MPF is installed
- `/tools`: A few random tools

The MPF package

The MPF package (e.g. the `/mpf` subfolder which is copied to your install location when you install MPF) contains the following folders:

- `/assets`: Contains the asset classes used in MPF (the "shows" asset class)
- `/commands`: Modules for the command-line interface for MPF
- `/config_players`: Modules for the built-in config_players
- `/core`: Core MPF system modules

- /devices: Device modules
- /exceptions: MPF exception classes
- /file_interfaces: MPF file interfaces (current just YAML, could support more in the future)
- /migrator: MPF Migrator files
- /modes: Code for built-in modes (game, attract, tilt, credits, etc.)
- /platforms: Hardware platform modules
- /plugins: Built-in MPF plugins
- /tests: MPF unit tests

It also includes the following files in the package root:

- __init__.py: Makes the MPF folder a package
- __main__.py: Allows the MPF commands to run
- _version.py: Contains version strings used throughout MPF for the current version
- mpfconfig.yaml: The "base" machine config file that is used for all machines (unless this is specifically overridden via the command-line options)

8.1.2 How MPF installs itself

This guide explains what happens when MPF is installed.

MPF contains a setup.py file in the root of the MPF repository. This is the file that's called by *pip* when MPF is installed. (You can also install MPF without using pip by running python3 setup.py from the root folder.)

Dependencies

MPF requires Python 3.4 or newer. In our installation instructions, we also recommend that users install/update the following Python packages to their latest versions:

- pip
- setuptools (for Linux & Mac)
- Cython 0.24.1 (for Linux & Mac)

The additional packages for Linux & Mac are used because MPF-MC is actually compiled on built on those platforms. For Windows we have pre-built wheels, so compiling is not necessary.

MPF has the following additional dependencies which are specified in the setup.py file and automatically installed when MPF is installed.

- ruamel.yaml >=0.10,<0.11: Used for reading & writing YAML files.
- pyserial >= 3.2.0: Used for serial communication with several types of hardware
- pyserial-asyncio >= 0.3: Also used for serial communication
- typing Used for type-checking & type hinting.

Note that some of these dependencies will install their own dependencies.

The setup.py file also specifies a console_scripts entry point called mpf. This is what lets the user type mpf from the command environment to launch MPF.

8.1.3 Understanding the MPF boot up / start process

A user runs "mpf" from the command line, which is registered as a console script entry point when MPF is installed. That entry point calls the function `run_from_command_line()` in `mpf.commands.__init__` module.

That module parses the command line arguments, figures out the machine path that's being executed, and figures out which MPF command is being called. (MPF commands are things like "both" or "mc".)

Some commands are built-in to MPF (in the `mpf/commands` folder), and others are registered as MPF via plugin entry points when other packages are installed. (For example, MPF-MC registers the "mc" command, the MPF Monitor registers the "monitor" command, etc.)

When you launch MPF (via `mpf game` or just plain `mpf`), the `mpf.commands.game` module's `Command` class is instantiated. This class processes the command line arguments, sets up logging, and then creates an instance of the `mpf.core.machine.MachineController` class.

(This class is run inside a `try:` block, with all exceptions captured and then sent to the log. This is how MPF is able to capture crashes and stack traces into the log file when it crashes.

The Machine Controller

The Machine Controller can be thought of as the main "kernel" of MPF. It does a lot of things, including:

- Loading, merging, & validating the config files
- Setting up the clock
- Loading platform modules (based on what's used in the configs)
- Loading MPF core modules
- Loading MPF plugins
- Loading custom machine code
- Stepping through the initialization and reset phases

8.1.4 MPF's divergence for pure YAML

MPF uses the YAML file format for config and show files. That said, MPF diverges from the pure YAML 1.2 specification for unquoted strings in a few ways. Those are cases where YAML guesses which data type the value is which led to problems/confusion in the past:

Values beginning with "+" are strings

> The YAML spec essentially ignores a leading plus sign, so a value `+1` would be read in as the integer `1`. However MPF needs to differentiate between `+1` and `1` since the plus sign is used to mean the value is a delta in certain situations, so MPF's YAML interfaces will process any numeric values with a leading plus sign as strings.

Values beginning with a leading "0" are strings

> The YAML spec will process values that are only digits 0-7 with leading zeros as octals. However MPF could have color values like `050505` which should be read as strings. So the MPF YAML interface processes any value with at least 3 digits and leading zeros as strings.

"On" and "Off" values are strings

The YAML spec defines `on` and `off` values as bools. But many MPF users create show names called "on" and "off", so MPF's YAML processor interprets those as strings. (True, False, Yes, and No are still processes as bools.)

Values with only digits and "e" are strings

The YAML spec will process a value like `123e45` as "123 exponent 45". Since those could be hex color codes, MPF's YAML interface processes values that are all digits with a single "e" character as strings.

8.2 Adding custom code to your game

While one of the goals of MPF is to allow you to do as much of your game's configuration as possible with the config files, we recognize that many people will want to mix in some custom code to their machines.

Fortunately that's easy to do, and you don't have to "hack" MPF or break anything to make it happen!

The amount of custom code you use is up to you, depending on your personal preferences, your comfort with Python, and what exactly you want to do with your machine.

Some people will use the config files for 99% of their machine, and only add a little custom code here and there. Others will only want to use the configs for the "basic" stuff and then write all their game logic in Python. Either option is fine with us!

When you decide that you want to add some custom Python code into your game, there are three ways you can do this:

- *Mode-specific code*, which allows you to write custom Python code which is only active when a particular game mode is active.
- *Machine-wide code*, useful for dealing with custom hardware, like the crane in *Demolition Man*.

8.2.1 How to add custom Python code to a game mode

The easiest and most common way to add custom Python code into your MPF game is to add a code module to a mode folder. That lets you run code when that mode is active and helps you break up any custom code you write per mode.

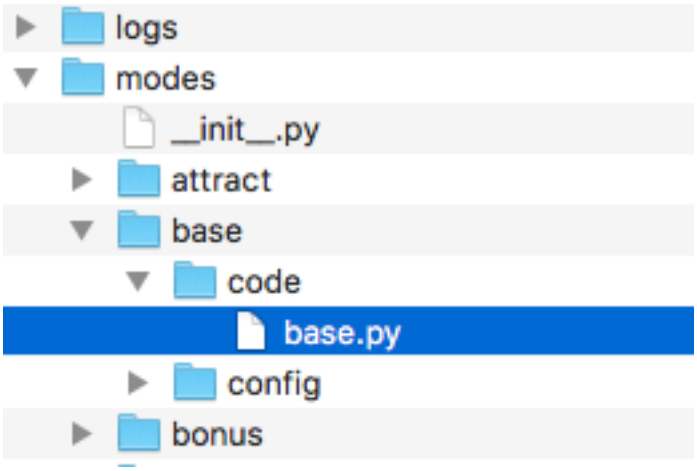
This "mode code" (as we call it) has access to the full MPF API. You can post events, register event handlers which run custom things when events are posted, access device state and control devices, read and set player variables, post slides... really anything MPF can do, you can do.

Here's how you get started with custom mode code:

1. Create the module (file) to hold you code

First, go into the folder where you want to create your custom code, and add a "code" folder to that mode's folder. Then inside that folder, create a file (we usually give this file the same name as the mode) with a `.py` extension.

For example, if you wanted to create custom code for your base mode, it would look like this:



2. Open up the new Python file you just created

Next, open the new mode code Python file you just created and add the bare minimum, which would look like this:

```
from mpf.core.mode import Mode

class Base(Mode):
    pass
```

MPF includes a `Mode` class which acts as the base class for every mode that runs in a game. That base class lives in the MPF package at `mpf.core.mode`. You can see it online in GitHub here.

Notice that we named our custom class `Base`. You can name it whatever you want.

3. Update your mode config file to use the custom code

Once you create your custom mode code, you need to tell MPF that this mode uses custom code instead of just the built-in code.

To do this, add a `code:` entry into the mode config file for the mode where you're adding custom code. So in this case, that would be in the /modes/base/config/base.yaml file, like this:

```
mode:
  start_events: ball_starting
  priority: 100
  code: base.Base
```

Note that the value for the `code:` section is the name of the Python module (the file), then a dot, then the name of the class from that file. So in this case, that's `base.Base`.

4. Run your game!

At this point you should be able to run your game and nothing should happen. This is good, because if it doesn't crash, that means you did everything right. :) Of course nothing special happens because you didn't actually add any code to your custom mode code, so you won't see anything different.

5. Add some custom methods to do things

You can look at the Mode base class (the link from GitHub from earlier) to see what the base Mode class does. However, we have created a few "convenience" methods that you can use. They are:

mode_init Called once when MPF is starting up

mode_start Called every time the mode starts, just *after* the *mode_<name>_started* event is posted.

mode_stop Called every time the mode stops, just *before* the *mode_<name>_stopping* event is posted.

add_mode_event_handler This is the same as the main `add_event_handler()` method from the Event Manager, except since it's mode-specific it will *also* automatically remove any event handlers that you registered when the mode stops. (If you want to register event handlers that are always watching for events even when the mode is not running, you can use the regular `self.machine.mode.add_handler()` method.

You don't have to use all of these if you don't want to.

Also, modes have additional convenience attributes you can use within your mode code:

self.config A link to the config dictionary for the mode's config file.

self.priority The priority the mode is running at. (Don't change this. Just read it.)

self.delay An instance of the delay manager you can use to set delayed callbacks for this mode. Any active ones will be automatically removed when the mode ends.

self.player A link to the current player object that's automatically updated when the player changes. This will be `None` if the mode is running outside of a game.

self.active A boolean (True/False) value you can query to see if the mode is running.

6. Example usage

Here's an example of some mode code in use. This example is just a bunch of random things, but again, since you're writing code here, the sky's the limit! Seriously you could do all your game logic in mode code and not use the MPF configs at all if you wanted to.

```
from mpf.core.mode import Mode


class Base(Mode):

    def mode_init(self):
        print("My custom mode code is being initialized")

    def mode_start(self, **kwargs):
        # The mode_start method needs **kwargs because some events that
        # start modes pass additional parameters

        print("My custom mode code is starting")

        # call a delay in 5 seconds
        self.delay.add(5000, self.my_callback)

        # what player are we?
        print(self.player.number)

        # what's the player's score?
```

(continues on next page)

(continued from previous page)

```
        print('Score: {}'.format(self.player.score))

        self.add_mode_event_handler('player_score', self.player_score_change)

        # turn LED "led01" red
        self.machine.leds.led01.color('red')

    def my_callback(self):
        print("My delayed call was just called!")

    def player_score_change(self, **kwargs):
        print("The new player's score is {}".format(self.player.score))

    def mode_stop(self, **kwargs):
        # The mode_stop method needs **kwargs because some events that
        # stop modes pass additional parameters

        print("My custom mode code is stopping")
```

You can use the API reference (or just look at the source code) to see what options exist. Really you can do anything you want.

8.2.2 How to add machine-wide custom code

MPF contains a "CustomCode" concept which lets you add custom code to your game.

CustomCode classes are Python modules that run at the "root" of your game. You can use them to do anything you want.

Note that MPF also has the ability to run custom *mode code* which is code that is associated with a certain game mode and is generally only active when the mode it's in is active. So if you just want to write your own custom game logic, you'll probably use mode code.

CustomCode classes, on the other hand, are sort of "machine-level" custom code. CustomCode classes are nice if you have some kind of custom device type that doesn't match up to any of MPF's built in devices. The elevator and claw unloader in *Demolition Man* is a good example, and what we'll use here.

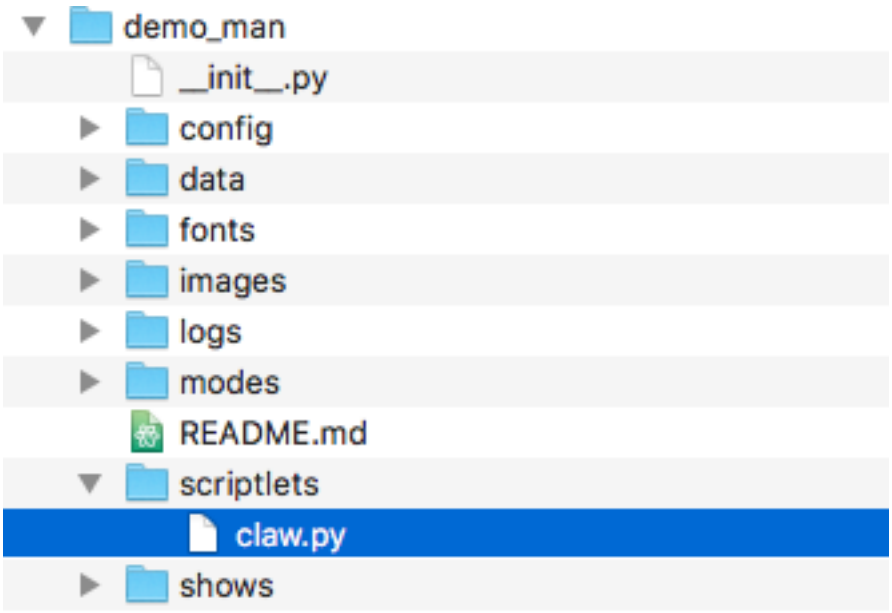
(You can read about how to download and run *Demo Man* in the example games section section of the MPF User Documentation.)

Here's how to create a custom code class:

1. Create your custom code file

First, add a `code` folder to your machine folder (you can use another name if you want). Then inside there, create the Python file that will hold your custom code classes. You can name this file whatever you want, just remember the name for the next step.

In the *Demo Man* example, it looks like this:



Add an empty `__init__.py` file into your folder to make it a package. It become the package `code` and all your classes will be referenced as `code.file_name.ClassName`.

2. Open and edit your custom code class file

Next, edit the class file you created. At a bare minimum, you'll need this:

```
from mpf.core.custom_code import CustomCode


class Claw(CustomCode):
    pass
```

Note that MPF contains a `CustomCode` base class which is very simple. (You can see the source of it on GitHub here.) We called our class `Claw` in this case.

Pretty much all this does is give you a reference to the main MPF machine controller at `self.machine`, as well as setup a delay manager you can use and set the name of your class. There's also an `on_load()` method which is called when the class is loaded which you can use in your own code.

3. Add the class to your machine config

Next, edit your machine config file and add a `custom_code:` section, then under there add the package (folder), followed by a dot, then the module (file name) for your class, followed by a dot, followed by the class name for your class.

For *Demo Man*, that looks like this:

```
custom_code:
  - code.claw.Claw
```

This references class `Claw` in file `claw`.py which lives package `code`.

4. Real-world example

At this point you should be able to run your game, though nothing should happen because you haven't added any code to your code.

Take a look at the final *Demo Man* claw class to see what we did there. Since custom code classes have access to `self.machine` and they load when MPF loads, you can do anything you want in them.

```
"""Claw controller for Demo Man"""

from mpf.core.custom_code import CustomCode


class Claw(CustomCode):

    def on_load(self):

        self.auto_release_in_progress = False

        # if the elevator switch is active for more than 100ms, that means
        # a ball is there, so we want to get it and deliver it to the claw
        self.machine.switch_controller.add_switch_handler(
            's_elevator_hold', self.get_ball, ms=100)

        # This is a one-time thing to check to see if there's a ball in
        # the elevator when MPF starts, and if so, we want to get it.
        if self.machine.switch_controller.is_active('s_elevator_hold'):
            self.auto_release_in_progress = True
            self.get_ball()

        # We'll use the event 'light_claw' to light the claw, so in the
        # future all we have to do is post this event and everything else
        # will be automatic.
        self.machine.events.add_handler('light_claw', self.light_claw)

    def enable(self):
        """Enable the claw."""

        # move left & right with the flipper switches, and stop moving when
        # they're released

        self.machine.switch_controller.add_switch_handler(
            's_flipper_lower_left', self.move_left)
        self.machine.switch_controller.add_switch_handler(
            's_flipper_lower_left', self.stop_moving, state=0)
        self.machine.switch_controller.add_switch_handler(
            's_flipper_lower_right', self.move_right)
        self.machine.switch_controller.add_switch_handler(
            's_flipper_lower_right', self.stop_moving, state=0)

        # release the ball when the launch button is hit
        self.machine.switch_controller.add_switch_handler(
            's_ball_launch', self.release)

        # stop moving if the claw hits a limit switch
        self.machine.switch_controller.add_switch_handler(
            's_claw_position_1', self.stop_moving)
```

(continues on next page)

(continued from previous page)

```
        # We can use this event for slides to explain what's going on for
        # the player.
        self.machine.events.post('claw_enabled')

    def disable(self):
        """Disable the claw."""

        self.stop_moving()

        # remove all the switch handlers
        self.machine.switch_controller.remove_switch_handler(
            's_flipper_lower_left', self.move_left)
        self.machine.switch_controller.remove_switch_handler(
            's_flipper_lower_left', self.stop_moving, state=0)
        self.machine.switch_controller.remove_switch_handler(
            's_flipper_lower_right', self.move_right)
        self.machine.switch_controller.remove_switch_handler(
            's_flipper_lower_right', self.stop_moving, state=0)
        self.machine.switch_controller.remove_switch_handler(
            's_ball_launch', self.release)
        self.machine.switch_controller.remove_switch_handler(
            's_claw_position_1', self.stop_moving)
        self.machine.switch_controller.remove_switch_handler(
            's_claw_position_1', self.release, state=0)
        self.machine.switch_controller.remove_switch_handler(
            's_claw_position_2', self.release)

        self.machine.events.post('claw_disabled')

    def move_left(self):
        """Start the claw moving to the left."""
        # before we turn on the driver to move the claw, make sure we're not
        # at the left limit
        if (self.machine.switch_controller.is_active('s_claw_position_2') and
                self.machine.switch_controller.is_active('s_claw_position_1')):
            return
        self.machine.coils['c_claw_motor_left'].enable()

    def move_right(self):
        """Start the claw moving to the right."""
        # before we turn on the driver to move the claw, make sure we're not
        # at the right limit
        if (self.machine.switch_controller.is_active('s_claw_position_1') and
                self.machine.switch_controller.is_inactive('s_claw_position_2')):
            return
        self.machine.coils['c_claw_motor_right'].enable()

    def stop_moving(self):
        """Stop the claw moving."""
        self.machine.coils['c_claw_motor_left'].disable()
        self.machine.coils['c_claw_motor_right'].disable()

    def release(self):
        """Release the ball by disabling the claw magnet."""
        self.disable_claw_magnet()
        self.auto_release_in_progress = False
```

(continues on next page)

(continued from previous page)

```
        # Disable the claw since it doesn't have a ball anymore
        self.disable()

    def auto_release(self):
        """Aumatically move and release the ball."""
        # disable the switches since the machine is in control now
        self.disable()

        # If we're at the left limit, we need to move right before we can
        # release the ball.
        if (self.machine.switch_controller.is_active('s_claw_position_2') and
                self.machine.switch_controller.is_active('s_claw_position_1')):
            self.machine.switch_controller.add_switch_handler(
                's_claw_position_1', self.release, state=0)
            # move right, drop when switch 1 opens
            self.move_right()

        # If we're at the right limit, we need to move left before we can
        # release the ball
        elif (self.machine.switch_controller.is_active('s_claw_position_1') and
                self.machine.switch_controller.is_inactive('s_claw_position_2')):
            self.machine.switch_controller.add_switch_handler(
                's_claw_position_2', self.release)
            # move left, drop when switch 2 closes
            self.move_left()

        # If we're not at any limit, we can release the ball now.
        else:
            self.release()

    def get_ball(self):
        """Get a ball from the elevator."""

        # If there's no game in progress, we're going to auto pickup and
        # drop the ball with no player input

        if not self.machine.game:
            self.auto_release_in_progress = True

        # If the claw is not already in the ball pickup position, then move it
        # to the right.
        if not (self.machine.switch_controller.is_active('s_claw_position_1') and
                self.machine.switch_controller.is_inactive('s_claw_position_2')):
            self.move_right()

            self.machine.switch_controller.add_switch_handler(
                's_claw_position_1', self.do_pickup)

        # If the claw is in position for a pickup, we can do that pickup now
        else:
            self.do_pickup()

    def do_pickup(self):
        """Pickup a ball from the elevator"""
        self.stop_moving()
        self.machine.switch_controller.remove_switch_handler(
            's_claw_position_1', self.do_pickup)
```

(continues on next page)

(continued from previous page)

```
        self.enable_claw_magnet()
        self.machine.coils['c_elevator_motor'].enable()
        self.machine.switch_controller.add_switch_handler('s_elevator_index',
                                                          self.stop_elevator)

        # If this is not an auto release, enable control of the claw for the
        # player
        if not self.auto_release_in_progress:
            self.enable()

    def stop_elevator(self):
        """Stop the elevator."""
        self.machine.coils['c_elevator_motor'].disable()

        if self.auto_release_in_progress:
            self.auto_release()

    def light_claw(self, **kwargs):
        """Lights the claw."""

        # Lighting the claw just enables the diverter so that the ball shot
        # that way will go to the elevator. Once the ball hits the elevator,
        # the other methods kick in to deliver it to the claw, and then once
        # the claw has it, the player can move and release it on their own.
        self.machine.diverters['diverter'].enable()

    def disable_claw_magnet(self):
        """Disable the claw magnet."""
        self.machine.coils['c_claw_magnet'].disable()

    def enable_claw_magnet(self):
        """Enable the claw magnet."""
        self.machine.coils['c_claw_magnet'].enable()
```

8.3 API Reference

MPF's API reference is broken into several categories. All of it is presented in the way that the modules and classes are actually used in MPF.

Core Components

MPF core components.

Devices

MPF devices, including physical devices like flippers, ball devices, switches, lights, etc. as well as logical devices like ball saves, extra balls, multiballs, etc.

Modes

Built-in modes, such as game, attract, tilt, credits, etc.

Platforms

Hardware platforms interfacess for all supported hardware.

Config Players

Modules responsible for all config players (show_player, light_player, score_player, etc.)

Tests

All unit test base classes for writing tests for MPF and your own game.

Miscellaneous Components

Things that don't fit into other categories, including utility functions, the base classes for modes, players, timers, and other utility functions.

8.3.1 Core Components

Core MPF machine components, accessible to programmers at `self.machine.*name*`. For example, the ball controller is at `self.machine.ball_controller`, the event manager is `self.machine.events`, etc.

self.machine.asset_manager

class mpf.core.assets.**AsyncioSyncAssetManager**(*machine: mpf.core.machine.MachineController*)
Bases: mpf.core.assets.BaseAssetManager

AssetManager which uses asyncio to load assets.

Accessing the asset_manager in code

There is only one instance of the asset_manager in MPF, and it's accessible via `self.machine.asset_manager`.

Methods & Attributes

The asset_manager has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

load_asset(*asset*)
Load an asset.

wait_for_asset_load(*asset*)
Wait for an asset to load.

self.machine.auditor

class mpf.plugins.auditor.**Auditor**(*machine: MachineController*)
Bases: object

Writes switch events, regular events, and player variables to an audit log file.

Accessing the auditor in code

There is only one instance of the auditor in MPF, and it's accessible via `self.machine.auditor`.

Methods & Attributes

The auditor has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

audit(*audit_class*, *event*, ***kwargs*)
Log an auditable event.

Parameters

- **audit_class** – A string of the section we want this event to be logged to.
- **event** – A string name of the event we're auditing.
- ****kwargs** – Not used, but included since some of the audit events might include random kwargs.

audit_event(*eventname*, ***kwargs*)
Record this event in the audit log.

Parameters

- **eventname** – The string name of the event.
- ****kwargs** – not used, but included since some types of events include kwargs.

audit_player(***kwargs*)
Write player data to the audit log.

Typically this is only called at the end of a game.

Parameters ****kwargs** – not used, but included since some types of events include kwargs.

audit_shot(*name*, *profile*, *state*)
Record shot hit.

audit_switch(*change: mpf.core.switch_controller.MonitoredSwitchChange*)
Record switch change.

disable(***kwargs*)
Disable the auditor.

enable(***kwargs*)
Enable the auditor.

This method lets you enable the auditor so it only records things when you want it to. Typically this is called at the beginning of a game.

Parameters ****kwargs** – No function here. They just exist to allow this method to be registered as a handler for events that might contain keyword arguments.

enabled
Attribute that's viewed by other core components to let them know they should send auditing events. Set this via the enable() and disable() methods.

self.machine.ball_controller

class mpf.core.ball_controller.**BallController**(*machine: mpf.core.machine.MachineController*)
Bases: mpf.core.mpf_controller.MpfController

Tracks and manages all the balls in a pinball machine.

Accessing the ball_controller in code

There is only one instance of the ball_controller in MPF, and it's accessible via `self.machine.ball_controller`.

Methods & Attributes

The ball_controller has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

add_captured_ball(*source: mpf.devices.ball_device.ball_device.BallDevice*) → None
Inform ball controller about a capured ball (which might be new).

are_balls_collected(*target: Iterable[str]*) → bool
Check to see if all the balls are contained in devices tagged with the parameter that was passed.

Note if you pass a target that's not used in any ball devices, this method will return True. (Because you're asking if all balls are nowhere, and they always are. :)

Parameters **target** – String or list of strings of the tags you'd like to collect the balls to. Default of None will be replaced with 'home' and 'trough'.

collect_balls(*target='home', trough'*) → None
Ensure that all balls are in contained in ball devices with the tag or list of tags you pass.

Typically this would be used after a game ends, or when the machine is reset or first starts up, to ensure that all balls are in devices tagged with 'home' and/or 'trough'.

Parameters **target** – A string of the tag name or a list of tags names of the ball devices you want all the balls to end up in. Default is ['home', 'trough'].

dump_ball_counts() → None
Dump ball count of all devices.

request_to_start_game(***kwargs*) → bool
Handle result of the *request_to_start_game* event.

Checks to make sure that the balls are in all the right places and returns. If too many balls are missing (based on the config files 'Min Balls' setting), it will return False to reject the game start request.

self.machine.bcp

class mpf.core.bcp.bcp.**Bcp**(*machine: MachineController*)
Bases: mpf.core.mpf_controller.MpfController

BCP Module.

Accessing the bcp in code

There is only one instance of the bcp in MPF, and it's accessible via `self.machine.bcp`.

Methods & Attributes

The bcp has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

send(*bcp_command*, ***kwargs*)
Emulate legacy send.

Parameters **bcp_command** – Commmand to send

self.machine.device_manager

class mpf.core.device_manager.**DeviceManager**(*machine*)
Bases: mpf.core.mpf_controller.MpfController

Manages all the devices in MPF.

Accessing the device_manager in code

There is only one instance of the device_manager in MPF, and it's accessible via `self.machine.device_manager`.

Methods & Attributes

The device_manager has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

create_devices(*collection_name*, *config*)
Create devices for a collection.

create_machinewide_device_control_events(***kwargs*)
Create machine wide control events.

get_device_control_events(*config*)
Scan a config dictionary for control_events.

Yields events, methods, delays, and devices for all the devices and control_events in that config.

Parameters **config** – An MPF config dictionary (either machine-wide or mode- specific).

Returns

- The event name
- The callback method of the device
- The delay in ms
- The device object

Return type A generator of 4-item tuples

get_monitorable_devices()
Return all devices which are registered as monitorable.

initialize_devices()
Initialise devices.

load_devices_config(*validate=True*)
Load all devices.

notify_device_changes(*device*, *notify*, *old*, *value*)
Notify subscribers about changes in a registered device.

Parameters

- **device** – The device that changed.
- **notify** –
- **old** – The old value.
- **value** – The new value.

register_monitorable_device(*device*)
Register a monitorable device.

Parameters **device** – The device to register.

stop_devices()
Stop all devices in the machine.

self.machine.events

class mpf.core.events.**EventManager**(*machine: MachineController*)
Bases: mpf.core.mpf_controller.MpfController

Handles all the events and manages the handlers in MPF.

Accessing the events in code

There is only one instance of the events in MPF, and it's accessible via `self.machine.events`.

Methods & Attributes

The events has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

add_async_handler(*event: str*, *handler: Any*, *priority: int = 1*, *blocking_facility: Any = None*, ***kwargs*) → mpf.core.events.EventHandlerKey
Register a coroutine as event handler.

add_handler(*event: str*, *handler: Any*, *priority: int = 1*, *blocking_facility: Any = None*, ***kwargs*)
→ mpf.core.events.EventHandlerKey
Register an event handler to respond to an event.

Parameters

- **event** – String name of the event you're adding a handler for. Since events are text strings, they don't have to be pre-defined.

- **handler** – The callable method that will be called when the event is fired. Since it's possible for events to have kwargs attached to them, the handler method must include `**kwargs` in its signature.
- **priority** – An arbitrary integer value that defines what order the handlers will be called in. The default is 1, so if you have a handler that you want to be called first, add it here with a priority of 2. (Or 3 or 10 or 100000.) The numbers don't matter. They're called from highest to lowest. (i.e. priority 100 is called before priority 1.)
- ****kwargs** – Any any additional keyword/argument pairs entered here will be attached to the handler and called whenever that handler is called. Note these are in addition to kwargs that could be passed as part of the event post. If there's a conflict, the event-level ones will win.

Returns A GUID reference to the handler which you can use to later remove the handler via `remove_handler_by_key`.

For example:

```
my_handler = self.machine.events.add_handler('ev', self.test))
```

Then later to remove all the handlers that a module added, you could: for handler in handler_list: `events.remove_handler(my_handler)`

does_event_exist(*event_name: str*) → bool

Check to see if any handlers are registered for the event name that is passed.

Parameters **event_name** – The string name of the event you want to check.

Returns True or False

get_event_and_condition_from_string(*event_string: str*) → Tuple[str, Optional[BaseTemplate]]

Parse an event string to divide the event name from a possible placeholder / conditional in braces.

Parameters **event_string** – String to parse

Returns

First item is the event name

Second item is the condition (A BoolTemplate instance) if it exists, or None if it doesn't.

Return type 2-item tuple

post(*event: str*, *callback=None*, ***kwargs*) → None

Post an event which causes all the registered handlers to be called.

Events are processed serially (e.g. one at a time), so if the event core is in the process of handling another event, this event is added to a queue and processed after the current event is done.

You can control the order the handlers will be called by optionally specifying a priority when the handlers were registered. (Higher priority values will be processed first.)

Parameters

- **event** – A string name of the event you're posting. Note that you can post whatever event you want. You don't have to set up anything ahead of time, and if no handlers are registered for the event you post, so be it.
- **callback** – An optional method which will be called when the final handler is done processing this event. Default is None.

- ****kwargs** – One or more options keyword/value pairs that will be passed to each handler. (The event manager will enforce that handlers have `**kwargs` in their signatures when they're registered to prevent run-time crashes from unexpected kwargs that were included in `post()` calls.

post_async(*event: str*, ***kwargs*) → asyncio.futures.Future
Post event and wait until all handlers are done.

post_boolean(*event: str*, *callback=None*, ***kwargs*) → None
Post an boolean event which causes all the registered handlers to be called one-by-one.

Boolean events differ from regular events in that if any handler returns False, the remaining handlers will not be called.

Events are processed serially (e.g. one at a time), so if the event core is in the process of handling another event, this event is added to a queue and processed after the current event is done.

You can control the order the handlers will be called by optionally specifying a priority when the handlers were registered. (Higher priority values will be processed first.)

Parameters

- **event** – A string name of the event you're posting. Note that you can post whatever event you want. You don't have to set up anything ahead of time, and if no handlers are registered for the event you post, so be it.
- **callback** – An optional method which will be called when the final handler is done processing this event. Default is None. If any handler returns False and cancels this boolean event, the callback will still be called, but a new kwarg ev_result=False will be passed to it.
- ****kwargs** – One or more options keyword/value pairs that will be passed to each handler.

post_queue(*event*, *callback*, ***kwargs*)
Post a queue event which causes all the registered handlers to be called.

Queue events differ from standard events in that individual handlers are given the option to register a "wait", and the callback will not be called until any handler(s) that registered a wait will have to release that wait. Once all the handlers release their waits, the callback is called.

Events are processed serially (e.g. one at a time), so if the event core is in the process of handling another event, this event is added to a queue and processed after the current event is done.

You can control the order the handlers will be called by optionally specifying a priority when the handlers were registered. (Higher numeric values will be processed first.)

Parameters

- **event** – A string name of the event you're posting. Note that you can post whatever event you want. You don't have to set up anything ahead of time, and if no handlers are registered for the event you post, so be it.
- **callback** – The method which will be called when the final handler is done processing this event and any handlers that registered waits have cleared their waits.
- ****kwargs** – One or more options keyword/value pairs that will be passed to each handler. (Just make sure your handlers are expecting them. You can add `**kwargs` to your handler methods if certain ones don't need them.)

Examples

Post the queue event called *pizza_time*, and then call `self.pizza_done` when done:

```
self.machine.events.post_queue('pizza_time', self.pizza_done)
```

post_queue_async(*event: str*, ***kwargs*) → asyncio.futures.Future
Post queue event, wait until all handlers are done and locks are released.

post_relay(*event: str*, *callback=None*, ***kwargs*) → None
Post a relay event which causes all the registered handlers to be called.

A dictionary can be passed from handler-to-handler and modified as needed.

Parameters

- **event** – A string name of the event you're posting. Note that you can post whatever event you want. You don't have to set up anything ahead of time, and if no handlers are registered for the event you post, so be it.
- **callback** – The method which will be called when the final handler is done processing this event. Default is None.
- ****kwargs** – One or more options keyword/value pairs that will be passed to each handler. (Just make sure your handlers are expecting them. You can add `**kwargs` to your handler methods if certain ones don't need them.)

Events are processed serially (e.g. one at a time), so if the event core is in the process of handling another event, this event is added to a queue and processed after the current event is done.

You can control the order the handlers will be called by optionally specifying a priority when the handlers were registered. (Higher priority values will be processed first.)

Relay events differ from standard events in that the resulting kwargs from one handler are passed to the next handler. (In other words, standard events mean that all the handlers get the same initial kwargs, whereas relay events "relay" the resulting kwargs from one handler to the next.)

post_relay_async(*event: str*, ***kwargs*) → asyncio.futures.Future
Post relay event, wait until all handlers are done and return result.

process_event_queue() → None
Check if there are any other events that need to be processed, and then process them.

remove_all_handlers_for_event(*event: str*) → None
Remove all handlers for event.

Use carefully. This is currently used to remove handlers for all init events which only occur once.

remove_handler(*method: Any*) → None
Remove an event handler from all events a method is registered to handle.

Parameters **method** – The method whose handlers you want to remove.

remove_handler_by_event(*event: str*, *handler: Any*) → None
Remove the handler you pass from the event you pass.

Parameters

- **event** – The name of the event you want to remove the handler from.
- **handler** – The handler method you want to remove.

Note that keyword arguments for the handler are not taken into consideration. In other words, this method only removes the registered handler / event combination, regardless of whether the keyword arguments match or not.

remove_handler_by_key(*key: mpf.core.events.EventHandlerKey*) → None
Remove a registered event handler by key.

Parameters **key** – The key of the handler you want to remove

remove_handlers_by_keys(*key_list: List[mpf.core.events.EventHandlerKey]*) → None
Remove multiple event handlers based on a passed list of keys.

Parameters **key_list** – A list of keys of the handlers you want to remove

replace_handler(*event: str*, *handler: Any*, *priority: int = 1*, ***kwargs*) → mpf.core.events.EventHandlerKey
Check to see if a handler (optionally with kwargs) is registered for an event and replaces it if so.

Parameters

- **event** – The event you want to check to see if this handler is registered for.
- **handler** – The method of the handler you want to check.
- **priority** – Optional priority of the new handler that will be registered.
- ****kwargs** – The kwargs you want to check and the kwargs that will be registered with the new handler.

If you don't pass kwargs, this method will just look for the handler and event combination. If you do pass kwargs, it will make sure they match before replacing the existing entry.

If this method doesn't find a match, it will still add the new handler.

wait_for_any_event(*event_names: List[str]*) → asyncio.futures.Future
Wait for any event from event_names.

wait_for_event(*event_name: str*) → asyncio.futures.Future
Wait for event.

self.machine.info_lights

class mpf.plugins.info_lights.**InfoLights**(*machine*)
Bases: object

Uses lights to represent game state.

Info lights are primarily used in EM and early solid state machines, typically lights in the backbox for game over, tilt, which player is up, the current ball number, etc.

Accessing the info_lights in code

There is only one instance of the info_lights in MPF, and it's accessible via `self.machine.info_lights`.

Methods & Attributes

The info_lights has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

self.machine.light_controller

class mpf.core.light_controller.**LightController**(*machine: mpf.core.machine.MachineController*)

Bases: mpf.core.mpf_controller.MpfController

Handles light updates and light monitoring.

Accessing the light_controller in code

There is only one instance of the light_controller in MPF, and it's accessible via `self.machine.light_controller`.

Methods & Attributes

The light_controller has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

initialise_light_subsystem()
Initialise the light subsystem.

monitor_lights()
Update the color of lights for the monitor.

self.machine

class mpf.core.machine.**MachineController**(*mpf_path: str*, *machine_path: str*, *options: dict*)

Bases: *mpf.core.logging.LogMixin*

Base class for the Machine Controller object.

The machine controller is the main entity of the entire framework. It's the main part that's in charge and makes things happen.

Parameters

- **options** (*dict*) – A dictionary of options built from the command line options used to launch mpf.py.
- **machine_path** – The root path of this machine_files folder

Accessing the machine controller in code

The machine controller is the main component in MPF, accessible via `self.machine`. See the *Overview & Tour of MPF code* for details.

Methods & Attributes

The machine controller has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

add_platform(*name: str*) → None
Make an additional hardware platform interface available to MPF.

Parameters **name** – String name of the platform to add. Must match the name of a platform file in the mpf/platforms folder (without the .py extension).

clear_boot_hold(*hold: str*) → None
Clear a boot hold.

configure_machine_var(*name: str*, *persist: bool*, *expire_secs: int = None*) → None
Create a new machine variable.

Parameters

- **name** – String name of the variable.
- **persist** – Boolean as to whether this variable should be saved to disk so it's available the next time MPF boots.
- **expire_secs** – Optional number of seconds you'd like this variable to persist on disk for. When MPF boots, if the expiration time of the variable is in the past, it will not be loaded. For example, this lets you write the number of credits on the machine to disk to persist even during power off, but you could set it so that those only stay persisted for an hour.

create_data_manager(*config_name: str*) → mpf.core.data_manager.DataManager
Return a new DataManager for a certain config.

Parameters **config_name** – Name of the config

get_machine_var(*name: str*) → Any
Return the value of a machine variable.

Parameters **name** – String name of the variable you want to get that value for.

Returns The value of the variable if it exists, or None if the variable does not exist.

get_platform_sections(*platform_section: str*, *overwrite: str*) → SmartVirtualHardwarePlatform
Return platform section.

init_done() → Generator[[int, None], None]
Finish init.

Called when init is done and all boot holds are cleared.

initialise() → Generator[[int, None], None]
Initialise machine.

initialise_core_and_hardware() → Generator[[int, None], None]
Load core modules and hardware.

initialise_mpf()
Initialise MPF.

is_machine_var(*name: str*) → bool
Return true if machine variable exists.

register_boot_hold(*hold: str*) → None
Register a boot hold.

register_monitor(*monitor_class: str, monitor: Callable[..., Any]*) → None
Register a monitor.

Parameters

- **monitor_class** – String name of the monitor class for this monitor that's being registered.

- **monitor** – Callback to notify

MPF uses monitors to allow components to monitor certain internal elements of MPF.

For example, a player variable monitor could be setup to be notified of any changes to a player variable, or a switch monitor could be used to allow a plugin to be notified of any changes to any switches.

The MachineController's list of registered monitors doesn't actually do anything. Rather it's a dictionary of sets which the monitors themselves can reference when they need to do something. We just needed a central registry of monitors.

remove_machine_var(*name: str*) → None
Remove a machine variable by name.

If this variable persists to disk, it will remove it from there too.

Parameters **name** – String name of the variable you want to remove.

remove_machine_var_search(*startswith: str = ''*, *endswith: str = ''*) → None
Remove a machine variable by matching parts of its name.

Parameters

- **startswith** – Optional start of the variable name to match.
- **endswith** – Optional end of the variable name to match.

For example, if you pass startswit='player' and endswith='score', this method will match and remove player1_score, player2_score, etc.

reset() → Generator[[int, None], None]
Reset the machine.

This method is safe to call. It essentially sets up everything from scratch without reloading the config files and assets from disk. This method is called after a game ends and before attract mode begins.

run() → None
Start the main machine run loop.

set_default_platform(*name: str*) → None
Set the default platform.

It is used if a device class-specific or device-specific platform is not specified.

Parameters **name** – String name of the platform to set to default.

set_machine_var(*name: str*, *value: Any*) → None
Set the value of a machine variable.

Parameters

- **name** – String name of the variable you're setting the value for.
- **value** – The value you're setting. This can be any Type.

shutdown() → None
Shutdown the machine.

stop(***kwargs*) → None
Perform a graceful exit of MPF.

validate_machine_config_section(*section: str*) → None
Validate a config section.

verify_system_info()
Dump information about the Python installation to the log.

Information includes Python version, Python executable, platform, and core architecture.

self.machine.mode_controller

class mpf.core.mode_controller.**ModeController**(*machine: mpf.core.machine.MachineController*)
Bases: mpf.core.mpf_controller.MpfController

Responsible for loading, unloading, and managing all modes in MPF.

Accessing the mode_controller in code

There is only one instance of the mode_controller in MPF, and it's accessible via `self.machine.mode_controller`.

Methods & Attributes

The mode_controller has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

create_mode_devices()
Create mode devices.

dump()
Dump the current status of the running modes to the log file.

initialise_modes(***kwargs*)
Initialise modes.

is_active(*mode_name*)
Return true if the mode is active.

Parameters **mode_name** – String name of the mode to check.

Returns True if the mode is active, False if it is not.

load_mode_devices()
Load mode devices.

load_modes(***kwargs*)
Load the modes from the modes: section of the machine configuration file.

register_load_method(*load_method*, *config_section_name=None*, *priority=0*, ***kwargs*)
Register a method which is called when the mode is loaded.

Used by core components, plugins, etc. to register themselves with the Mode Controller for anything they need a mode to do when it's registered.

Parameters

- **load_method** – The method that will be called when this mode code loads.
- **config_section_name** – An optional string for the section of the configuration file that will be passed to the load_method when it's called.

- **priority** – Int of the relative priority which allows remote methods to be called in a specific order. Default is 0. Higher values will be called first.
- ****kwargs** – Any additional keyword arguments specified will be passed to the load_method.

Note that these methods will be called once, when the mode code is first initialized during the MPF boot process.

register_start_method(*start_method*, *config_section_name=None*, *priority=0*, ***kwargs*)
Register a method which is called anytime a mode is started.

Used by core components, plugins, etc. to register themselves with the Mode Controller for anything that they a mode to do when it starts.

Parameters

- **start_method** – The method that will be called when this mode code loads.
- **config_section_name** – An optional string for the section of the configuration file that will be passed to the start_method when it's called.
- **priority** – Int of the relative priority which allows remote methods to be called in a specific order. Default is 0. Higher values will be called first.
- ****kwargs** – Any additional keyword arguments specified will be passed to the start_method.

register_stop_method(*callback*, *priority=0*)
Register a method which is called when the mode is stopped.

These are universal, in that they're called every time a mode stops priority is the priority they're called. Has nothing to do with mode priority.

remove_start_method(*start_method*, *config_section_name=None*, *priority=0*, ***kwargs*)
Remove an existing start method.

remove_stop_method(*callback*, *priority=0*)
Remove an existing stop method.

set_mode_state(*mode: mpf.core.mode.Mode*, *active: bool*)
Remember mode state.

self.machine.osc

class mpf.plugins.osc.**Osc**(*machine*)
Bases: object

Plays back switch sequences from a config file, used for testing.

Accessing the osc in code

There is only one instance of the osc in MPF, and it's accessible via `self.machine.osc`.

Methods & Attributes

The osc has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

handle_switch(*switch_name*, *state*)
Handle Switch change from OSC.

self.machine.placeholder_manager

class mpf.core.placeholder_manager.**PlaceholderManager**(*machine*)
Bases: mpf.core.placeholder_manager.BasePlaceholderManager

Manages templates and placeholders for MPF.

Accessing the placeholder_manager in code

There is only one instance of the placeholder_manager in MPF, and it's accessible via `self.machine.placeholder_manager`.

Methods & Attributes

The placeholder_manager has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

get_global_parameters(*name*)
Return global params.

self.machine.platform_controller

class mpf.core.platform_controller.**PlatformController**(*machine: MachineController*)
Bases: mpf.core.mpf_controller.MpfController

Manages all platforms and rules.

Accessing the platform_controller in code

There is only one instance of the platform_controller in MPF, and it's accessible via `self.machine.platform_controller`.

Methods & Attributes

The platform_controller has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

clear_hw_rule(*rule: mpf.core.platform_controller.HardwareRule*)
Clear all rules for switch and this driver.

Parameters **rule** – Hardware rule to clean.

set_pulse_on_hit_and_enable_and_release_and_disable_rule(*enable_switch: mpf.core.platform_controller.SwitchRuleSet disable_switch: mpf.core.platform_controller.SwitchRuleSet driver: mpf.core.platform_controller.DriverRuleSet pulse_setting: mpf.core.platform_controller.PulseRuleSetti = None, hold_settings: mpf.core.platform_controller.HoldRuleSetti = None*) → mpf.core.platform_controller.HardwareRule

Add pulse on hit and enable and release and disable rule to driver.

Pulse and then enable driver. Cancel pulse and enable when switch is released or a disable switch is hit.

Parameters

- **enable_switch** –
- **disable_switch** –
- **driver** –
- **pulse_setting** –
- **hold_settings** –

set_pulse_on_hit_and_enable_and_release_rule(*enable_switch: mpf.core.platform_controller.SwitchRuleSettings, driver: mpf.core.platform_controller.DriverRuleSettings, pulse_setting: mpf.core.platform_controller.PulseRuleSettings = None, hold_settings: mpf.core.platform_controller.HoldRuleSettings = None*) → mpf.core.platform_controller.HardwareRule

Add pulse on hit and enable and relase rule to driver.

Pulse and enable a driver. Cancel pulse and enable if switch is released.

Parameters

- **enable_switch** – Switch which triggers the rule.
- **driver** – Driver to trigger.
- **pulse_setting** –

  class PulseRuleSettings
- **hold_settings** –

  class HoldRuleSettings

set_pulse_on_hit_and_release_rule(*enable_switch: mpf.core.platform_controller.SwitchRuleSettings, driver: mpf.core.platform_controller.DriverRuleSettings, pulse_setting: mpf.core.platform_controller.PulseRuleSettings = None*) → mpf.core.platform_controller.HardwareRule

Add pulse on hit and relase rule to driver.

Pulse a driver but cancel pulse when switch is released.

Parameters

- **enable_switch** – Switch which triggers the rule.
- **driver** –

  class DriverRuleSettings
- **pulse_setting** –

  class PulseRuleSettings

set_pulse_on_hit_rule(*enable_switch: mpf.core.platform_controller.SwitchRuleSettings*, *driver: mpf.core.platform_controller.DriverRuleSettings*, *pulse_setting: mpf.core.platform_controller.PulseRuleSettings = None*) → mpf.core.platform_controller.HardwareRule

Add pulse on hit rule to driver.

Always do the full pulse. Even when the switch is released.

Parameters

- **enable_switch** – Switch which triggers the rule.
- **driver** –

  class DriverRuleSettings
- **pulse_setting** –

  class PulseRuleSettings

self.machine.service

class mpf.core.service_controller.**ServiceController**(*machine*)

Bases: mpf.core.mpf_controller.MpfController

Provides all service information and can perform service tasks.

Accessing the service in code

There is only one instance of the service in MPF, and it's accessible via `self.machine.service`.

Methods & Attributes

The service has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

get_coil_map() → List[mpf.core.service_controller.CoilMap]

Return a map of all coils in the machine.

get_light_map() → List[mpf.core.service_controller.LightMap]

Return a map of all lights in the machine.

get_switch_map()

Return a map of all switches in the machine.

is_in_service() → bool
Return true if in service mode.

start_service()
Start service mode.

stop_service()
Stop service mode.

self.machine.settings

class mpf.core.settings_controller.**SettingsController**(*machine*)
Bases: mpf.core.mpf_controller.MpfController

Manages operator controllable settings.

Accessing the settings in code

There is only one instance of the settings in MPF, and it's accessible via `self.machine.settings`.

Methods & Attributes

The settings has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

add_setting(*setting: mpf.core.settings_controller.SettingEntry*)
Add a setting.

get_setting_machine_var(*setting_name*)
Return machine var name.

get_setting_value(*setting_name*)
Return the current value of a setting.

get_setting_value_label(*setting_name*)
Return label for value.

get_settings() → List[mpf.core.settings_controller.SettingEntry]
Return all available settings.

set_setting_value(*setting_name*, *value*)
Set the value of a setting.

self.machine.show_controller

class mpf.core.show_controller.**ShowController**(*machine*)
Bases: mpf.core.mpf_controller.MpfController

Manages all the shows in a pinball machine.

The ShowController handles priorities, restores, running and stopping shows, etc.

Accessing the show_controller in code

There is only one instance of the show_controller in MPF, and it's accessible via `self.machine.show_controller`.

Methods & Attributes

The show_controller has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

create_show_config(*name*, *priority=0*, *speed=1.0*, *loops=-1*, *sync_ms=None*, *manual_advance=False*, *show_tokens=None*, *events_when_played=None*, *events_when_stopped=None*, *events_when_looped=None*, *events_when_paused=None*, *events_when_resumed=None*, *events_when_advanced=None*, *events_when_stepped_back=None*, *events_when_updated=None*, *events_when_completed=None*)

Create a show config.

get_next_show_id()

Return the next show id.

play_show_with_config(*config*, *mode=None*, *start_time=None*)

Play and return a show from config.

Will add the mode priority if a mode is passed.

register_show(*name*, *settings*)

Register a named show.

replace_or_advance_show(*old_instance*, *config: mpf.assets.show.ShowConfig*, *start_step*, *start_time=None*, *stop_callback=None*)

Replace or advance show.

Compare a given show (may be empty) to a show config and ensure that the new config becomes effective. If the old show runs a config which is equal to the new config nothing will be done. If the old_instance is set to manual_advance and one step behind the target step it will advance the show. Otherwise, the old show is stopped and the new show is stopped in sync.

self.machine.switch_controller

class mpf.core.switch_controller.**SwitchController**(*machine: mpf.core.machine.MachineController*)

Bases: `mpf.core.mpf_controller.MpfController`

Tracks all switches in the machine, receives switch activity, and converts switch changes into events.

Accessing the switch_controller in code

There is only one instance of the switch_controller in MPF, and it's accessible via `self.machine.switch_controller`.

Methods & Attributes

The switch_controller has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

add_monitor(*monitor: Callable[[mpf.core.switch_controller.MonitoredSwitchChange], None]*)
Add a monitor callback which is called on switch changes.

add_switch_handler(*switch_name*, *callback*, *state=1*, *ms=0*, *return_info=False*, *callback_kwargs=None*) → mpf.core.switch_controller.SwitchHandler
Register a handler to take action on a switch event.

Parameters

- **switch_name** – String name of the switch you're adding this handler for.
- **callback** – The method you want called when this switch handler fires.
- **state** – Integer of the state transition you want to callback to be triggered on. Default is 1 which means it's called when the switch goes from inactive to active, but you can also use 0 which means your callback will be called when the switch becomes inactive
- **ms** – Integer. If you specify a 'ms' parameter, the handler won't be called until the witch is in that state for that many milliseconds.
- **return_info** – If True, the switch controller will pass the parameters of the switch handler as arguments to the callback, including switch_name, state, and ms. If False (default), it just calls the callback with no parameters.
- **callback_kwargs** – Additional kwargs that will be passed with the callback.

You can mix & match entries for the same switch here.

add_switch_handler_obj(*switch*, *callback*, *state=1*, *ms=0*, *return_info=False*, *callback_kwargs=None*)
Register a handler to take action on a switch event.

Same as add_switch_handler but you can pass a switch object instead of a name.

static get_active_event_for_switch(*switch_name*)
Return the event name which is posted when switch_name becomes active.

get_next_timed_switch_event()
Return time of the next timed switch event.

is_active(*switch_name*, *ms=None*)
Query whether a switch is active.

Parameters

- **switch_name** – String name of the switch to check.
- **ms** – Milliseconds that the switch has been active. If this is non-zero, then this method will only return True if the switch has been in that state for at least the number of ms specified.

Returns: True if the switch_name has been active for the given number of ms. If ms is not specified, returns True if the switch is in the state regardless of how long it's been in that state.

is_inactive(*switch_name*, *ms=None*)
Query whether a switch is inactive.

Parameters

- **switch_name** – String name of the switch to check.

- **ms** – Milliseconds that the switch has been inactive. If this is non-zero, then this method will only return True if the switch has been in that state for at least the number of ms specified.

Returns: True if the switch_name has been inactive for the given number of ms. If ms is not specified, returns True if the switch is in the state regardless of how long it's been in that state.

is_state(*switch_name*, *state*, *ms=0.0*)
Check if switch is in state.

Query whether a switch is in a given state and (optionally) whether it has been in that state for the specified number of ms.

Parameters

- **switch_name** – String name of the switch to check.
- **state** – Bool of the state to check. True is active and False is inactive.
- **ms** – Milliseconds that the switch has been in that state. If this is non-zero, then this method will only return True if the switch has been in that state for at least the number of ms specified.

Returns: True if the switch_name has been in the state for the given number of ms. If ms is not specified, returns True if the switch is in the state regardless of how long it's been in that state.

log_active_switches(***kwargs*)
Write out entries to the INFO log file of all switches that are currently active.

ms_since_change(*switch_name*)
Return the number of ms that have elapsed since this switch last changed state.

Parameters **switch_name** – String name of the switch to check.

Returns Integer of milliseconds.

process_switch(*name*, *state=1*, *logical=False*)
Process a new switch state change for a switch by name.

This is the method that is called by the platform driver whenever a switch changes state. It's also used by the "other" modules that activate switches, including the keyboard and OSC interfaces.

State 0 means the switch changed from active to inactive, and 1 means it changed from inactive to active. (The hardware & platform code handles NC versus NO switches and translates them to 'active' versus 'inactive'.)

Parameters

- **name** – The string name of the switch.
- **state** – Boolean or int of state of the switch you're processing, True/1 is active, False/0 is inactive.
- **logical** – Boolean which specifies whether the 'state' argument represents the "physical" or "logical" state of the switch. If True, a 1 means this switch is active and a 0 means it's inactive, regardless of the NC/NO configuration of the switch. If False, then the state parameter passed will be inverted if the switch is configured to be an 'NC' type. Typically the hardware will send switch states in their raw (logical=False) states, but other interfaces like the keyboard and OSC will use logical=True.

process_switch_by_num(*num*, *state*, *platform*, *logical=False*)
Process a switch state change by switch number.

Parameters

- **num** – The switch number (based on the platform number) for the switch you're setting.
- **state** – The state to set, either 0 or 1.
- **platform** – The platform this switch is on.
- **logical** – Whether the state you're setting is the logical or physical state of the switch. If a switch is NO (normally open), then the logical and physical states will be the same. NC (normally closed) switches will have physical and logical states that are inverted from each other.

process_switch_obj(*obj: mpf.devices.switch.Switch*, *state*, *logical*)
Process a new switch state change for a switch by name.

Parameters

- **obj** – The switch object.
- **state** – Boolean or int of state of the switch you're processing, True/1 is active, False/0 is inactive.
- **logical** – Boolean which specifies whether the 'state' argument represents the "physical" or "logical" state of the switch. If True, a 1 means this switch is active and a 0 means it's inactive, regardless of the NC/NO configuration of the switch. If False, then the state parameter passed will be inverted if the switch is configured to be an 'NC' type. Typically the hardware will send switch states in their raw (logical=False) states, but other interfaces like the keyboard and OSC will use logical=True.

This is the method that is called by the platform driver whenever a switch changes state. It's also used by the "other" modules that activate switches, including the keyboard and OSC interfaces.

State 0 means the switch changed from active to inactive, and 1 means it changed from inactive to active. (The hardware & platform code handles NC versus NO switches and translates them to 'active' versus 'inactive'.)

register_switch(*switch: mpf.devices.switch.Switch*)
Add the name of a switch to the switch controller for tracking.

Parameters **switch** – Switch object to add

remove_monitor(*monitor: Callable[[mpf.core.switch_controller.MonitoredSwitchChange], None]*)
Remove a monitor callback.

remove_switch_handler(*switch_name*, *callback*, *state=1*, *ms=0*)
Remove a registered switch handler.

Currently this only works if you specify everything exactly as you set it up. (Except for return_info, which doesn't matter if true or false, it will remove either / both.

remove_switch_handler_by_key(*switch_handler: mpf.core.switch_controller.SwitchHandler*)
Remove switch handler by key returned from add_switch_handler.

remove_switch_handler_obj(*switch*, *callback*, *state=1*, *ms=0*)
Remove a registered switch handler.

Same as remove_switch_handler but takes a switch object instead of the name.

set_state(*switch_name*, *state=1*, *reset_time=False*)
Set the state of a switch.

Note that since this method just sets the logical state of the switch, weird things can happen if the state diverges from the physical state of the switch.

It's mainly used with the virtual platforms to set the initial states of switches on MPF boot.

Parameters

- **switch_name** – String name of the switch to set.
- **state** – Logical state to set. 0 is inactive and 1 is active.
- **reset_time** – Sets the timestamp of the change to -100000 which indicates that this switch was in this state when the machine was powered on and therefore the various timed switch handlers will not be triggered.

update_switches_from_hw()
Update the states of all the switches be re-reading the states from the hardware platform.

This method works silently and does not post any events if any switches changed state.

verify_switches() → bool
Verify that switches states match the hardware.

Loops through all the switches and queries their hardware states via their platform interfaces and then compares that to the state that MPF thinks the switches are in.

Throws logging warnings if anything doesn't match.

This method is notification only. It doesn't fix anything.

wait_for_any_switch(*switch_names: List[str], state: int = 1, only_on_change=True, ms=0*)
Wait for the first switch in the list to change into state.

Parameters

- **switch_names** – Iterable of strings of switch names. Whichever switch changes first will trigger this wait.
- **state** – The state to wait for. 0 = inactive, 1 = active, 2 = opposite to current.
- **only_on_change** – Bool which controls whether this wait will be triggered now if the switch is already in the state, or whether it will wait until the switch changes into that state.
- **ms** – How long the switch needs to be in the new state to trigger the wait.

wait_for_switch(*switch_name: str*, *state: int = 1*, *only_on_change=True*, *ms=0*)
Wait for a switch to change into a state.

Parameters

- **switch_name** – String name of the switch to wait for.
- **state** – The state to wait for. 0 = inactive, 1 = active, 2 = opposite to current.
- **only_on_change** – Bool which controls whether this wait will be triggered now if the switch is already in the state, or whether it will wait until the switch changes into that state.
- **ms** – How long the switch needs to be in the new state to trigger the wait.

self.machine.switch_player

class mpf.plugins.switch_player.**SwitchPlayer**(*machine*)
Bases: object

Plays back switch sequences from a config file, used for testing.

Accessing the switch_player in code

There is only one instance of the switch_player in MPF, and it's accessible via `self.machine.switch_player`.

Methods & Attributes

The switch_player has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

self.machine.text_ui

class mpf.core.text_ui.**TextUi**(*machine: MachineController*)
Bases: mpf.core.mpf_controller.MpfController

Handles the text-based UI.

Accessing the text_ui in code

There is only one instance of the text_ui in MPF, and it's accessible via `self.machine.text_ui`.

Methods & Attributes

The text_ui has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

stop(***kwargs*)
Stop the Text UI and restore the original console screen.

8.3.2 Devices

Instances of MPF devices, available at `self.machine.*device_collection*.*device_name*`. For example, a flipper device called "right_flipper" is at `self.machine.flippers.right_flipper`, and a multiball called "awesome" is accessible at `self.machine.multiballs.awesome`.

Note that device collections are accessible as attributes and items, so the right flipper mentioned above is also available to programmers at `self.machine.flippers['right_flipper']`.

Note: "Devices" in MPF are more than physical hardware devices. Many of the "game logic" components listed in the user documentation (achievements, ball holds, extra balls, etc.) are implemented as "devices" in MPF code. (So you can think of devices as being either physical or logical.)

Here's a list of all the device types in MPF, linked to their API references.

self.machine.accelerometers.*

class mpf.devices.accelerometer.**Accelerometer**(**args*, ***kwargs*)
Bases: mpf.core.system_wide_device.SystemWideDevice

Implements a multi-axis accelerometer.

In modern machines, accelerometers can be used for tilt detection and to detect whether a machine is properly leveled.

The accelerometer device produces a data stream of readings which MPF converts to g-forces, and then events can be posted when the "hit" (or g-force) of an accelerometer exceeds a predefined threshold.

Accessing accelerometers in code

The device collection which contains the accelerometers in your machine is available via `self.machine.accelerometers`. For example, to access one called "foo", you would use `self.machine.accelerometers.foo`. You can also access accelerometers in dictionary form, e.g. `self.machine.accelerometers['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Accelerometers have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

get_level_xyz() → float
Return current 3D level.

get_level_xz() → float
Return current 2D x/z level.

get_level_yz() → float
Return current 2D y/z level.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

update_acceleration(*x: float*, *y: float*, *z: float*) → None
Calculate acceleration based on readings from hardware.

self.machine.accruals.*

class mpf.devices.logic_blocks.**Accrual**(*machine*, *name*)
Bases: mpf.devices.logic_blocks.LogicBlock

A type of LogicBlock which tracks many different events (steps) towards a goal.

The steps are able to happen in any order.

Accessing accruals in code

The device collection which contains the accruals in your machine is available via `self.machine.accruals`. For example, to access one called "foo", you would use `self.machine.accruals.foo`. You can also access accruals in dictionary form, e.g. `self.machine.accruals['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Accruals have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

complete()
Mark this logic block as complete.

Posts the 'events_when_complete' events and optionally restarts this logic block or disables it, depending on this block's configuration settings.

completed
Return if completed.

disable(***kwargs*)
Disable this logic block.

Automatically called when one of the disable_event events is posted. Can also manually be called.

enable(***kwargs*)
Enable this logic block.

Automatically called when one of the enable_event events is posted. Can also manually be called.

enabled
Return if enabled.

get_start_value() → List[bool]
Return start states.

hit(*step: int*, ***kwargs*)
Increase the hit progress towards completion.

Automatically called when one of the *count_events* is posted. Can also manually be called.

Parameters **step** – Integer of the step number (0 indexed) that was just hit.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

reset(***kwargs*)
Reset the progress towards completion of this logic block.

Automatically called when one of the reset_event events is called. Can also be manually called.

restart(***kwargs*)
Restart this logic block by calling reset() and enable().

Automatically called when one of the restart_event events is called. Can also be manually called.

value
Return value or None if that is currently not possible.

self.machine.achievement_groups.*

class mpf.devices.achievement_group.**AchievementGroup**(**args*, ***kwargs*)
Bases: mpf.core.mode_device.ModeDevice

An achievement group in a pinball machine.

It is tracked per player and can automatically restore state on the next ball.

Accessing achievement_groups in code

The device collection which contains the achievement_groups in your machine is available via `self.machine.achievement_groups`. For example, to access one called "foo", you would use `self.machine.achievement_groups.foo`. You can also access achievement_groups in dictionary form, e.g. `self.machine.achievement_groups['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Achievement_groups have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

disable(***kwargs*)
Disable achievement group.

enable(***kwargs*)
Enable achievement group.

enabled
Return enabled state.

member_state_changed()
Notify the group that one of its members has changed state.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

rotate_left(***kwargs*)
Rotate to the left.

rotate_right(*reverse=False*, ***kwargs*)
Rotate to the right.

select_random_achievement(***kwargs*)
Select a random achievement.

start_selected(***kwargs*)
Start the currently selected achievement.

self.machine.achievements.*

class mpf.devices.achievement.**Achievement**(**args*, ***kwargs*)
Bases: mpf.core.mode_device.ModeDevice

An achievement in a pinball machine.

It is tracked per player and can automatically restore state on the next ball.

Accessing achievements in code

The device collection which contains the achievements in your machine is available via `self.machine.achievements`. For example, to access one called "foo", you would use `self.machine.achievements.foo`. You can also access achievements in dictionary form, e.g. `self.machine.achievements['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Achievements have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

add_to_group(*group*)
Add this achievement to an achievement group.

Parameters **group** – The achievement group to add this achievement to.

complete(***kwargs*)
Complete achievement.

disable(***kwargs*)
Disable achievement.

enable(***kwargs*)
Enable the achievement.

It can only start if it was enabled before.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

remove_from_group(*group*)
Remove this achievement from an achievement group.

Parameters **group** – The achievement group to remove this achievement from.

reset(***kwargs*)
Reset the achievement to its initial state.

select(***kwargs*)
Highlight (select) this achievement.

start(***kwargs*)
Start achievement.

state
Return current state.

stop(***kwargs*)
Stop achievement.

self.machine.autofires.*

class mpf.devices.autofire.**AutofireCoil**(**args*, ***kwargs*)
Bases: mpf.core.system_wide_device.SystemWideDevice

Autofire coils which fire based on switch hits with a hardware rule.

Coils in the pinball machine which should fire automatically based on switch hits using defined hardware switch rules.

Autofire coils work with rules written to the hardware pinball controller that allow them to respond "instantly" to switch hits versus waiting for the lag of USB and the host computer.

Examples of Autofire Coils are pop bumpers, slingshots, and kicking targets. (Flippers use the same autofire rules under the hood, but flipper devices have their own device type in MPF.

Accessing autofires in code

The device collection which contains the autofires in your machine is available via `self.machine.autofires`. For example, to access one called "foo", you would use `self.machine.autofires.foo`. You can also access autofires in dictionary form, e.g. `self.machine.autofires['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Autofires have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

disable(***kwargs*)
Disable the autofire device.

This is typically called at the end of a ball and when a tilt event happens.

Parameters ****kwargs** – Not used, just included so this method can be used as an event callback.

enable(***kwargs*)
Enable the autofire device.

This causes the coil to respond to the switch hits. This is typically called when a ball starts to enable the slingshots, pops, etc.

Note that there are several options for both the coil and the switch which can be incorporated into this rule, including recycle times, switch debounce, reversing the switch (fire the coil when the switch goes inactive), etc. These rules vary by hardware platform. See the user documentation for the hardware platform for details.

Parameters ****kwargs** – Not used, just included so this method can be used as an event callback.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

self.machine.ball_devices.*

class mpf.devices.ball_device.ball_device.**BallDevice**(**args*, ***kwargs*)
Bases: mpf.core.system_wide_device.SystemWideDevice

Base class for a 'Ball Device' in a pinball machine.

A ball device is anything that can hold one or more balls, such as a trough, an eject hole, a VUK, a catapult, etc.

Args: Same as Device.

Accessing ball_devices in code

The device collection which contains the ball_devices in your machine is available via `self.machine.ball_devices`. For example, to access one called "foo", you would use `self.machine.ball_devices.foo`. You can also access ball_devices in dictionary form, e.g. `self.machine.ball_devices['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Ball_devices have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

add_incoming_ball(*incoming_ball: mpf.devices.ball_device.incoming_balls_handler.IncomingBall*)
Notify this device that there is a ball heading its way.

available_balls
Number of balls that are available to be ejected. This differs from *balls* since it's possible that this device could have balls that are being used for some other eject, and thus not available.

balls
Return the number of balls we expect in the near future.

cancel_path_if_target_is(*start*, *target*)
Check if the ball is going to a certain target and cancel the path in that case.

capacity
Return the ball capacity.

eject(*balls=1*, *target=None*, ***kwargs*) → int
Eject balls to target.

Return the number of balls found for eject. The remaining balls are queued for eject when available.

eject_all(*target=None*, ***kwargs*)
Eject all the balls from this device.

Parameters

- **target** – The string or BallDevice target for this eject. Default of None means *playfield*.
- ****kwargs** – unused

Returns True if there are balls to eject. False if this device is empty.

entrance(***kwargs*)
Event handler for entrance events.

expected_ball_received()
Handle an expected ball.

find_available_ball_in_path(*start*)
Try to remove available ball at the end of the path.

find_next_trough()
Find next trough after device.

find_one_available_ball(*path=deque([])*)
Find a path to a source device which has at least one available ball.

find_path_to_target(*target*)
Find a path to this target.

handle_mechanial_eject_during_idle()
Handle mechanical eject.

hold(***kwargs*)
Event handler for hold event.

classmethod is_playfield()
Return True if this ball device is a Playfield-type device, False if it's a regular ball device.

lost_ejected_ball(*target*)
Handle an outgoing lost ball.

lost_idle_ball()
Lost an ball while the device was idle.

lost_incoming_ball(*source*)
Handle lost ball which was confirmed to have left source.

raise_config_error(*msg*, *error_no*, ***, *context=None*)
Raise a ConfigFileError exception.

remove_incoming_ball(*incoming_ball: mpf.devices.ball_device.incoming_balls_handler.IncomingBall*)
Remove a ball from the incoming balls queue.

request_ball(*balls=1*, ***kwargs*)
Request that one or more balls is added to this device.

Parameters

- **balls** – Integer of the number of balls that should be added to this device. A value of -1 will cause this device to try to fill itself.
- ****kwargs** – unused

requested_balls
Return the number of requested balls.

set_eject_state(*state*)
Set the current device state.

setup_eject_chain(*path*, *player_controlled=False*)
Set up an eject chain.

setup_eject_chain_next_hop(*path*, *player_controlled*)
Set up one hop of the eject chain.

setup_player_controlled_eject(*target=None*)
Set up a player controlled eject.

state
Return the device state.

stop_device()
Stop device.

unexpected_ball_received()
Handle an unexpected ball.

wait_for_ready_to_receive(*source*)
Wait until this device is ready to receive a ball.

self.machine.ball_holds.*

class mpf.devices.ball_hold.**BallHold**(**args*, ***kwargs*)
Bases: mpf.core.system_wide_device.SystemWideDevice, mpf.core.mode_device.ModeDevice

Ball hold device which can be used to keep balls in ball devices and control their eject later on.

Accessing ball_holds in code

The device collection which contains the ball_holds in your machine is available via `self.machine.ball_holds`. For example, to access one called "foo", you would use `self.machine.ball_holds.foo`. You can also access ball_holds in dictionary form, e.g. `self.machine.ball_holds['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Ball_holds have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

disable(***kwargs*)
Disable the hold.

If the hold is not enabled, no balls will be held.

Parameters ****kwargs** – unused

enable(***kwargs*)
Enable the hold.

If the hold is not enabled, no balls will be held.

Parameters ****kwargs** – unused

is_full()
Return true if hold is full.

raise_config_error(*msg*, *error_no*, ***, *context=None*)
Raise a ConfigFileError exception.

release_all(***kwargs*)
Release all balls in hold.

release_balls(*balls_to_release*)
Release all balls and return the actual amount of balls released.

Parameters **balls_to_release** – number of ball to release from hold

release_one(***kwargs*)
Release one ball.

Parameters ****kwargs** – unused

release_one_if_full(***kwargs*)
Release one ball if hold is full.

remaining_space_in_hold()
Return the remaining capacity of the hold.

reset(***kwargs*)
Reset the hold.

Will release held balls. Device status will stay the same (enabled/disabled). It will wait for those balls to drain and block ball_ending until they do. Those balls are not included in ball_in_play.

Parameters ****kwargs** – unused

self.machine.ball_locks.*

class mpf.devices.ball_lock.**BallLock**(**args*, ***kwargs*)
Bases: mpf.core.system_wide_device.SystemWideDevice, mpf.core.mode_device.ModeDevice

Ball lock device which can be used to keep balls in ball devices and control their eject later on.

Accessing ball_locks in code

The device collection which contains the ball_locks in your machine is available via `self.machine.ball_locks`. For example, to access one called "foo", you would use `self.machine.ball_locks.foo`. You can also access ball_locks in dictionary form, e.g. `self.machine.ball_locks['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Ball_locks have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

disable(***kwargs*)
Disable the lock.

If the lock is not enabled, no balls will be locked.

Parameters ****kwargs** – unused

enable(***kwargs*)
Enable the lock.

If the lock is not enabled, no balls will be locked.

Parameters ****kwargs** – unused

is_full()
Return true if lock is full.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

release_all_balls()
Release all balls in lock.

release_balls(*balls_to_release*)
Release all balls and return the actual amount of balls released.

Parameters **balls_to_release** – number of ball to release from lock

release_one(***kwargs*)
Release one ball.

Parameters ****kwargs** – unused

release_one_if_full(***kwargs*)
Release one ball if lock is full.

remaining_space_in_lock()
Return the remaining capacity of the lock.

reset(***kwargs*)
Reset the lock.

Will release locked balls. Device will status will stay the same (enabled/disabled). It will wait for those balls to drain and block ball_ending until they did. Those balls are not included in ball_in_play.

Parameters ****kwargs** – unused

self.machine.ball_saves.*

class mpf.devices.ball_save.**BallSave**(**args*, ***kwargs*)
Bases: mpf.core.system_wide_device.SystemWideDevice, mpf.core.mode_device.ModeDevice

Ball save device which will give back the ball within a certain time.

Accessing ball_saves in code

The device collection which contains the ball_saves in your machine is available via `self.machine.ball_saves`. For example, to access one called "foo", you would use `self.machine.ball_saves.foo`. You can also access ball_saves in dictionary form, e.g. `self.machine.ball_saves['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Ball_saves have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

delayed_eject(***kwargs*)
Trigger eject of all scheduled balls.

disable(***kwargs*) → None
Disable ball save.

early_ball_save(***kwargs*) → None
Perform early ball save if enabled.

enable(***kwargs*) → None
Enable ball save.

raise_config_error(*msg*, *error_no*, ***, *context=None*)
Raise a ConfigFileError exception.

timer_start(***kwargs*) → None
Start the timer.

This is usually called after the ball was ejected while the ball save may have been enabled earlier.

self.machine.coils.*

class mpf.devices.driver.**Driver**(*machine: mpf.core.machine.MachineController*, *name: str*)
Bases: mpf.core.system_wide_device.SystemWideDevice

Generic class that holds driver objects.

A 'driver' is any device controlled from a driver board which is typically the high-voltage stuff like coils and flashers.

This class exposes the methods you should use on these driver types of devices. Each platform module (i.e. P-ROC, FAST, etc.) subclasses this class to actually communicate with the physical hardware and perform the actions.

Args: Same as the Device parent class

Accessing coils in code

The device collection which contains the coils in your machine is available via `self.machine.coils`. For example, to access one called "foo", you would use `self.machine.coils.foo`. You can also access coils in dictionary form, e.g. `self.machine.coils['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Coils have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

disable(***kwargs*)
Disable this driver.

enable(*pulse_ms: int = None*, *pulse_power: float = None*, *hold_power: float = None*, ***kwargs*)
Enable a driver by holding it 'on'.

Parameters

- **pulse_ms** – The number of milliseconds the driver should be enabled for. If no value is provided, the driver will be enabled for the value specified in the config dictionary.
- **pulse_power** – The pulse power. A float between 0.0 and 1.0.
- **hold_power** – The pulse power. A float between 0.0 and 1.0.

If this driver is configured with a holdpatter, then this method will use that holdpatter to pwm pulse the driver.

If not, then this method will just enable the driver. As a safety precaution, if you want to enable() this driver without pwm, then you have to add the following option to this driver in your machine configuration files:

allow_enable: True

get_and_verify_hold_power(*hold_power: Optional[float]*) → float
Return the hold power to use.

If hold_power is None it will use the default_hold_power. Additionally it will verify the limits.

get_and_verify_pulse_ms(*pulse_ms: Optional[int]*) → int
Return and verify pulse_ms to use.

If pulse_ms is None return the default.

get_and_verify_pulse_power(*pulse_power: Optional[float]*) → float
Return the pulse power to use.

If pulse_power is None it will use the default_pulse_power. Additionally it will verify the limits.

pulse(*pulse_ms: int = None*, *pulse_power: float = None*, *max_wait_ms: int = None*, ***kwargs*) → int
Pulse this driver.

Parameters

- **pulse_ms** – The number of milliseconds the driver should be enabled for. If no value is provided, the driver will be enabled for the value specified in the config dictionary.
- **pulse_power** – The pulse power. A float between 0.0 and 1.0.

raise_config_error(*msg*, *error_no*, ***, *context=None*)
Raise a ConfigFileError exception.

self.machine.combo_switches.*

class mpf.devices.combo_switch.**ComboSwitch**(**args*, ***kwargs*)
Bases: mpf.core.system_wide_device.SystemWideDevice, mpf.core.mode_device.ModeDevice

Combo Switch device.

Accessing combo_switches in code

The device collection which contains the combo_switches in your machine is available via `self.machine.combo_switches`. For example, to access one called "foo", you would use `self.machine.combo_switches.foo`. You can also access combo_switches in dictionary form, e.g. `self.machine.combo_switches['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Combo_switches have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

enable(**kwarg) → None
Enable handler.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

state
Return current state.

self.machine.counters.*

class mpf.devices.logic_blocks.**Counter**(*machine: mpf.core.machine.MachineController*, *name: str*)
Bases: mpf.devices.logic_blocks.LogicBlock

A type of LogicBlock that tracks multiple hits of a single event.

This counter can be configured to track hits towards a specific end-goal (like number of tilt hits to tilt), or it can be an open-ended count (like total number of ramp shots).

It can also be configured to count up or to count down, and can have a configurable counting interval.

Accessing counters in code

The device collection which contains the counters in your machine is available via `self.machine.counters`. For example, to access one called "foo", you would use `self.machine.counters.foo`. You can also access counters in dictionary form, e.g. `self.machine.counters['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Counters have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

complete()
Mark this logic block as complete.

Posts the 'events_when_complete' events and optionally restarts this logic block or disables it, depending on this block's configuration settings.

completed
Return if completed.

count(**kwargs)
Increase the hit progress towards completion.

This method is also automatically called when one of the `count_events` is posted.

disable(**kwargs)
Disable this logic block.

Automatically called when one of the disable_event events is posted. Can also manually be called.

enable(**kwargs)
Enable this logic block.

Automatically called when one of the enable_event events is posted. Can also manually be called.

enabled
Return if enabled.

get_start_value() → int
Return start count.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

reset(**kwargs*)
Reset the progress towards completion of this logic block.

Automatically called when one of the reset_event events is called. Can also be manually called.

restart(**kwargs*)
Restart this logic block by calling reset() and enable().

Automatically called when one of the restart_event events is called. Can also be manually called.

stop_ignoring_hits(**kwargs*)
Cause the Counter to stop ignoring subsequent hits that occur within the 'multiple_hit_window'.

Automatically called when the window time expires. Can safely be manually called.

value
Return value or None if that is currently not possible.

self.machine.digital_outputs.*

class mpf.devices.digital_output.**DigitalOutput**(*machine: mpf.core.machine.MachineController*, *name: str*)
Bases: mpf.core.system_wide_device.SystemWideDevice

A digital output on either a light or driver platform.

Accessing digital_outputs in code

The device collection which contains the digital_outputs in your machine is available via `self.machine.digital_outputs`. For example, to access one called "foo", you would use `self.machine.digital_outputs.foo`. You can also access digital_outputs in dictionary form, e.g. `self.machine.digital_outputs['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Digital_outputs have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

disable(**kwargs*)
Disable digital output.

enable(**kwargs*)
Enable digital output.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

self.machine.diverters.*

class mpf.devices.diverter.**Diverter**(**args*, ***kwargs*)
Bases: mpf.core.system_wide_device.SystemWideDevice

Represents a diverter in a pinball machine.

Args: Same as the Device parent class.

Accessing diverters in code

The device collection which contains the diverters in your machine is available via `self.machine.diverters`. For example, to access one called "foo", you would use `self.machine.diverters.foo`. You can also access diverters in dictionary form, e.g. `self.machine.diverters['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Diverters have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

activate(***kwargs*)
Physically activate this diverter's coil.

deactivate(***kwargs*)
Deactivate this diverter.

This method will disable the activation_coil, and (optionally) if it's configured with a deactivation coil, it will pulse it.

disable(*auto=False*, ***kwargs*)
Disable this diverter.

This method will remove the hardware rule if this diverter is activated via a hardware switch.

Parameters

- **auto** – Boolean value which is used to indicate whether this diverter disabled itself automatically. This is passed to the event which is posted.
- ****kwargs** – This is here because this disable method is called by whatever event the game programmer specifies in their machine configuration file, so we don't know what event that might be or whether it has random kwargs attached to it.

enable(*auto=False*, ***kwargs*)
Enable this diverter.

Parameters

- **auto** – Boolean value which is used to indicate whether this diverter enabled itself automatically. This is passed to the event which is posted.
- ****kwargs** – unused

If an 'activation_switches' is configured, then this method writes a hardware autofire rule to the pinball controller which fires the diverter coil when the switch is activated.

If no *activation_switches* is specified, then the diverter is activated immediately.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

reset(**kwargs*)
Reset and deactivate the diverter.

schedule_deactivation()
Schedule a delay to deactivate this diverter.

self.machine.dmds.*

class mpf.devices.dmd.**Dmd**(*machine*, *name*)
Bases: mpf.core.system_wide_device.SystemWideDevice

A physical DMD.

Accessing dmds in code

The device collection which contains the dmds in your machine is available via `self.machine.dmds`. For example, to access one called "foo", you would use `self.machine.dmds.foo`. You can also access dmds in dictionary form, e.g. `self.machine.dmds['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Dmds have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

update(*data: bytes*)
Update data on the dmd.

Parameters **data** – bytes to send

self.machine.drop_target_banks.*

class mpf.devices.drop_target.**DropTargetBank**(**args*, ***kwargs*)
Bases: mpf.core.system_wide_device.SystemWideDevice, mpf.core.mode_device.ModeDevice

A bank of drop targets in a pinball machine by grouping together multiple *DropTarget* class devices.

Accessing drop_target_banks in code

The device collection which contains the drop_target_banks in your machine is available via `self.machine.drop_target_banks`. For example, to access one called "foo", you would use `self.machine.drop_target_banks.foo`. You can also access drop_target_banks in dictionary form, e.g. `self.machine.drop_target_banks['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Drop_target_banks have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

enable(***kwarg*) → None
Enable handler.

member_target_change()
Handle that a member drop target has changed state.

This method causes this group to update its down and up counts and complete status.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

reset(***kwargs*)
Reset this bank of drop targets.

This method has some intelligence to figure out what coil(s) it should fire. It builds up a set by looking at its own reset_coil and reset_coils settings, and also scanning through all the member drop targets and collecting their coils. Then it pulses each of them. (This coil list is a "set" which means it only sends a single pulse to each coil, even if each drop target is configured with its own coil.)

self.machine.drop_targets.*

class mpf.devices.drop_target.**DropTarget**(**args*, ***kwargs*)
Bases: mpf.core.system_wide_device.SystemWideDevice

Represents a single drop target in a pinball machine.

Args: Same as the *Target* parent class

Accessing drop_targets in code

The device collection which contains the drop_targets in your machine is available via `self.machine.drop_targets`. For example, to access one called "foo", you would use `self.machine.drop_targets.foo`. You can also access drop_targets in dictionary form, e.g. `self.machine.drop_targets['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Drop_targets have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

add_to_bank(*bank*)
Add this drop target to a drop target bank.

This allows the bank to update its status based on state changes to this drop target.

Parameters **bank** – DropTargetBank object to add this drop target to.

disable_keep_up(***kwargs*)
No longer keep up the target up.

enable_keep_up(***kwargs*)
Keep the target up by enabling the coil.

knockdown(***kwargs*)
Pulse the knockdown coil to knock down this drop target.

raise_config_error(*msg*, *error_no*, ***, *context=None*)
Raise a ConfigFileError exception.

remove_from_bank(*bank*)
Remove the DropTarget from a bank.

Parameters **bank** – DropTargetBank object to remove

reset(***kwargs*)
Reset this drop target.

If this drop target is configured with a reset coil, then this method will pulse that coil. If not, then it checks to see if this drop target is part of a drop target bank, and if so, it calls the reset() method of the drop target bank.

This method does not reset the target profile, however, the switch event handler should reset the target profile on its own when the drop target physically moves back to the up position.

self.machine.dual_wound_coils.*

class mpf.devices.dual_wound_coil.**DualWoundCoil**(*machine*, *name*)
Bases: mpf.core.system_wide_device.SystemWideDevice

An instance of a dual wound coil which consists of two coils.

Accessing dual_wound_coils in code

The device collection which contains the dual_wound_coils in your machine is available via `self.machine.dual_wound_coils`. For example, to access one called "foo", you would use `self.machine.dual_wound_coils.foo`. You can also access dual_wound_coils in dictionary form, e.g. `self.machine.dual_wound_coils['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Dual_wound_coils have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

disable(***kwargs*)
Disable a driver.

enable(***kwargs*)
Enable a dual wound coil.

Pulse main coil and enable hold coil.

pulse(*milliseconds: int = None*, *power: float = None*, ***kwargs*)
Pulse this driver.

Parameters

- **milliseconds** – The number of milliseconds the driver should be enabled for. If no value is provided, the driver will be enabled for the value specified in the config dictionary.
- **power** – A multiplier that will be applied to the default pulse time, typically a float between 0.0 and 1.0. (Note this is can only be used if milliseconds is also specified.)

raise_config_error(*msg*, *error_no*, ***, *context=None*)
Raise a ConfigFileError exception.

self.machine.extra_ball_groups.*

class mpf.devices.extra_ball_group.**ExtraBallGroup**(**args*, ***kwargs*)
Bases: mpf.core.system_wide_device.SystemWideDevice

Tracks and manages groups of extra balls devices.

Accessing extra_ball_groups in code

The device collection which contains the extra_ball_groups in your machine is available via `self.machine.extra_ball_groups`. For example, to access one called "foo", you would use `self.machine.extra_ball_groups.foo`. You can also access extra_ball_groups in dictionary form, e.g. `self.machine.extra_ball_groups['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Extra_ball_groups have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

award(*posted_unlit_events=False*, ***kwargs*)
Immediately awards an extra ball.

This event first checks to make sure the limits of the max extra balls have not been exceeded and that this group is enabled.

Note that this method will work even if this group does not have any extra balls or extra balls lit. You can use this to directly award an extra ball.

award_disabled()
Post the events when an extra ball connect be awarded.

award_lit(***kwargs*)
Award a lit extra ball.

If the player does not have any lit extra balls, this method does nothing.

enabled
Return whether this extra ball group is enabled.

This attribute considers the enabled setting plus the max balls per game and ball settings.

is_ok_to_light()
Check if it's possible to light an extra ball.

Returns True of False

This method checks to see if the group is enabled and whether the max_lit setting has been exceeded.

light(***kwargs*)
Light the extra ball for possible collection by the player.

This method checks that the group is enabled and that the max lit value has not been exceeded. If so, this method will post the extra ball disabled events.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

self.machine.extra_balls.*

class mpf.devices.extra_ball.**ExtraBall**(**args*, ***kwargs*)
Bases: mpf.core.mode_device.ModeDevice

An extra ball which can be awarded once per player.

Accessing extra_balls in code

The device collection which contains the extra_balls in your machine is available via `self.machine.extra_balls`. For example, to access one called "foo", you would use `self.machine.extra_balls.foo`. You can also access extra_balls in dictionary form, e.g. `self.machine.extra_balls['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Extra_balls have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

award(***kwargs*)
Award extra ball to player (if enabled).

enable(***kwarg*) → None
Enable handler.

enabled
Return whether this extra ball group is enabled.

This takes into consideration the enabled setting plus the max balls per game setting.

group = None
The ExtraBallGroup this ExtraBall belongs to, or None.

is_ok_to_award()
Check whether this extra ball can be awarded.

This method takes into consideration whether this extra ball is enabled, whether the `max_per_game` has been exceeded, and, if this extra ball is a member of a group, whether the group is enabled and will allow an additional extra ball to be awarded.

Returns True or False

is_ok_to_light()
Check whether this extra ball can be lit.

This method takes into consideration whether this extra ball is enabled, and, if this extra ball is a member of a group, whether the group is enabled and will allow an additional extra ball to lit.

Returns True or False

light(***kwargs*)
Light an extra ball for potential collection by the player.

Lighting an extra ball will immediately increase count against the `max_per_game` setting, even if the extra ball is a member of a group that's disabled or if the player never actually collects the extra ball.

Note that this only really does anything if this extra ball is a member of a group.

player = None
The current player

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

self.machine.flippers.*

class mpf.devices.flipper.**Flipper**(**args*, ***kwargs*)
Bases: `mpf.core.system_wide_device.SystemWideDevice`

Represents a flipper in a pinball machine. Subclass of Device.

Contains several methods for actions that can be performed on this flipper, like *enable()*, *disable()*, etc.

Flippers have several options, including player buttons, EOS swtiches, multiple coil options (pulsing, hold coils, etc.)

Parameters

- **machine** – A reference to the machine controller instance.
- **name** – A string of the name you'll refer to this flipper object as.

Accessing flippers in code

The device collection which contains the flippers in your machine is available via `self.machine.flippers`. For example, to access one called "foo", you would use `self.machine.flippers.foo`. You can also access flippers in dictionary form, e.g. `self.machine.flippers['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Flippers have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

disable(***kwargs*)
Disable the flipper.

This method makes it so the cabinet flipper buttons no longer control the flippers. Used when no game is active and when the player has tilted.

enable(***kwargs*)
Enable the flipper by writing the necessary hardware rules to the hardware controller.

The hardware rules for coils can be kind of complex given all the options, so we've mapped all the options out here. We literally have methods to enable the various rules based on the rule letters here, which we've implemented below. Keeps it easy to understand. :)

Note there's a platform feature saved at: self.machine.config['platform']['hw_enable_auto_disable']. If True, it means that the platform hardware rules will automatically disable a coil that has been enabled when the trigger switch is disabled. If False, it means the hardware platform needs its own rule to disable the coil when the switch is disabled. Methods F and G below check for that feature setting and will not be applied to the hardware if it's True.

Two coils, using EOS switch to indicate the end of the power stroke: Rule Type Coil Switch Action A. Enable Main Button active D. Enable Hold Button active E. Disable Main EOS active

One coil, using EOS switch (not implemented): Rule Type Coil Switch Action A. Enable Main Button active H. PWM Main EOS active

Two coils, not using EOS switch: Rule Type Coil Switch Action B. Pulse Main Button active D. Enable Hold Button active

One coil, not using EOS switch: Rule Type Coil Switch Action C. Pulse/PWM Main button active

Use EOS switch for safety (for platforms that support mutiple switch rules). Note that this rule is the letter "i", not a numeral 1. I. Enable power if button is active and EOS is not active

raise_config_error(*msg*, *error_no*, ***, *context=None*)
Raise a ConfigFileError exception.

sw_flip(***kwargs*)
Activate the flipper via software as if the flipper button was pushed.

This is needed because the real flipper activations are handled in hardware, so if you want to flip the flippers with the keyboard or OSC interfaces, you have to call this method.

Note this method will keep this flipper enabled until you call sw_release().

sw_release(***kwargs*)
Deactive the flipper via software as if the flipper button was released.

See the documentation for sw_flip() for details.

self.machine.hardware_sound_systems.*

class mpf.devices.hardware_sound_system.**HardwareSoundSystem**(*machine*, *name*)
Bases: mpf.core.system_wide_device.SystemWideDevice

Hardware sound system using in EM and SS machines.

Accessing hardware_sound_systems in code

The device collection which contains the hardware_sound_systems in your machine is available via `self.machine.hardware_sound_systems`. For example, to access one called "foo", you would use `self.machine.hardware_sound_systems.foo`. You can also access hardware_sound_systems in dictionary form, e.g. `self.machine.hardware_sound_systems['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Hardware_sound_systems have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

decrease_volume(*volume: float*)
Increase volume.

increase_volume(*volume: float*)
Increase volume.

play(*sound_number: int*)
Play a sound.

play_file(*file: str*, *platform_options*)
Play a sound file.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

set_volume(*volume: float*)
Set volume.

stop_all_sounds()
Stop all sounds.

text_to_speech(*text: str*, *platform_options*)
Text to speech output.

self.machine.kickbacks.*

class mpf.devices.kickback.**Kickback**(**args*, ***kwargs*)
Bases: *mpf.devices.autofire.AutofireCoil*

A kickback device which will fire a ball back into the playfield.

Accessing kickbacks in code

The device collection which contains the kickbacks in your machine is available via `self.machine.kickbacks`. For example, to access one called "foo", you would use `self.machine.kickbacks.foo`. You can also access kickbacks in dictionary form, e.g. `self.machine.kickbacks['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Kickbacks have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

disable(***kwargs*)
Disable the autofire device.

This is typically called at the end of a ball and when a tilt event happens.

Parameters ****kwargs** – Not used, just included so this method can be used as an event callback.

enable(***kwargs*)
Enable the autofire device.

This causes the coil to respond to the switch hits. This is typically called when a ball starts to enable the slingshots, pops, etc.

Note that there are several options for both the coil and the switch which can be incorporated into this rule, including recycle times, switch debounce, reversing the switch (fire the coil when the switch goes inactive), etc. These rules vary by hardware platform. See the user documentation for the hardware platform for details.

Parameters ****kwargs** – Not used, just included so this method can be used as an event callback.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

self.machine.light_rings.*

class mpf.devices.light_group.**LightRing**(*machine: mpf.core.machine.MachineController*, *name*)
Bases: mpf.devices.light_group.LightGroup

A light ring.

Accessing light_rings in code

The device collection which contains the light_rings in your machine is available via `self.machine.light_rings`. For example, to access one called "foo", you would use `self.machine.light_rings.foo`. You can also access light_rings in dictionary form, e.g. `self.machine.light_rings['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Light_rings have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

color(*color*, *fade_ms=None*, *priority=0*, *key=None*)
Call color on all lights in this group.

get_token()
Return all lights in group as token.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

self.machine.light_stripes.*

class mpf.devices.light_group.**LightStrip**(*machine: mpf.core.machine.MachineController*, *name*)
Bases: mpf.devices.light_group.LightGroup

A light stripe.

Accessing light_stripes in code

The device collection which contains the light_stripes in your machine is available via `self.machine.light_stripes`. For example, to access one called "foo", you would use `self.machine.light_stripes.foo`. You can also access light_stripes in dictionary form, e.g. `self.machine.light_stripes['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Light_stripes have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

color(*color*, *fade_ms=None*, *priority=0*, *key=None*)
Call color on all lights in this group.

get_token()
Return all lights in group as token.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

self.machine.lights.*

class mpf.devices.light.**Light**(**args*, ***kwargs*)
Bases: mpf.core.system_wide_device.SystemWideDevice, mpf.devices.device_mixins.DevicePositionMixin

A light in a pinball machine.

Accessing lights in code

The device collection which contains the lights in your machine is available via `self.machine.lights`. For example, to access one called "foo", you would use `self.machine.lights.foo`. You can also access lights in dictionary form, e.g. `self.machine.lights['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Lights have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

clear_stack()
Remove all entries from the stack and resets this light to 'off'.

color(*color*, *fade_ms=None*, *priority=0*, *key=None*)
Add or update a color entry in this light's stack.

Calling this methods is how you tell this light what color you want it to be.

Parameters

- **color** – RGBColor() instance, or a string color name, hex value, or 3-integer list/tuple of colors.
- **fade_ms** – Int of the number of ms you want this light to fade to the color in. A value of 0 means it's instant. A value of None (the default) means that it will use this light's and/or the machine's default fade_ms setting.
- **priority** – Int value of the priority of these incoming settings. If this light has current settings in the stack at a higher priority, the settings you're adding here won't take effect. However they're still added to the stack, so if the higher priority settings are removed, then the next-highest apply.
- **key** – An arbitrary identifier (can be any immutable object) that's used to identify these settings for later removal. If any settings in the stack already have this key, those settings will be replaced with these new settings.

color_correct(*color*)
Apply the current color correction profile to the color passed.

Parameters **color** – The RGBColor() instance you want to get color corrected.

Returns An updated RGBColor() instance with the current color correction profile applied.

Note that if there is no current color correction profile applied, the returned color will be the same as the color that was passed.

fade_in_progress
Return true if a fade is in progress.

gamma_correct(*color*)
Apply max brightness correction to color.

Parameters **color** – The RGBColor() instance you want to have gamma applied.

Returns An updated RGBColor() instance with gamma corrected.

get_color()
Return an RGBColor() instance of the 'color' setting of the highest color setting in the stack.

This is usually the same color as the physical light, but not always (since physical lights are updated once per frame, this value could vary.

Also note the color returned is the "raw" color that does has not had the color correction profile applied.

get_color_below(*priority*, *key*)
Return an RGBColor() instance of the 'color' setting of the highest color below a certain key.

Similar to get_color.

get_hw_numbers()
Return a list of all hardware driver numbers.

off(*fade_ms=None*, *priority=0*, *key=None*, ***kwargs*)
Turn light off.

Parameters

- **key** – key for removal later on
- **priority** – priority on stack
- **fade_ms** – duration of fade

on(*brightness=None*, *fade_ms=None*, *priority=0*, *key=None*, ***kwargs*)
Turn light on.

Parameters

- **key** – key for removal later on
- **priority** – priority on stack
- **fade_ms** – duration of fade

raise_config_error(*msg*, *error_no*, ***, *context=None*)
Raise a ConfigFileError exception.

remove_from_stack_by_key(*key*, *fade_ms=None*)
Remove a group of color settings from the stack.

Parameters **key** – The key of the settings to remove (based on the 'key' parameter that was originally passed to the color() method.)

This method triggers a light update, so if the highest priority settings were removed, the light will be updated with whatever's below it. If no settings remain after these are removed, the light will turn off.

stack
A list of dicts which represents different commands that have come in to set this light to a certain color (and/or fade). Each entry in the list contains the following key/value pairs:

priority: The relative priority of this color command. Higher numbers take precedent, and the highest priority entry will be the command that's currently active. In the event of a tie, whichever entry was added last wins (based on 'start_time' below).

start_time: The clock time when this command was added. Primarily used to calculate fades, but also used as a tie-breaker for multiple entries with the same priority.

start_color: RGBColor() of the color of this light when this command came in.

dest_time: Clock time that represents when a fade (from start_color to dest_color) will be done. If this is 0, that means there is no fade. When a fade is complete, this value is reset to 0.

dest_color: RGBColor() of the destination this light is fading to. If a command comes in with no fade, then this will be the same as the 'color' below.

key: An arbitrary unique identifier to keep multiple entries in the stack separate. If a new color command comes in with a key that already exists for an entry in the stack, that entry will be replaced by the new entry. The key is also used to remove entries from the stack (e.g. when shows or modes end and they want to remove their commands from the light).

x

Get the X value from the config.

Returns the devices x position from config

y

Get the Y value from the config.

Returns the devices y position from config

z

Get the Z value from the config.

Returns the devices z position from config

self.machine.magnets.*

class mpf.devices.magnet.**Magnet**(**args*, ***kwargs*)

Bases: mpf.core.system_wide_device.SystemWideDevice

Controls a playfield magnet in a pinball machine.

Accessing magnets in code

The device collection which contains the magnets in your machine is available via `self.machine.magnets`. For example, to access one called "foo", you would use `self.machine.magnets.foo`. You can also access magnets in dictionary form, e.g. `self.machine.magnets['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Magnets have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

disable(***kwargs*)
Disable magnet.

enable(***kwargs*)
Enable magnet.

fling_ball(***kwargs*)
Fling the grabbed ball.

grab_ball(***kwargs*)
Grab a ball.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

release_ball(***kwargs*)
Release the grabbed ball.

reset(***kwargs*)
Release ball and disable magnet.

self.machine.motors.*

class mpf.devices.motor.**Motor**(*machine*, *name*)
Bases: mpf.core.system_wide_device.SystemWideDevice

A motor which can be controlled using drivers.

Accessing motors in code

The device collection which contains the motors in your machine is available via `self.machine.motors`. For example, to access one called "foo", you would use `self.machine.motors.foo`. You can also access motors in dictionary form, e.g. `self.machine.motors['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Motors have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

go_to_position(*position*, ***kwargs*)
Move motor to a specific position.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

reset(***kwargs*)
Go to reset position.

self.machine.multiball_locks.*

class mpf.devices.multiball_lock.**MultiballLock**(**args*, ***kwargs*)
Bases: mpf.core.mode_device.ModeDevice

Ball lock device which locks balls for a multiball.

Accessing multiball_locks in code

The device collection which contains the multiball_locks in your machine is available via `self.machine.multiball_locks`. For example, to access one called "foo", you would use `self.machine.multiball_locks.foo`. You can also access multiball_locks in dictionary form, e.g. `self.machine.multiball_locks['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Multiball_locks have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

disable(***kwargs*)
Disable the lock.

If the lock is not enabled, no balls will be locked.

Parameters ****kwargs** – unused

enable(***kwargs*)
Enable the lock.

If the lock is not enabled, no balls will be locked.

Parameters ****kwargs** – unused

is_virtually_full
Return true if lock is full.

locked_balls
Return the number of locked balls for the current player.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

remaining_virtual_space_in_lock
Return the remaining capacity of the lock.

reset_all_counts(***kwargs*)
Reset the locked balls for all players.

reset_count_for_current_player(***kwargs*)
Reset the locked balls for the current player.

self.machine.multiballs.*

class mpf.devices.multiball.**Multiball**(**args*, ***kwargs*)
Bases: mpf.core.system_wide_device.SystemWideDevice, mpf.core.mode_device.ModeDevice

Multiball device for MPF.

Accessing multiballs in code

The device collection which contains the multiballs in your machine is available via `self.machine.multiballs`. For example, to access one called "foo", you would use `self.machine.multiballs.foo`. You can also access multiballs in dictionary form, e.g. `self.machine.multiballs['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Multiballs have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

add_a_ball(***kwargs*)
Add a ball if multiball has started.

disable(***kwargs*)
Disable the multiball.

If the multiball is not enabled, it cannot start. Will not stop a running multiball.

Parameters ****kwargs** – unused

enable(***kwargs*)
Enable the multiball.

If the multiball is not enabled, it cannot start.

Parameters ****kwargs** – unused

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

reset(***kwargs*)
Reset the multiball and disable it.

Parameters ****kwargs** – unused

start(***kwargs*)
Start multiball.

start_or_add_a_ball(***kwargs*)
Start multiball or add a ball if multiball has started.

stop(***kwargs*)
Stop shoot again.

self.machine.playfield_transfers.*

class mpf.devices.playfield_transfer.**PlayfieldTransfer**(*machine*, *name*)
Bases: mpf.core.system_wide_device.SystemWideDevice

Device which move a ball from one playfield to another.

Accessing playfield_transfers in code

The device collection which contains the playfield_transfers in your machine is available via `self.machine.playfield_transfers`. For example, to access one called "foo", you would use `self.machine.playfield_transfers.foo`. You can also access playfield_transfers in dictionary form, e.g. `self.machine.playfield_transfers['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Playfield_transfers have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

transfer(***kwargs*)
Transfer a ball to the target playfield.

self.machine.playfields.*

class mpf.devices.playfield.**Playfield**(**args*, ***kwargs*)
Bases: mpf.core.system_wide_device.SystemWideDevice

One playfield in a pinball machine.

Accessing playfields in code

The device collection which contains the playfields in your machine is available via `self.machine.playfields`. For example, to access one called "foo", you would use `self.machine.playfields.foo`. You can also access playfields in dictionary form, e.g. `self.machine.playfields['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Playfields have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

add_ball(*balls=1*, *source_device=None*, *player_controlled=False*)
Add live ball(s) to the playfield.

Parameters

- **balls** – Integer of the number of balls you'd like to add.
- **source_device** – Optional ball device object you'd like to add the ball(s) from.
- **player_controlled** – Boolean which specifies whether this event is player controlled. (See not below for details)

Returns True if it's able to process the add_ball() request, False if it cannot.

The source_device arg is included to give you an options for specifying the source of the ball(s) to be added. This argument is optional, so if you don't supply them then MPF will use the default_source_device of this playfield.

This method does *not* increase the game controller's count of the number of balls in play. So if you want to add balls (like in a multiball scenario), you need to call this method along with `self.machine.game.add_balls_in_play()`.)

MPF tracks the number of balls in play separately from the actual balls on the playfield because there are numerous situations where the two counts are not the same. For example, if a ball is in a VUK while some animation is playing, there are no balls on the playfield but still one ball in play, or if the player has a two-ball multiball and they shoot them both into locks, there are still two balls in play even though there are no balls on the playfield. The opposite can also be true, like when the player tilts then there are still balls on the playfield but no balls in play.

Explanation of the player_controlled parameter:

Set player_controlled to True to indicate that MPF should wait for the player to eject the ball from the source_device rather than firing a coil. The logic works like this:

If the source_device does not have an eject_coil defined, then it's assumed that player_controlled is the only option. (e.g. this is a traditional plunger.) If the source_device does have an eject_coil defined, then there are two ways the eject could work. (1) there could be a "launch" button of some kind that's used to fire the eject coil, or (2) the device could be the auto/manual combo style where there's a mechanical plunger but also a coil which can eject the ball.

If player_controlled is true and the device has an eject_coil, MPF will look for the player_controlled_eject_tag and eject the ball when a switch with that tag is activated.

If there is no player_controlled_eject_tag, MPF assumes it's a manual plunger and will wait for the ball to disappear from the device based on the device's ball count decreasing.

add_incoming_ball(*incoming_ball: mpf.devices.ball_device.incoming_balls_handler.IncomingBall*)
Track an incoming ball.

add_missing_balls(*balls*)
Notify the playfield that it probably received a ball which went missing elsewhere.

ball_arrived()
Confirm first ball in queue.

ball_search
An instance of *mpf.core.ball_search.BallSearch* which handles ball search for this playfield.

ball_search_block(***kwargs*)
Block ball search for this playfield.

Blocking will disable ball search if it's enabled or running, and will prevent ball search from enabling if it's disabled until ball_search_resume() is called.

ball_search_disable(***kwargs*)
Disable ball search for this playfield.

If the ball search timer is running, it will stop and disable it. If an actual ball search process is running, it will stop.

ball_search_enable(***kwargs*)
Enable ball search for this playfield.

Note this does not start the ball search process, rather, it starts the timer running.

ball_search_unblock(***kwargs*)
Unblock ball search for this playfield.

This will check to see if there are balls on the playfield, and if so, enable ball search.

balls
Return the number of balls on the playfield.

delay
An instance of *mpf.core.delays.DelayManager* which handles delays for this playfield.

expected_ball_received()
Handle an expected ball.

classmethod get_additional_ball_capacity()
Return the number of ball which can be added.

Used to find out how many more balls this device can hold. Since this is the playfield device, this method always returns 999.

Returns: 999

classmethod is_playfield()
Return true since it is a playfield.

mark_playfield_active_from_device_action()
Mark playfield active because a device on the playfield detected activity.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

remove_incoming_ball(*incoming_ball: mpf.devices.ball_device.incoming_balls_handler.IncomingBall*)
Stop tracking an incoming ball.

unexpected_ball_received()
Handle an unexpected ball.

static wait_for_ready_to_receive(*source*)
Playfield is always ready to receive.

self.machine.psus.*

class mpf.devices.power_supply_unit.**PowerSupplyUnit**(*machine*, *name*)
Bases: mpf.core.system_wide_device.SystemWideDevice

Represents a power supply in a pinball machine.

Accessing psus in code

The device collection which contains the psus in your machine is available via `self.machine.psus`. For example, to access one called "foo", you would use `self.machine.psus.foo`. You can also access psus in dictionary form, e.g. `self.machine.psus['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Psus have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

get_wait_time_for_pulse(*pulse_ms*, *max_wait_ms*) → int
Return a wait time for a pulse or 0.

notify_about_instant_pulse(*pulse_ms*)
Notify PSU about pulse.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

self.machine.rgb_dmds.*

class mpf.devices.rgb_dmd.**RgbDmd**(*machine*, *name*)
Bases: mpf.core.system_wide_device.SystemWideDevice

A physical DMD.

Accessing rgb_dmds in code

The device collection which contains the rgb_dmds in your machine is available via `self.machine.rgb_dmds`. For example, to access one called "foo", you would use `self.machine.rgb_dmds.foo`. You can also access rgb_dmds in dictionary form, e.g. `self.machine.rgb_dmds['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Rgb_dmds have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

update(*data: bytes*)
Update data on the dmd.

Parameters **data** – bytes to send

self.machine.score_reel_groups.*

class mpf.devices.score_reel_group.**ScoreReelGroup**(*machine*, *name*)
Bases: mpf.core.system_wide_device.SystemWideDevice

Represents a logical grouping of score reels in a pinball machine.

Multiple individual ScoreReel object make up the individual digits of this group. This group also has support for the blank zero "inserts" that some machines use. This is a subclass of mpf.core.device.Device.

Accessing score_reel_groups in code

The device collection which contains the score_reel_groups in your machine is available via `self.machine.score_reel_groups`. For example, to access one called "foo", you would use `self.machine.score_reel_groups.foo`. You can also access score_reel_groups in dictionary form, e.g. `self.machine.score_reel_groups['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Score_reel_groups have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

classmethod chime(*chime*, **kwargs*)
Pulse chime.

int_to_reel_list(*value*)
Convert an integer to a list of integers that represent each positional digit in this ScoreReelGroup.

The list returned is in reverse order. (See the example below.)

The list returned is customized for this ScoreReelGroup both in terms of number of elements and values of *None* used to represent blank plastic zero inserts that are not controlled by a score reel unit.

For example, if you have a 5-digit score reel group that has 4 phyiscial reels in the tens through ten-thousands position and a fake plastic "0" insert for the ones position, if you pass this method a value of *12300*, it will return *[None, 0, 3, 2, 1]*

This method will pad shorter ints with zeros, and it will chop off leading digits for ints that are too long. (For example, if you pass a value of 10000 to a ScoreReelGroup which only has 4 digits, the returns list would correspond to 0000, since your score reel unit has rolled over.)

Parameters **value** – The interger value you'd like to convert.

Returns A list containing the values for each corresponding score reel, with the lowest reel digit position in list position 0.

light(***kwargs*)
Light up this ScoreReelGroup based on the 'light_tag' in its config.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

set_value(*value*)
Reset the score reel group to display the value passed.

This method will "jump" the score reel group to display the value that's passed as an it. (Note this "jump" technique means it will just move the reels as fast as it can, and nonsensical values might show up on the reel while the movement is in progress.)

This method is used to "reset" a reel group to all zeros at the beginning of a game, and can also be used to reset a reel group that is confused or to switch a reel to the new player's score if multiple players a sharing the same reel group.

Note you can choose to pass either an integer representation of the value, or a value list.

Parameters **value** – An integer value of what the new displayed value (i.e. score) should be. This is the default option if you only pass a single positional argument, e.g. *set_value(2100)*.

unlight(***kwargs*)
Turn off the lights for this ScoreReelGroup based on the 'light_tag' in its config.

wait_for_ready()
Return a future which will be done when all reels reached their destination.

self.machine.score_reels.*

class mpf.devices.score_reel.**ScoreReel**(*machine*, *name*)
Bases: mpf.core.system_wide_device.SystemWideDevice

Represents an individual electro-mechanical score reel in a pinball machine.

Multiples reels of this class can be grouped together into ScoreReelGroups which collectively make up a display like "Player 1 Score" or "Player 2 card value", etc.

This device class is used for all types of mechanical number reels in a machine, including reels that have more than ten numbers and that can move in multiple directions (such as the credit reel).

Accessing score_reels in code

The device collection which contains the score_reels in your machine is available via `self.machine.score_reels`. For example, to access one called "foo", you would use `self.machine.score_reels.foo`. You can also access score_reels in dictionary form, e.g. `self.machine.score_reels['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Score_reels have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

check_hw_switches()
Check all the value switches for this score reel.

This check only happens if *self.ready* is *True*. If the reel is not ready, it means another advance request has come in after the initial one. In that case then the subsequent advance will call this method again when after that advance is done.

If this method finds an active switch, it sets *self.physical_value* to that. Otherwise it sets it to -999. It will also update *self.assumed_value* if it finds an active switch. Otherwise it leaves that value unchanged.

This method is automatically called (via a delay) after the reel advances. The delay is based on the config value *self.config['hw_confirm_time']*.

TODO: What happens if there are multiple active switches? Currently it will return the highest one. Is that ok?

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

set_destination_value(*value*)
Return the integer value of the destination this reel is moving to.

Args:

Returns: The value of the destination. If the current *self.assumed_value* is -999, this method will always return -999 since it doesn't know where the reel is and therefore doesn't know what the destination value would be.

stop(***kwargs*)
Stop device.

wait_for_ready()
Return a future for ready.

self.machine.segment_displays.*

class mpf.devices.segment_display.**SegmentDisplay**(**args*, ***kwargs*)
Bases: mpf.core.system_wide_device.SystemWideDevice

A physical segment display in a pinball machine.

Accessing segment_displays in code

The device collection which contains the segment_displays in your machine is available via `self.machine.segment_displays`. For example, to access one called "foo", you would use `self.machine.segment_displays.foo`. You can also access segment_displays in dictionary form, e.g. `self.machine.segment_displays['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Segment_displays have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

add_text(*text: str*, *priority: int = 0*, *key: str = None*) → None
Add text to display stack.

This will replace texts with the same key.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

remove_text_by_key(*key: str*)
Remove entry from text stack.

set_flashing(*flashing: bool*)
Enable/Disable flashing.

self.machine.sequence_shots.*

class mpf.devices.sequence_shot.**SequenceShot**(*machine*, *name*)
Bases: mpf.core.system_wide_device.SystemWideDevice, mpf.core.mode_device.ModeDevice

A device which represents a sequence shot.

Accessing sequence_shots in code

The device collection which contains the sequence_shots in your machine is available via `self.machine.sequence_shots`. For example, to access one called "foo", you would use `self.machine.sequence_shots.foo`. You can also access sequence_shots in dictionary form, e.g. `self.machine.sequence_shots['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Sequence_shots have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

cancel(**kwargs*)
Reset all sequences.

enable(**kwarg*) → None
Enable handler.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

self.machine.sequences.*

class mpf.devices.logic_blocks.**Sequence**(*machine: mpf.core.machine.MachineController*, *name: str*)

Bases: mpf.devices.logic_blocks.LogicBlock

A type of LogicBlock which tracks many different events (steps) towards a goal.

The steps have to happen in order.

Accessing sequences in code

The device collection which contains the sequences in your machine is available via `self.machine.sequences`. For example, to access one called "foo", you would use `self.machine.sequences.foo`. You can also access sequences in dictionary form, e.g. `self.machine.sequences['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Sequences have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

complete()
Mark this logic block as complete.

Posts the 'events_when_complete' events and optionally restarts this logic block or disables it, depending on this block's configuration settings.

completed
Return if completed.

disable(***kwargs*)
Disable this logic block.

Automatically called when one of the disable_event events is posted. Can also manually be called.

enable(***kwargs*)
Enable this logic block.

Automatically called when one of the enable_event events is posted. Can also manually be called.

enabled
Return if enabled.

get_start_value() → int
Return start step.

hit(*step: int = None*, ***kwargs*)
Increase the hit progress towards completion.

Automatically called when one of the *count_events* is posted. Can also manually be called.

raise_config_error(*msg*, *error_no*, ***, *context=None*)
Raise a ConfigFileError exception.

reset(***kwargs*)
Reset the progress towards completion of this logic block.

Automatically called when one of the reset_event events is called. Can also be manually called.

restart(***kwargs*)
Restart this logic block by calling reset() and enable().

Automatically called when one of the restart_event events is called. Can also be manually called.

value
Return value or None if that is currently not possible.

self.machine.servos.*

class mpf.devices.servo.**Servo**(**args*, ***kwargs*)
Bases: mpf.core.system_wide_device.SystemWideDevice

Represents a servo in a pinball machine.

Args: Same as the Device parent class.

Accessing servos in code

The device collection which contains the servos in your machine is available via `self.machine.servos`. For example, to access one called "foo", you would use `self.machine.servos.foo`. You can also access servos in dictionary form, e.g. `self.machine.servos['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Servos have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

go_to_position(*position*)
Move servo to position.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

reset(***kwargs*)
Go to reset position.

set_acceleration_limit(*acceleration_limit*)
Set acceleration parameter.

set_speed_limit(*speed_limit*)
Set speed parameter.

self.machine.shot_groups.*

class mpf.devices.shot_group.**ShotGroup**(*machine*, *name*)
Bases: mpf.core.mode_device.ModeDevice

Represents a group of shots in a pinball machine by grouping together multiple *Shot* class devices.

This is used so you get get "group-level" functionality, like shot rotation, shot group completion, etc. This would be used for a group of rollover lanes, a bank of standups, etc.

Accessing shot_groups in code

The device collection which contains the shot_groups in your machine is available via `self.machine.shot_groups`. For example, to access one called "foo", you would use `self.machine.shot_groups.foo`. You can also access shot_groups in dictionary form, e.g. `self.machine.shot_groups['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Shot_groups have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

disable(***kwargs*)
Disable all member shots.

Parameters **kwargs** – passed to member shots

disable_rotation(***kwargs*)
Disable shot rotation.

If disabled, rotation events do not actually rotate the shots.

enable(***kwargs*)
Enable all member shots.

Parameters **kwargs** – passed to member shots

enable_rotation(***kwargs*)
Enable shot rotation.

If disabled, rotation events do not actually rotate the shots.

raise_config_error(*msg*, *error_no*, ***, *context=None*)
Raise a ConfigFileError exception.

reset(***kwargs*)
Reset all member shots.

Parameters **kwargs** – passed to member shots

restart(***kwargs*)
Restart all member shots.

rotate(*direction=None*, ***kwargs*)
Rotate (or "shift") the state of all the shots in this group.

This is used for things like lane change, where hitting the flipper button shifts all the states of the shots in the group to the left or right.

This method actually transfers the current state of each shot profile to the left or the right, and the shot on the end rolls over to the taret on the other end.

Parameters

- **direction** – String that specifies whether the rotation direction is to the left or right. Values are 'right' or 'left'. Default of None will cause the shot group to rotate in the direction as specified by the rotation_pattern.
- **states** – A string of a state or a list of strings that represent the targets that will be selected to rotate. If None (default), then all targets will be included.

- **exclude_states** – A string of a state or a list of strings that controls whether any targets will *not* be rotated. (Any targets with an active profile in one of these states will not be included in the rotation. Default is None which means all targets will be rotated)
- **kwargs** – unused

Note that this shot group must, and rotation_events for this shot group, must both be enabled for the rotation events to work.

rotate_left(*mode=None*, ***kwargs*)
Rotate the state of the shots to the left.

This method is the same as calling rotate('left')

Parameters **kwargs** – unused

rotate_right(*mode=None*, ***kwargs*)
Rotate the state of the shots to the right.

This method is the same as calling rotate('right')

Parameters **kwargs** – unused

self.machine.shot_profiles.*

class mpf.devices.shot_profile.**ShotProfile**(*machine: mpf.core.machine.MachineController*, *name: str*)
Bases: mpf.core.mode_device.ModeDevice, mpf.core.system_wide_device.SystemWideDevice

A shot profile.

Accessing shot_profiles in code

The device collection which contains the shot_profiles in your machine is available via `self.machine.shot_profiles`. For example, to access one called "foo", you would use `self.machine.shot_profiles.foo`. You can also access shot_profiles in dictionary form, e.g. `self.machine.shot_profiles['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Shot_profiles have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

enable(***kwarg*) → None
Enable handler.

raise_config_error(*msg*, *error_no*, ***, *context=None*)
Raise a ConfigFileError exception.

self.machine.shots.*

class mpf.devices.shot.**Shot**(*machine*, *name*)
Bases: mpf.core.enable_disable_mixin.EnableDisableMixin, mpf.core.mode_device.ModeDevice

A device which represents a generic shot.

Accessing shots in code

The device collection which contains the shots in your machine is available via `self.machine.shots`. For example, to access one called "foo", you would use `self.machine.shots.foo`. You can also access shots in dictionary form, e.g. `self.machine.shots['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Shots have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

active_sequences = None
(id, current_position_index, next_switch)

Type List of tuples

advance(*force=False*, ***kwargs*) → bool
Advance a shot profile forward.

If this profile is at the last step and configured to loop, it will roll over to the first step. If this profile is at the last step and not configured to loop, this method has no effect.

disable(***kwargs*)
Disable device.

enable(***kwargs*) → None
Enable device.

enabled
Return true if enabled.

hit(***kwargs*)
Advance the currently-active shot profile.

Note that the shot must be enabled in order for this hit to be processed.

jump(*state*, *force=True*)
Jump to a certain state in the active shot profile.

Parameters

- **state** – int of the state number you want to jump to. Note that states are zero-based, so the first state is 0.
- **show_step** – The step number that the associated light script should start playing at. Useful with rotations so this shot can pick up right where it left off. Default is 1 (the first step in the show)

monitor_enabled = False
Class attribute which specifies whether any monitors have been registered to track shots.

persist_enabled
Return if enabled is persisted.

profile
Return profile.

profile_name
Return profile name.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

reset(***kwargs*)
Reset the shot profile for the passed mode back to the first state (State 0) and reset all sequences.

restart(***kwargs*)
Restart the shot profile by calling reset() and enable().

Automatically called when one fo the restart_events is called.

state
Return current state index.

state_name
Return current state name.

self.machine.state_machines.*

class mpf.devices.state_machine.**StateMachine**(*machine*, *name*)
Bases: mpf.core.system_wide_device.SystemWideDevice, mpf.core.mode_device.ModeDevice

A generic state machine.

Accessing state_machines in code

The device collection which contains the state_machines in your machine is available via `self.machine.state_machines`. For example, to access one called "foo", you would use `self.machine.state_machines.foo`. You can also access state_machines in dictionary form, e.g. `self.machine.state_machines['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

State_machines have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

enable(***kwarg*) → None
Enable handler.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

state
Return the current state.

self.machine.steppers.*

class mpf.devices.stepper.**Stepper**(**args*, ***kwargs*)
Bases: mpf.core.system_wide_device.SystemWideDevice

Represents an stepper motor based axis in a pinball machine.

Args: Same as the Device parent class.

Accessing steppers in code

The device collection which contains the steppers in your machine is available via `self.machine.steppers`. For example, to access one called "foo", you would use `self.machine.steppers.foo`. You can also access steppers in dictionary form, e.g. `self.machine.steppers['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Steppers have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

reset(***kwargs*)
Move to reset position.

stop()
Stop motor.

self.machine.switches.*

class mpf.devices.switch.**Switch**(**args*, ***kwargs*)
Bases: mpf.core.system_wide_device.SystemWideDevice, mpf.devices.device_mixins.DevicePositionMixin

A switch in a pinball machine.

Accessing switches in code

The device collection which contains the switches in your machine is available via `self.machine.switches`. For example, to access one called "foo", you would use `self.machine.switches.foo`. You can also access switches in dictionary form, e.g. `self.machine.switches['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Switches have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

add_handler(*callback*, *state=1*, *ms=0*, *return_info=False*, *callback_kwargs=None*)
Add switch handler (callback) for this switch which is called when this switch state changes.

Note that this method just calls the *Switch Controller's* `add_switch_handler()` method behind the scenes.

Parameters

- **callback** – A callable method that will be called when the switch state changes.
- **state** – The state that the switch which change into which triggers the callback to be called. Values are 0 or 1, with 0 meaning the switch changed to inactive, and 1 meaning the switch changed to an active state.
- **ms** – How many milliseconds the switch needs to be in the new state before the callback is called. Default is 0 which means that the callback will be called immediately. You can use this setting as a form of software debounce, as the switch needs to be in the state consistently before the callback is called.
- **return_info** – If True, the switch controller will pass the parameters of the switch handler as arguments to the callback, including switch_name, state, and ms.
- **callback_kwargs** – Additional kwargs that will be passed with the callback.

hw_state
The physical hardware state of the switch. 1 = active, 0 = inactive. This is what the actual hardware is reporting and does not consider whether a switch is NC or NO.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

remove_handler(*callback*, *state=1*, *ms=0*)
Remove switch handler for this switch.

state
The logical state of a switch. 1 = active, 0 = inactive. This takes into consideration the NC or NO settings for the switch.

x
Get the X value from the config.

Returns the devices x position from config

y
Get the Y value from the config.

Returns the devices y position from config

z
Get the Z value from the config.

Returns the devices z position from config

self.machine.timed_switches.*

class mpf.devices.timed_switch.**TimedSwitch**(**args*, ***kwargs*)
Bases: mpf.core.system_wide_device.SystemWideDevice, mpf.core.mode_device.ModeDevice

Timed Switch device.

Accessing timed_switches in code

The device collection which contains the timed_switches in your machine is available via `self.machine.timed_switches`. For example, to access one called "foo", you would use `self.machine.timed_switches.foo`. You can also access timed_switches in dictionary form, e.g. `self.machine.timed_switches['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Timed_switches have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

enable(***kwarg*) → None
Enable handler.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

self.machine.timers.*

class mpf.devices.timer.**Timer**(**args*, ***kwargs*)
Bases: mpf.core.mode_device.ModeDevice

Parent class for a mode timer.

Parameters

- **machine** – The main MPF MachineController object.
- **name** – The string name of this timer.

Accessing timers in code

The device collection which contains the timers in your machine is available via `self.machine.timers`. For example, to access one called "foo", you would use `self.machine.timers.foo`. You can also access timers in dictionary form, e.g. `self.machine.timers['foo']`.

You can also get devices by tag or hardware number. See the DeviceCollection documentation for details.

Methods & Attributes

Timers have the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

add(*timer_value*, ***kwargs*)
Add ticks to this timer.

Parameters

- **timer_value** – The number of ticks you want to add to this timer's current value.
- **kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

change_tick_interval(*change=0.0*, ***kwargs*)
Change the interval for each "tick" of this timer.

Parameters

- **change** – Float or int of the change you want to make to this timer's tick rate. Note this value is added to the current tick interval. To set an absolute value, use the set_tick_interval() method. To shorten the tick rate, use a negative value.
- ****kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

enable(***kwarg*) → None
Enable handler.

jump(*timer_value*, ***kwargs*)
Set the current amount of time of this timer.

This value is expressed in "ticks" since the interval per tick can be something other than 1 second).

Parameters

- **timer_value** – Integer of the current value you want this timer to be.
- ****kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

pause(*timer_value=0*, ***kwargs*)
Pause the timer and posts the 'timer_<name>_paused' event.

Parameters

- **timer_value** – How many seconds you want to pause the timer for. Note that this pause time is real-world seconds and does not take into consideration this timer's tick interval.
- ****kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

raise_config_error(*msg*, *error_no*, ***, *context=None*)
Raise a ConfigFileError exception.

reset(***kwargs*)
Reset this timer based to the starting value that's already been configured.

Does not start or stop the timer.

Parameters ****kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

restart(***kwargs*)
Restart the timer by resetting it and then starting it.

Essentially this is just a reset() then a start().

Parameters ****kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

set_tick_interval(*timer_value*, ***kwargs*)
Set the number of seconds between ticks for this timer.

This is an absolute setting. To apply a change to the current value, use the change_tick_interval() method.

Parameters

- **timer_value** – The new number of seconds between each tick of this timer. This value should always be positive.
- ****kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

start(***kwargs*)
Start this timer based on the starting value that's already been configured.

Use jump() if you want to set the starting time value.

Parameters ****kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

stop(***kwargs*)
Stop the timer and posts the 'timer_<name>_stopped' event.

Parameters ****kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

subtract(*timer_value*, ***kwargs*)
Subtract ticks from this timer.

Parameters

- **timer_value** – The number of ticks you want to subtract from this timer's current value.
- ****kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

ticks
Return ticks.

timer_complete(***kwargs*)
Automatically called when this timer completes.

Posts the 'timer_<name>_complete' event. Can be manually called to mark this timer as complete.

Parameters ****kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

8.3.3 Modes

Covers all the "built-in" modes. They're accessible via `self.machine.modes.*name*`, for example, `self.machine.modes.game` or `self.machine.modes.base`.

self.machine.modes.attract

class mpf.modes.attract.code.attract.**Attract**(*machine*, *config*, *name*, *path*)
Bases: *mpf.core.mode.Mode*

Default mode running in a machine when a game is not in progress.

The attract mode's main job is to watch for the start button to be pressed, to post the requests to start games, and to move the machine flow to the next mode if the request to start game comes back as approved.

Accessing the attract mode via code

You can access the attract mode from anywhere via `self.machine.modes.attract`.

Methods & Attributes

The attract mode has the following methods & attributes available. Note that methods & attributes inherited from the base Mode class are not included here.

active
Return *True* if this mode is active.

add_mode_event_handler(*event: str*, *handler: Callable*, *priority: int = 0*, ***kwargs*)
Register an event handler which is automatically removed when this mode stops.

This method is similar to the Event Manager's add_handler() method, except this method automatically unregisters the handlers when the mode ends.

Parameters

- **event** – String name of the event you're adding a handler for. Since events are text strings, they don't have to be pre-defined.
- **handler** – The method that will be called when the event is fired.
- **priority** – An arbitrary integer value that defines what order the handlers will be called in. The default is 1, so if you have a handler that you want to be called first, add it here with a priority of 2. (Or 3 or 10 or 100000.) The numbers don't matter. They're called from highest to lowest. (i.e. priority 100 is called before priority 1.)
- ****kwargs** – Any any additional keyword/argument pairs entered here will be attached to the handler and called whenever that handler is called. Note these are in addition to kwargs that could be passed as part of the event post. If there's a conflict, the event-level ones will win.

Returns A GUID reference to the handler which you can use to later remove the handler via `remove_handler_by_key`. Though you don't need to remove the handler since the whole point of this method is they're automatically removed when the mode stops.

Note that if you do add a handler via this method and then remove it manually, that's ok too.

configure_logging(*logger: str*, *console_level: str = 'basic'*, *file_level: str = 'basic'*)
Configure logging.

Parameters

- **logger** – The string name of the logger to use.

- **console_level** – The level of logging for the console. Valid options are "none", "basic", or "full".
- **file_level** – The level of logging for the console. Valid options are "none", "basic", or "full".

create_mode_devices() → None
Create new devices that are specified in a mode config that haven't been created in the machine-wide.

debug_log(*msg: str*, **args*, ***kwargs*) → None
Log a message at the debug level.

Note that whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

error_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the error level.

These messages will always be shown in the console and the log file.

ignorable_runtime_exception(*msg: str*) → None
Handle ignorable runtime exception.

During development or tests raise an exception for easier debugging. Log an error during production.

info_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the info level.

Whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

initialise_mode() → None
Initialise this mode.

is_game_mode
Return true if this is a game mode.

load_mode_devices() → None
Load config of mode devices.

mode_will_start(***kwargs*) → None
User-overrideable method which will be called whenever this mode starts (i.e. before it becomes active).

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

result_of_start_request(*ev_result=True*)
Handle the result of the start request.

Called after the *request_to_start_game* event is posted.

If *result* is True, this method posts the event *game_start*. If False, nothing happens, as the game start request was denied by some handler.

Parameters **ev_result** – Bool result of the boolean event *request_to_start_game*. If any registered event handler did not want the game to start, this will be False. Otherwise it's True.

start(*mode_priority=None*, *callback=None*, ***kwargs*) → None
Start this mode.

Parameters

- **mode_priority** – Integer value of what you want this mode to run at. If you don't specify one, it will use the "Mode: priority" setting from this mode's configuration file.

- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode starts in the mode_start method which will be called automatically.

start_button_pressed()
Handle start button press.

Called when the a switch tagged with *start* is activated.

Note that in MPF, the game start process is initiated when the start button is *released*, so when the button is first pressed, MPF just records the time stamp. This allows the total time the start button was pressed to be note, so that, for example, different types of games can be started based on long-presses of the start button.

start_button_released(***kwargs*)
Handle start button release.

Called when the a switch tagged with *start* is deactivated.

Since this is the Attract mode, this method posts a boolean event called *request_to_start_game*. If that event comes back True, this method calls *result_of_start_request()*.

stop(*callback: Any = None*, ***kwargs*) → bool
Stop this mode.

Parameters

- **callback** – Method which will be called once this mode has stopped. Will only be called when the mode is running (includes currently stopping)
- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode stops in the mode_stop method which will be called automatically.

Returns true if the mode is running. Otherwise false.

warning_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the warning level.

These messages will always be shown in the console and the log file.

self.machine.modes.bonus

class mpf.modes.bonus.code.bonus.**Bonus**(*machine*, *config*, *name*, *path*)
Bases: *mpf.core.mode.Mode*

Bonus mode for MPF.

Give a player bonus for their achievements, but only if the machine is not tilted.

Accessing the bonus mode via code

You can access the bonus mode from anywhere via `self.machine.modes.bonus`.

Methods & Attributes

The bonus mode has the following methods & attributes available. Note that methods & attributes inherited from the base Mode class are not included here.

active
Return *True* if this mode is active.

add_mode_event_handler(*event: str*, *handler: Callable*, *priority: int = 0*, ***kwargs*)
Register an event handler which is automatically removed when this mode stops.

This method is similar to the Event Manager's add_handler() method, except this method automatically unregisters the handlers when the mode ends.

Parameters

- **event** – String name of the event you're adding a handler for. Since events are text strings, they don't have to be pre-defined.
- **handler** – The method that will be called when the event is fired.
- **priority** – An arbitrary integer value that defines what order the handlers will be called in. The default is 1, so if you have a handler that you want to be called first, add it here with a priority of 2. (Or 3 or 10 or 100000.) The numbers don't matter. They're called from highest to lowest. (i.e. priority 100 is called before priority 1.)
- ****kwargs** – Any any additional keyword/argument pairs entered here will be attached to the handler and called whenever that handler is called. Note these are in addition to kwargs that could be passed as part of the event post. If there's a conflict, the event-level ones will win.

Returns A GUID reference to the handler which you can use to later remove the handler via `remove_handler_by_key`. Though you don't need to remove the handler since the whole point of this method is they're automatically removed when the mode stops.

Note that if you do add a handler via this method and then remove it manually, that's ok too.

configure_logging(*logger: str*, *console_level: str = 'basic'*, *file_level: str = 'basic'*)
Configure logging.

Parameters

- **logger** – The string name of the logger to use.
- **console_level** – The level of logging for the console. Valid options are "none", "basic", or "full".
- **file_level** – The level of logging for the console. Valid options are "none", "basic", or "full".

create_mode_devices() → None
Create new devices that are specified in a mode config that haven't been created in the machine-wide.

debug_log(*msg: str*, **args*, ***kwargs*) → None
Log a message at the debug level.

Note that whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

error_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the error level.

These messages will always be shown in the console and the log file.

hurry_up(***kwargs*)
Change the slide display delay to the "hurry up" setting.

This is typically used with a flipper cancel event to hurry up the bonus display when the player hits both flippers.

ignorable_runtime_exception(*msg: str*) → None
Handle ignorable runtime exception.

During development or tests raise an exception for easier debugging. Log an error during production.

info_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the info level.

Whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

initialise_mode() → None
Initialise this mode.

is_game_mode
Return true if this is a game mode.

load_mode_devices() → None
Load config of mode devices.

mode_will_start(***kwargs*) → None
User-overrideable method which will be called whenever this mode starts (i.e. before it becomes active).

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

start(*mode_priority=None*, *callback=None*, ***kwargs*) → None
Start this mode.

Parameters

- **mode_priority** – Integer value of what you want this mode to run at. If you don't specify one, it will use the "Mode: priority" setting from this mode's configuration file.
- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode starts in the mode_start method which will be called automatically.

stop(*callback: Any = None*, ***kwargs*) → bool
Stop this mode.

Parameters

- **callback** – Method which will be called once this mode has stopped. Will only be called when the mode is running (includes currently stopping)
- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode stops in the mode_stop method which will be called automatically.

Returns true if the mode is running. Otherwise false.

warning_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the warning level.

These messages will always be shown in the console and the log file.

self.machine.modes.carousel

class mpf.modes.carousel.code.carousel.**Carousel**(*machine*, *config*, *name*, *path*)
Bases: *mpf.core.mode.Mode*

Mode which allows the player to select another mode to run.

Accessing the carousel mode via code

You can access the carousel mode from anywhere via `self.machine.modes.carousel`.

Methods & Attributes

The carousel mode has the following methods & attributes available. Note that methods & attributes inherited from the base Mode class are not included here.

active
Return *True* if this mode is active.

add_mode_event_handler(*event: str*, *handler: Callable*, *priority: int = 0*, ***kwargs*)
Register an event handler which is automatically removed when this mode stops.

This method is similar to the Event Manager's add_handler() method, except this method automatically unregisters the handlers when the mode ends.

Parameters

- **event** – String name of the event you're adding a handler for. Since events are text strings, they don't have to be pre-defined.
- **handler** – The method that will be called when the event is fired.
- **priority** – An arbitrary integer value that defines what order the handlers will be called in. The default is 1, so if you have a handler that you want to be called first, add it here with a priority of 2. (Or 3 or 10 or 100000.) The numbers don't matter. They're called from highest to lowest. (i.e. priority 100 is called before priority 1.)
- ****kwargs** – Any any additional keyword/argument pairs entered here will be attached to the handler and called whenever that handler is called. Note these are in addition to kwargs that could be passed as part of the event post. If there's a conflict, the event-level ones will win.

Returns A GUID reference to the handler which you can use to later remove the handler via `remove_handler_by_key`. Though you don't need to remove the handler since the whole point of this method is they're automatically removed when the mode stops.

Note that if you do add a handler via this method and then remove it manually, that's ok too.

configure_logging(*logger: str*, *console_level: str = 'basic'*, *file_level: str = 'basic'*)
Configure logging.

Parameters

- **logger** – The string name of the logger to use.
- **console_level** – The level of logging for the console. Valid options are "none", "basic", or "full".
- **file_level** – The level of logging for the console. Valid options are "none", "basic", or "full".

create_mode_devices() → None
Create new devices that are specified in a mode config that haven't been created in the machine-wide.

debug_log(*msg: str*, **args*, ***kwargs*) → None
Log a message at the debug level.

Note that whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

error_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the error level.

These messages will always be shown in the console and the log file.

ignorable_runtime_exception(*msg: str*) → None
Handle ignorable runtime exception.

During development or tests raise an exception for easier debugging. Log an error during production.

info_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the info level.

Whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

initialise_mode() → None
Initialise this mode.

is_game_mode
Return true if this is a game mode.

load_mode_devices() → None
Load config of mode devices.

mode_will_start(***kwargs*) → None
User-overrideable method which will be called whenever this mode starts (i.e. before it becomes active).

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

start(*mode_priority=None*, *callback=None*, ***kwargs*) → None
Start this mode.

Parameters

- **mode_priority** – Integer value of what you want this mode to run at. If you don't specify one, it will use the "Mode: priority" setting from this mode's configuration file.
- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode starts in the mode_start method which will be called automatically.

stop(*callback: Any = None*, ***kwargs*) → bool
Stop this mode.

Parameters

- **callback** – Method which will be called once this mode has stopped. Will only be called when the mode is running (includes currently stopping)
- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode stops in the mode_stop method which will be called automatically.

Returns true if the mode is running. Otherwise false.

warning_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the warning level.

These messages will always be shown in the console and the log file.

self.machine.modes.credits

class mpf.modes.credits.code.credits.**Credits**(*machine*, *config*, *name*, *path*)
Bases: *mpf.core.mode.Mode*

Mode which manages the credits and prevents the game from starting without credits.

Accessing the credits mode via code

You can access the credits mode from anywhere via `self.machine.modes.credits`.

Methods & Attributes

The credits mode has the following methods & attributes available. Note that methods & attributes inherited from the base Mode class are not included here.

active
Return *True* if this mode is active.

add_credit(*price_tiering=True*)
Add a single credit to the machine.

Parameters **price_tiering** – Boolean which controls whether this credit will be eligible for the pricing tier bonuses. Default is True.

add_mode_event_handler(*event: str*, *handler: Callable*, *priority: int = 0*, ***kwargs*)
Register an event handler which is automatically removed when this mode stops.

This method is similar to the Event Manager's add_handler() method, except this method automatically unregisters the handlers when the mode ends.

Parameters

- **event** – String name of the event you're adding a handler for. Since events are text strings, they don't have to be pre-defined.
- **handler** – The method that will be called when the event is fired.

- **priority** – An arbitrary integer value that defines what order the handlers will be called in. The default is 1, so if you have a handler that you want to be called first, add it here with a priority of 2. (Or 3 or 10 or 100000.) The numbers don't matter. They're called from highest to lowest. (i.e. priority 100 is called before priority 1.)
- ****kwargs** – Any any additional keyword/argument pairs entered here will be attached to the handler and called whenever that handler is called. Note these are in addition to kwargs that could be passed as part of the event post. If there's a conflict, the event-level ones will win.

Returns A GUID reference to the handler which you can use to later remove the handler via `remove_handler_by_key`. Though you don't need to remove the handler since the whole point of this method is they're automatically removed when the mode stops.

Note that if you do add a handler via this method and then remove it manually, that's ok too.

clear_all_credits(***kwargs*)
Clear all credits.

configure_logging(*logger: str*, *console_level: str = 'basic'*, *file_level: str = 'basic'*)
Configure logging.

Parameters

- **logger** – The string name of the logger to use.
- **console_level** – The level of logging for the console. Valid options are "none", "basic", or "full".
- **file_level** – The level of logging for the console. Valid options are "none", "basic", or "full".

create_mode_devices() → None
Create new devices that are specified in a mode config that haven't been created in the machine-wide.

debug_log(*msg: str*, **args*, ***kwargs*) → None
Log a message at the debug level.

Note that whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

enable_credit_play(*post_event=True*, ***kwargs*)
Enable credits play.

enable_free_play(*post_event=True*, ***kwargs*)
Enable free play.

error_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the error level.

These messages will always be shown in the console and the log file.

ignorable_runtime_exception(*msg: str*) → None
Handle ignorable runtime exception.

During development or tests raise an exception for easier debugging. Log an error during production.

info_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the info level.

Whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

initialise_mode() → None
Initialise this mode.

is_game_mode
Return true if this is a game mode.

load_mode_devices() → None
Load config of mode devices.

mode_will_start(***kwargs*) → None
User-overrideable method which will be called whenever this mode starts (i.e. before it becomes active).

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

start(*mode_priority=None*, *callback=None*, ***kwargs*) → None
Start this mode.

Parameters

- **mode_priority** – Integer value of what you want this mode to run at. If you don't specify one, it will use the "Mode: priority" setting from this mode's configuration file.
- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode starts in the mode_start method which will be called automatically.

stop(*callback: Any = None*, ***kwargs*) → bool
Stop this mode.

Parameters

- **callback** – Method which will be called once this mode has stopped. Will only be called when the mode is running (includes currently stopping)
- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode stops in the mode_stop method which will be called automatically.

Returns true if the mode is running. Otherwise false.

toggle_credit_play(***kwargs*)
Toggle between free and credits play.

warning_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the warning level.

These messages will always be shown in the console and the log file.

self.machine.modes.game

class mpf.modes.game.code.game.**Game**(*machine*, *config*, *name*, *path*)
Bases: mpf.core.async_mode.AsyncMode

Base mode that runs an active game on a pinball machine.

The game mode is responsible for creating players, starting and ending balls, rotating to the next player, etc.

Accessing the game mode via code

You can access the game mode from anywhere via `self.machine.modes.game`.

Methods & Attributes

The game mode has the following methods & attributes available. Note that methods & attributes inherited from the base Mode class are not included here.

active
Return *True* if this mode is active.

add_mode_event_handler(*event: str*, *handler: Callable*, *priority: int = 0*, ***kwargs*)
Register an event handler which is automatically removed when this mode stops.

This method is similar to the Event Manager's add_handler() method, except this method automatically unregisters the handlers when the mode ends.

Parameters

- **event** – String name of the event you're adding a handler for. Since events are text strings, they don't have to be pre-defined.
- **handler** – The method that will be called when the event is fired.
- **priority** – An arbitrary integer value that defines what order the handlers will be called in. The default is 1, so if you have a handler that you want to be called first, add it here with a priority of 2. (Or 3 or 10 or 100000.) The numbers don't matter. They're called from highest to lowest. (i.e. priority 100 is called before priority 1.)
- ****kwargs** – Any any additional keyword/argument pairs entered here will be attached to the handler and called whenever that handler is called. Note these are in addition to kwargs that could be passed as part of the event post. If there's a conflict, the event-level ones will win.

Returns A GUID reference to the handler which you can use to later remove the handler via `remove_handler_by_key`. Though you don't need to remove the handler since the whole point of this method is they're automatically removed when the mode stops.

Note that if you do add a handler via this method and then remove it manually, that's ok too.

ball_drained(*balls=0*, ***kwargs*)
One or more balls has drained.

Drained balls will be subtracted from the number of balls in play.

Parameters **balls** – The number of balls that just drained.

Returns {balls: *number of balls drained*}

Return type A dictionary

ball_ending()
Handle ball ending.

DEPRECATED in v0.50. Use `end_ball()` instead.

balls_in_play
Property which holds the current number of balls in play.

Note that the number of balls in play is not necessarily the same as the number of balls that are active on the playfield. (For example, a ball could be held in a device while a show is playing, etc.)

You can set this property to change it, or get it's value.

If you set this value to 0, the ball ending process will be started.

configure_logging(*logger: str*, *console_level: str = 'basic'*, *file_level: str = 'basic'*)
Configure logging.

Parameters

- **logger** – The string name of the logger to use.
- **console_level** – The level of logging for the console. Valid options are "none", "basic", or "full".
- **file_level** – The level of logging for the console. Valid options are "none", "basic", or "full".

create_mode_devices() → None
Create new devices that are specified in a mode config that haven't been created in the machine-wide.

debug_log(*msg: str*, **args*, ***kwargs*) → None
Log a message at the debug level.

Note that whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

end_ball()
Set an event flag that will end the current ball.

end_game()
End the current game.

This triggers the game end manually.

error_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the error level.

These messages will always be shown in the console and the log file.

game_ending()
Handle game ending.

DEPRECATED in v0.50. Use `end_game()` instead.

ignorable_runtime_exception(*msg: str*) → None
Handle ignorable runtime exception.

During development or tests raise an exception for easier debugging. Log an error during production.

info_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the info level.

Whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

initialise_mode() → None
Initialise this mode.

is_game_mode
Return false.

We are the game and not a mode within the game.

load_mode_devices() → None
Load config of mode devices.

mode_will_start(***kwargs*) → None
User-overrideable method which will be called whenever this mode starts (i.e. before it becomes active).

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

request_player_add(***kwargs*)
Request to add a player to an active game.

This method contains the logic to verify whether it's ok to add a player. For example, the game must be on Ball 1 and the current number of players must be less than the max number allowed.

Assuming this method believes it's ok to add a player, it posts the boolean event *player_add_request* to give other modules the opportunity to deny it. For example, a credits module might deny the request if there are not enough credits in the machine.

If *player_add_request* comes back True, the event *player_added* is posted with a reference to the new player object as a *player* kwarg.

start(*mode_priority=None*, *callback=None*, ***kwargs*) → None
Start this mode.

Parameters

- **mode_priority** – Integer value of what you want this mode to run at. If you don't specify one, it will use the "Mode: priority" setting from this mode's configuration file.
- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode starts in the mode_start method which will be called automatically.

stop(*callback: Any = None*, ***kwargs*) → bool
Stop this mode.

Parameters

- **callback** – Method which will be called once this mode has stopped. Will only be called when the mode is running (includes currently stopping)
- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode stops in the mode_stop method which will be called automatically.

Returns true if the mode is running. Otherwise false.

warning_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the warning level.

These messages will always be shown in the console and the log file.

self.machine.modes.high_score

class mpf.modes.high_score.code.high_score.**HighScore**(*machine*, *config*, *name*, *path*)
Bases: mpf.core.async_mode.AsyncMode

High score mode.

Mode which runs during the game ending process to check for high scores and lets the players enter their names or initials.

Accessing the high_score mode via code

You can access the high_score mode from anywhere via `self.machine.modes.high_score`.

Methods & Attributes

The high_score mode has the following methods & attributes available. Note that methods & attributes inherited from the base Mode class are not included here.

active
Return *True* if this mode is active.

add_mode_event_handler(*event: str*, *handler: Callable*, *priority: int = 0*, ***kwargs*)
Register an event handler which is automatically removed when this mode stops.

This method is similar to the Event Manager's add_handler() method, except this method automatically unregisters the handlers when the mode ends.

Parameters

- **event** – String name of the event you're adding a handler for. Since events are text strings, they don't have to be pre-defined.
- **handler** – The method that will be called when the event is fired.
- **priority** – An arbitrary integer value that defines what order the handlers will be called in. The default is 1, so if you have a handler that you want to be called first, add it here with a priority of 2. (Or 3 or 10 or 100000.) The numbers don't matter. They're called from highest to lowest. (i.e. priority 100 is called before priority 1.)
- ****kwargs** – Any any additional keyword/argument pairs entered here will be attached to the handler and called whenever that handler is called. Note these are in addition to kwargs that could be passed as part of the event post. If there's a conflict, the event-level ones will win.

Returns A GUID reference to the handler which you can use to later remove the handler via `remove_handler_by_key`. Though you don't need to remove the handler since the whole point of this method is they're automatically removed when the mode stops.

Note that if you do add a handler via this method and then remove it manually, that's ok too.

configure_logging(*logger: str*, *console_level: str = 'basic'*, *file_level: str = 'basic'*)
Configure logging.

Parameters

- **logger** – The string name of the logger to use.
- **console_level** – The level of logging for the console. Valid options are "none", "basic", or "full".

- **file_level** – The level of logging for the console. Valid options are "none", "basic", or "full".

create_mode_devices() → None
Create new devices that are specified in a mode config that haven't been created in the machine-wide.

debug_log(*msg: str*, **args*, ***kwargs*) → None
Log a message at the debug level.

Note that whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

error_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the error level.

These messages will always be shown in the console and the log file.

ignorable_runtime_exception(*msg: str*) → None
Handle ignorable runtime exception.

During development or tests raise an exception for easier debugging. Log an error during production.

info_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the info level.

Whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

initialise_mode() → None
Initialise this mode.

is_game_mode
Return true if this is a game mode.

load_mode_devices() → None
Load config of mode devices.

mode_will_start(***kwargs*) → None
User-overrideable method which will be called whenever this mode starts (i.e. before it becomes active).

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

start(*mode_priority=None*, *callback=None*, ***kwargs*) → None
Start this mode.

Parameters

- **mode_priority** – Integer value of what you want this mode to run at. If you don't specify one, it will use the "Mode: priority" setting from this mode's configuration file.
- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode starts in the mode_start method which will be called automatically.

stop(*callback: Any = None*, ***kwargs*) → bool
Stop this mode.

Parameters

- **callback** – Method which will be called once this mode has stopped. Will only be called when the mode is running (includes currently stopping)

- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode stops in the mode_stop method which will be called automatically.

Returns true if the mode is running. Otherwise false.

warning_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the warning level.

These messages will always be shown in the console and the log file.

self.machine.modes.match

class mpf.modes.match.code.match.**Match**(*machine*, *config*, *name*, *path*)
Bases: mpf.core.async_mode.AsyncMode

Match mode.

Accessing the match mode via code

You can access the match mode from anywhere via `self.machine.modes.match`.

Methods & Attributes

The match mode has the following methods & attributes available. Note that methods & attributes inherited from the base Mode class are not included here.

active
Return *True* if this mode is active.

add_mode_event_handler(*event: str*, *handler: Callable*, *priority: int = 0*, ***kwargs*)
Register an event handler which is automatically removed when this mode stops.

This method is similar to the Event Manager's add_handler() method, except this method automatically unregisters the handlers when the mode ends.

Parameters

- **event** – String name of the event you're adding a handler for. Since events are text strings, they don't have to be pre-defined.
- **handler** – The method that will be called when the event is fired.
- **priority** – An arbitrary integer value that defines what order the handlers will be called in. The default is 1, so if you have a handler that you want to be called first, add it here with a priority of 2. (Or 3 or 10 or 100000.) The numbers don't matter. They're called from highest to lowest. (i.e. priority 100 is called before priority 1.)
- ****kwargs** – Any any additional keyword/argument pairs entered here will be attached to the handler and called whenever that handler is called. Note these are in addition to kwargs that could be passed as part of the event post. If there's a conflict, the event-level ones will win.

Returns A GUID reference to the handler which you can use to later remove the handler via `remove_handler_by_key`. Though you don't need to remove the handler since the whole point of this method is they're automatically removed when the mode stops.

Note that if you do add a handler via this method and then remove it manually, that's ok too.

configure_logging(*logger: str*, *console_level: str = 'basic'*, *file_level: str = 'basic'*)
Configure logging.

Parameters

- **logger** – The string name of the logger to use.
- **console_level** – The level of logging for the console. Valid options are "none", "basic", or "full".
- **file_level** – The level of logging for the console. Valid options are "none", "basic", or "full".

create_mode_devices() → None
Create new devices that are specified in a mode config that haven't been created in the machine-wide.

debug_log(*msg: str*, **args*, ***kwargs*) → None
Log a message at the debug level.

Note that whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

error_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the error level.

These messages will always be shown in the console and the log file.

ignorable_runtime_exception(*msg: str*) → None
Handle ignorable runtime exception.

During development or tests raise an exception for easier debugging. Log an error during production.

info_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the info level.

Whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

initialise_mode() → None
Initialise this mode.

is_game_mode
Return true if this is a game mode.

load_mode_devices() → None
Load config of mode devices.

mode_will_start(***kwargs*) → None
User-overrideable method which will be called whenever this mode starts (i.e. before it becomes active).

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

start(*mode_priority=None*, *callback=None*, ***kwargs*) → None
Start this mode.

Parameters

- **mode_priority** – Integer value of what you want this mode to run at. If you don't specify one, it will use the "Mode: priority" setting from this mode's configuration file.
- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode starts in the mode_start method which will be called automatically.

stop(*callback: Any = None*, ***kwargs*) → bool
Stop this mode.

Parameters

- **callback** – Method which will be called once this mode has stopped. Will only be called when the mode is running (includes currently stopping)
- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode stops in the mode_stop method which will be called automatically.

Returns true if the mode is running. Otherwise false.

warning_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the warning level.

These messages will always be shown in the console and the log file.

self.machine.modes.service

class mpf.modes.service.code.service.**Service**(*machine: MachineController*, *config: dict*, *name: str*, *path: str*)

Bases: mpf.core.async_mode.AsyncMode

The service mode.

Accessing the service mode via code

You can access the service mode from anywhere via `self.machine.modes.service`.

Methods & Attributes

The service mode has the following methods & attributes available. Note that methods & attributes inherited from the base Mode class are not included here.

active
Return *True* if this mode is active.

add_mode_event_handler(*event: str*, *handler: Callable*, *priority: int = 0*, ***kwargs*)
Register an event handler which is automatically removed when this mode stops.

This method is similar to the Event Manager's add_handler() method, except this method automatically unregisters the handlers when the mode ends.

Parameters

- **event** – String name of the event you're adding a handler for. Since events are text strings, they don't have to be pre-defined.
- **handler** – The method that will be called when the event is fired.
- **priority** – An arbitrary integer value that defines what order the handlers will be called in. The default is 1, so if you have a handler that you want to be called first, add it here with a priority of 2. (Or 3 or 10 or 100000.) The numbers don't matter. They're called from highest to lowest. (i.e. priority 100 is called before priority 1.)
- ****kwargs** – Any any additional keyword/argument pairs entered here will be attached to the handler and called whenever that handler is called. Note these are in addition to kwargs that could be passed as part of the event post. If there's a conflict, the event-level ones will win.

Returns A GUID reference to the handler which you can use to later remove the handler via `remove_handler_by_key`. Though you don't need to remove the handler since the whole point of this method is they're automatically removed when the mode stops.

Note that if you do add a handler via this method and then remove it manually, that's ok too.

configure_logging(*logger: str*, *console_level: str = 'basic'*, *file_level: str = 'basic'*)
Configure logging.

Parameters

- **logger** – The string name of the logger to use.
- **console_level** – The level of logging for the console. Valid options are "none", "basic", or "full".
- **file_level** – The level of logging for the console. Valid options are "none", "basic", or "full".

create_mode_devices() → None
Create new devices that are specified in a mode config that haven't been created in the machine-wide.

debug_log(*msg: str*, **args*, ***kwargs*) → None
Log a message at the debug level.

Note that whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

error_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the error level.

These messages will always be shown in the console and the log file.

ignorable_runtime_exception(*msg: str*) → None
Handle ignorable runtime exception.

During development or tests raise an exception for easier debugging. Log an error during production.

info_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the info level.

Whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

initialise_mode() → None
Initialise this mode.

is_game_mode
Return true if this is a game mode.

load_mode_devices() → None
Load config of mode devices.

mode_will_start(**kwargs) → None
User-overrideable method which will be called whenever this mode starts (i.e. before it becomes active).

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

start(*mode_priority=None*, *callback=None*, **kwargs) → None
Start this mode.

Parameters

- **mode_priority** – Integer value of what you want this mode to run at. If you don't specify one, it will use the "Mode: priority" setting from this mode's configuration file.
- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode starts in the mode_start method which will be called automatically.

stop(*callback: Any = None*, **kwargs) → bool
Stop this mode.

Parameters

- **callback** – Method which will be called once this mode has stopped. Will only be called when the mode is running (includes currently stopping)
- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode stops in the mode_stop method which will be called automatically.

Returns true if the mode is running. Otherwise false.

warning_log(*msg: str*, **args*, *context=None*, **kwargs) → None
Log a message at the warning level.

These messages will always be shown in the console and the log file.

self.machine.modes.tilt

class mpf.modes.tilt.code.tilt.**Tilt**(*machine: mpf.core.machine.MachineController*, *config: dict*, *name: str*, *path*)
Bases: *mpf.core.mode.Mode*

A mode which handles a tilt in a pinball machine.

Note that this mode is always running (even during attract mode) since the machine needs to watch for slam tilts at all times.

Accessing the tilt mode via code

You can access the tilt mode from anywhere via `self.machine.modes.tilt`.

Methods & Attributes

The tilt mode has the following methods & attributes available. Note that methods & attributes inherited from the base Mode class are not included here.

active
Return *True* if this mode is active.

add_mode_event_handler(*event: str*, *handler: Callable*, *priority: int = 0*, ***kwargs*)
Register an event handler which is automatically removed when this mode stops.

This method is similar to the Event Manager's add_handler() method, except this method automatically unregisters the handlers when the mode ends.

Parameters

- **event** – String name of the event you're adding a handler for. Since events are text strings, they don't have to be pre-defined.
- **handler** – The method that will be called when the event is fired.
- **priority** – An arbitrary integer value that defines what order the handlers will be called in. The default is 1, so if you have a handler that you want to be called first, add it here with a priority of 2. (Or 3 or 10 or 100000.) The numbers don't matter. They're called from highest to lowest. (i.e. priority 100 is called before priority 1.)
- ****kwargs** – Any any additional keyword/argument pairs entered here will be attached to the handler and called whenever that handler is called. Note these are in addition to kwargs that could be passed as part of the event post. If there's a conflict, the event-level ones will win.

Returns A GUID reference to the handler which you can use to later remove the handler via `remove_handler_by_key`. Though you don't need to remove the handler since the whole point of this method is they're automatically removed when the mode stops.

Note that if you do add a handler via this method and then remove it manually, that's ok too.

configure_logging(*logger: str*, *console_level: str = 'basic'*, *file_level: str = 'basic'*)
Configure logging.

Parameters

- **logger** – The string name of the logger to use.
- **console_level** – The level of logging for the console. Valid options are "none", "basic", or "full".
- **file_level** – The level of logging for the console. Valid options are "none", "basic", or "full".

create_mode_devices() → None
Create new devices that are specified in a mode config that haven't been created in the machine-wide.

debug_log(*msg: str*, **args*, ***kwargs*) → None
Log a message at the debug level.

Note that whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

error_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the error level.

These messages will always be shown in the console and the log file.

ignorable_runtime_exception(*msg: str*) → None
Handle ignorable runtime exception.

During development or tests raise an exception for easier debugging. Log an error during production.

info_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the info level.

Whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

initialise_mode() → None
Initialise this mode.

is_game_mode
Return true if this is a game mode.

load_mode_devices() → None
Load config of mode devices.

mode_will_start(***kwargs*) → None
User-overrideable method which will be called whenever this mode starts (i.e. before it becomes active).

raise_config_error(*msg*, *error_no*, ***, *context=None*)
Raise a ConfigFileError exception.

reset_warnings(***kwargs*)
Reset the tilt warnings for the current player.

slam_tilt(***kwargs*)
Process a slam tilt.

This method posts the *slam_tilt* event and (if a game is active) sets the game mode's `slam_tilted` attribute to *True*.

start(*mode_priority=None*, *callback=None*, ***kwargs*) → None
Start this mode.

Parameters

- **mode_priority** – Integer value of what you want this mode to run at. If you don't specify one, it will use the "Mode: priority" setting from this mode's configuration file.
- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode starts in the mode_start method which will be called automatically.

stop(*callback: Any = None*, ***kwargs*) → bool
Stop this mode.

Parameters

- **callback** – Method which will be called once this mode has stopped. Will only be called when the mode is running (includes currently stopping)
- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode stops in the mode_stop method which will be called automatically.

Returns true if the mode is running. Otherwise false.

tilt(***kwargs*)
Cause the ball to tilt.

This will post an event called *tilt*, set the game mode's `tilted` attribute to *True*, disable the flippers and autofire devices, end the current ball, and wait for all the balls to drain.

tilt_settle_ms_remaining()
Return the amount of milliseconds remaining until the tilt settle time has cleared.

Returns Integer of the number of ms remaining until tilt settled is cleared.

tilt_warning(***kwargs*)
Process a tilt warning.

If the number of warnings is the number to cause a tilt, a tilt will be processed.

warning_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the warning level.

These messages will always be shown in the console and the log file.

8.3.4 Hardware Platforms

Hardware platforms are stored in a machine `hardware_platforms` dictionary, for example, `self.machine.hardware_platforms['fast']` or `self.machine.hardware_platforms['p_roc']`.

self.machine.hardware_platforms['fadecandy']

class mpf.platforms.fadecandy.**FadecandyHardwarePlatform**(*machine: MachineController*)
Bases: *mpf.platforms.openpixel.OpenpixelHardwarePlatform*

Base class for the FadeCandy hardware platform.

Accessing the fadecandy platform via code

Hardware platforms are stored in the `self.machine.hardware_platforms` dictionary, so the fadecandy platform is available via `self.machine.hardware_platforms['fadecandy']`.

Methods & Attributes

The fadecandy platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

self.machine.hardware_platforms['fast']

class mpf.platforms.fast.fast.**FastHardwarePlatform**(*machine*)

Bases: mpf.core.platform.ServoPlatform, mpf.core.platform.LightsPlatform, mpf.core.platform.DmdPlatform, mpf.core.platform.SwitchPlatform, mpf.core.platform.DriverPlatform

Platform class for the FAST hardware controller.

Accessing the fast platform via code

Hardware platforms are stored in the `self.machine.hardware_platforms` dictionary, so the fast platform is available via `self.machine.hardware_platforms['fast']`.

Methods & Attributes

The fast platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

clear_hw_rule(*switch*, *coil*)

Clear a hardware rule.

This is used if you want to remove the linkage between a switch and some driver activity. For example, if you wanted to disable your flippers (so that a player pushing the flipper buttons wouldn't cause the flippers to flip), you'd call this method with your flipper button as the *sw_num*.

Parameters

- **switch** – The switch whose rule you want to clear.
- **coil** – The coil whose rule you want to clear.

configure_dmd()

Configure a hardware DMD connected to a FAST controller.

configure_driver(*config: mpf.core.platform.DriverConfig*, *number: str*, *platform_settings: dict*) → mpf.platforms.fast.fast_driver.FASTDriver

Configure a driver.

Parameters **config** – Driver config.

Returns: Driver object

configure_light(*number*, *subtype*, *platform_settings*) → mpf.platforms.interfaces.light_platform_interface.LightPlatformI

Configure light in platform.

configure_servo(*number: str*)

Configure a servo.

Parameters **number** – Number of servo

Returns: Servo object.

configure_switch(*number: str*, *config: mpf.core.platform.SwitchConfig*, *platform_config: dict*) → mpf.platforms.fast.fast_switch.FASTSwitch

Configure the switch object for a FAST Pinball controller.

FAST Controllers support two types of switches: *local* and *network*. Local switches are switches that are connected to the FAST controller board itself, and network switches are those connected to a FAST I/O board.

MPF needs to know which type of switch is this is. You can specify the switch's connection type in the config file via the `connection:` setting (either `local` or `network`).

If a connection type is not specified, this method will use some intelligence to try to figure out which default should be used.

If the DriverBoard type is `fast`, then it assumes the default is `network`. If it's anything else (`wpc`, `system11`, `bally`, etc.) then it assumes the connection type is `local`. Connection types can be mixed and matched in the same machine.

Parameters **config** – Switch config.

Returns: Switch object.

static convert_number_from_config(*number*)
Convert a number from config format to hex.

classmethod get_coil_config_section()
Return coil config section.

get_hw_switch_states()
Return hardware states.

get_info_string()
Dump infos about boards.

classmethod get_switch_config_section()
Return switch config section.

initialize()
Initialise platform.

parse_light_number_to_channels(*number: str*, *subtype: str*)
Parse light channels from number string.

process_received_message(*msg: str*)
Send an incoming message from the FAST controller to the proper method for servicing.

Parameters **msg** – messaged which was received

receive_local_closed(*msg*)
Process local switch closed.

Parameters **msg** – switch number

receive_local_open(*msg*)
Process local switch open.

Parameters **msg** – switch number

receive_nw_closed(*msg*)
Process network switch closed.

Parameters **msg** – switch number

receive_nw_open(*msg*)
Process network switch open.

Parameters **msg** – switch number

receive_sa(*msg*)
Receive all switch states.

Parameters **msg** – switch states as bytearray

register_io_board(*board*)
Register an IO board.

Parameters **board** – 'mpf.platform.fast.fast_io_board.FastIoBoard' to register

register_processor_connection(*name: str*, *communicator*)
Register processor.

Once a communication link has been established with one of the processors on the FAST board, this method lets the communicator let MPF know which processor it's talking to.

This is a separate method since we don't know which processor is on which serial port ahead of time.

Parameters

- **communicator** – communicator object
- **name** – name of processor

set_pulse_on_hit_and_enable_and_release_and_disable_rule(*enable_switch*, *disable_switch*, *coil*)
Set pulse on hit and enable and release and disable rule on driver.

set_pulse_on_hit_and_enable_and_release_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
Set pulse on hit and enable and relase rule on driver.

set_pulse_on_hit_and_release_rule(*enable_switch*, *coil*)
Set pulse on hit and release rule to driver.

set_pulse_on_hit_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
Set pulse on hit rule on driver.

start()
Start listening for commands and schedule watchdog.

stop()
Stop platform and close connections.

update_firmware() → str
Upgrade the firmware of the CPUs.

update_leds()
Update all the LEDs connected to a FAST controller.

This is done once per game loop for efficiency (i.e. all LEDs are sent as a single update rather than lots of individual ones).

Also, every LED is updated every loop, even if it doesn't change. This is in case some interference causes a LED to change color. Since we update every loop, it will only be the wrong color for one tick.

self.machine.hardware_platforms['i2c_servo_controller']

class mpf.platforms.i2c_servo_controller.**I2CServoControllerHardwarePlatform**(*machine*)
Bases: mpf.core.platform.ServoPlatform

Supports the PCA9685/PCA9635 chip via I2C.

Accessing the i2c_servo_controller platform via code

Hardware platforms are stored in the `self.machine.hardware_platforms` dictionary, so the i2c_servo_controller platform is available via `self.machine.hardware_platforms['i2c_servo_controller']`.

Methods & Attributes

The i2c_servo_controller platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

configure_servo(*number: str*)
Configure servo.

initialize()
Initialise platform.

stop()
Stop platform.

self.machine.hardware_platforms['lisy']

class mpf.platforms.lisy.lisy.**LisyHardwarePlatform**(*machine*)
Bases: mpf.core.platform.SwitchPlatform, mpf.core.platform.LightsPlatform, mpf.core.platform.DriverPlatform, mpf.core.platform.SegmentDisplaySoftwareFlashPlatform, mpf.core.platform.HardwareSoundPlatform, *mpf.core.logging.LogMixin*

LISY platform.

Accessing the lisy platform via code

Hardware platforms are stored in the `self.machine.hardware_platforms` dictionary, so the lisy platform is available via `self.machine.hardware_platforms['lisy']`.

Methods & Attributes

The lisy platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

clear_hw_rule(*switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
No rules on LISY.

configure_driver(*config: mpf.core.platform.DriverConfig*, *number: str*, *platform_settings: dict*)
→ mpf.platforms.interfaces.driver_platform_interface.DriverPlatformInterface
Configure a driver.

configure_hardware_sound_system() → mpf.platforms.interfaces.hardware_sound_platform_interface.HardwareSou
Configure hardware sound.

configure_light(*number: str*, *subtype: str*, *platform_settings: dict*) →
mpf.platforms.interfaces.light_platform_interface.LightPlatformSoftwareFade
Configure light on LISY.

configure_segment_display(*number: str*) → mpf.platforms.interfaces.segment_display_platform_interface.SegmentDi
Configure a segment display.

configure_switch(*number: str*, *config: mpf.core.platform.SwitchConfig*, *platform_config: dict*) → mpf.platforms.interfaces.switch_platform_interface.SwitchPlatformInterface
Configure a switch.

get_hw_switch_states()
Return current switch states.

initialize()
Initialise platform.

parse_light_number_to_channels(*number: str*, *subtype: str*)
Return a single light.

read_byte() → Generator[[int, None], int]
Read one byte.

read_string() → Generator[[int, None], bytes]
Read zero terminated string.

readuntil(*separator*, *min_chars: int = 0*)
Read until separator.

Parameters

- **separator** – Read until this separator byte.
- **min_chars** – Minimum message length before separator

send_byte(*cmd: int*, *byte: bytes = None*)
Send a command with optional payload.

send_string(*cmd: int*, *string: str*)
Send a command with null terminated string.

set_pulse_on_hit_and_enable_and_release_and_disable_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *disable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
No rules on LISY.

set_pulse_on_hit_and_enable_and_release_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
No rules on LISY.

set_pulse_on_hit_and_release_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
No rules on LISY.

set_pulse_on_hit_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
No rules on LISY.

start()
Start reading switch changes.

stop()
Stop platform.

self.machine.hardware_platforms['mma8451']

class mpf.platforms.mma8451.**MMA8451Platform**(*machine*)
Bases: mpf.core.platform.AccelerometerPlatform

MMA8451 accelerometer platform.

Accessing the mma8451 platform via code

Hardware platforms are stored in the `self.machine.hardware_platforms` dictionary, so the mma8451 platform is available via `self.machine.hardware_platforms['mma8451']`.

Methods & Attributes

The mma8451 platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

configure_accelerometer(*number*, *config*, *callback*) → mpf.platforms.mma8451.MMA8451Device
Configure MMA8451 accelerometer.

initialize()
Initialise MMA8451 platform.

stop()
Stop accelerometer poll tasks.

self.machine.hardware_platforms['mypinballs']

class mpf.platforms.mypinballs.mypinballs.**MyPinballsHardwarePlatform**(*machine*)
Bases: mpf.core.platform.SegmentDisplayPlatform

Hardware platform for MyPinballs 7-segment controller.

Accessing the mypinballs platform via code

Hardware platforms are stored in the `self.machine.hardware_platforms` dictionary, so the mypinballs platform is available via `self.machine.hardware_platforms['mypinballs']`.

Methods & Attributes

The mypinballs platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

configure_segment_display(*number: str*) → mpf.platforms.interfaces.segment_display_platform_interface.SegmentDi
Configure display.

initialize()
Initialise hardware.

send_cmd(*cmd: bytes*)
Send a byte command.

stop()
Stop platform.

self.machine.hardware_platforms['openpixel']

class mpf.platforms.openpixel.**OpenpixelHardwarePlatform**(*machine: MachineController*)

Bases: mpf.core.platform.LightsPlatform

Base class for the open pixel hardware platform.

Parameters **machine** – The main MachineController object.

Accessing the openpixel platform via code

Hardware platforms are stored in the self.machine.hardware_platforms dictionary, so the openpixel platform is available via self.machine.hardware_platforms['openpixel'].

Methods & Attributes

The openpixel platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

configure_light(*number*, *subtype*, *platform_settings*) → mpf.platforms.interfaces.light_platform_interface.LightPlatformI
Configure an LED.

initialize()
Initialise openpixel platform.

parse_light_number_to_channels(*number: str*, *subtype: str*)
Parse number to three channels.

stop()
Stop platform.

self.machine.hardware_platforms['opp']

class mpf.platforms.opp.opp.**OppHardwarePlatform**(*machine*)

Bases: mpf.core.platform.LightsPlatform, mpf.core.platform.SwitchPlatform, mpf.core.platform.DriverPlatform

Platform class for the OPP hardware.

Parameters **machine** – The main MachineController instance.

Accessing the opp platform via code

Hardware platforms are stored in the self.machine.hardware_platforms dictionary, so the opp platform is available via self.machine.hardware_platforms['opp'].

Methods & Attributes

The opp platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

clear_hw_rule(*switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
Clear a hardware rule.

This is used if you want to remove the linkage between a switch and some driver activity. For example, if you wanted to disable your flippers (so that a player pushing the flipper buttons wouldn't cause the flippers to flip), you'd call this method with your flipper button as the *sw_num*.

configure_driver(*config: mpf.core.platform.DriverConfig*, *number: str*, *platform_settings: dict*)
Configure a driver.

Parameters **config** – Config dict.

configure_light(*number*, *subtype*, *platform_settings*)
Configure a led or matrix light.

configure_switch(*number: str*, *config: mpf.core.platform.SwitchConfig*, *platform_config: dict*)
Configure a switch.

Parameters **config** – Config dict.

static eom_resp(*chain_serial*, *msg*)
Process an EOM.

Parameters

- **chain_serial** – Serial of the chain which received the message.
- **msg** – Message to parse.

classmethod get_coil_config_section()
Return coil config section.

get_gen2_cfg_resp(*chain_serial*, *msg*)
Process cfg response.

Parameters

- **chain_serial** – Serial of the chain which received the message.
- **msg** – Message to parse.

get_hw_switch_states()
Get initial hardware switch states.

This changes switches from active low to active high

get_info_string()
Dump infos about boards.

initialize()
Initialise connections to OPP hardware.

inv_resp(*chain_serial*, *msg*)
Parse inventory response.

Parameters

- **chain_serial** – Serial of the chain which received the message.
- **msg** – Message to parse.

light_sync()
Update lights.

Currently we only update neo pixels. Incands are updated separately in a task to provide better batching.

parse_light_number_to_channels(*number: str*, *subtype: str*)
Parse number and subtype to channel.

process_received_message(*chain_serial*, *msg*)
Send an incoming message from the OPP hardware to the proper method for servicing.

Parameters

- **chain_serial** – Serial of the chain which received the message.
- **msg** – Message to parse.

read_gen2_inp_resp(*chain_serial*, *msg*)
Read switch changes.

Parameters

- **chain_serial** – Serial of the chain which received the message.
- **msg** – Message to parse.

read_gen2_inp_resp_initial(*chain_serial*, *msg*)
Read initial switch states.

Parameters

- **chain_serial** – Serial of the chain which received the message.
- **msg** – Message to parse.

read_matrix_inp_resp(*chain_serial*, *msg*)
Read matrix switch changes.

Parameters

- **chain_serial** – Serial of the chain which received the message.
- **msg** – Message to parse.

read_matrix_inp_resp_initial(*chain_serial*, *msg*)
Read initial matrix switch states.

Parameters

- **chain_serial** – Serial of the chain which received the message.
- **msg** – Message to parse.

register_processor_connection(*serial_number*, *communicator*)
Register the processors to the platform.

Parameters

- **serial_number** – Serial number of chain.
- **communicator** – Instance of OPPSerialCommunicator

send_to_processor(*chain_serial*, *msg*)
Send message to processor with specific serial number.

Parameters

- **chain_serial** – Serial of the processor.

- **msg** – Message to send.

set_pulse_on_hit_and_enable_and_release_and_disable_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *disable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)

Set pulse on hit and enable and release and disable rule on driver.

Pulses a driver when a switch is hit. Then enables the driver (may be with pwm). When the switch is released the pulse is canceled and the driver gets disabled. When the second disable_switch is hit the pulse is canceled and the driver gets disabled. Typically used on the main coil for dual coil flippers with eos switch.

set_pulse_on_hit_and_enable_and_release_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)

Set pulse on hit and enable and relase rule on driver.

Pulses a driver when a switch is hit. Then enables the driver (may be with pwm). When the switch is released the pulse is canceled and the driver gets disabled. Typically used for single coil flippers.

set_pulse_on_hit_and_release_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)

Set pulse on hit and release rule to driver.

Pulses a driver when a switch is hit. When the switch is released the pulse is canceled. Typically used on the main coil for dual coil flippers without eos switch.

set_pulse_on_hit_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)

Set pulse on hit rule on driver.

Pulses a driver when a switch is hit. When the switch is released the pulse continues. Typically used for autofire coils such as pop bumpers.

start()

Start polling and listening for commands.

stop()

Stop hardware and close connections.

update_incand()

Update all the incandescents connected to OPP hardware.

This is done once per game loop if changes have been made.

It is currently assumed that the UART oversampling will guarantee proper communication with the boards. If this does not end up being the case, this will be changed to update all the incandescents each loop.

vers_resp(*chain_serial*, *msg*)

Process version response.

Parameters

- **chain_serial** – Serial of the chain which received the message.
- **msg** – Message to parse.

self.machine.hardware_platforms['p3_roc']

class mpf.platforms.p3_roc.**P3RocHardwarePlatform**(*machine*)
Bases: mpf.platforms.p_roc_common.PROCBasePlatform, mpf.core.platform.I2cPlatform, mpf.core.platform.AccelerometerPlatform

Platform class for the P3-ROC hardware controller.

Parameters **machine** – The MachineController instance.

machine
The MachineController instance.

Accessing the p3_roc platform via code

Hardware platforms are stored in the `self.machine.hardware_platforms` dictionary, so the p3_roc platform is available via `self.machine.hardware_platforms['p3_roc']`.

Methods & Attributes

The p3_roc platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

configure_accelerometer(*number*, *config*, *callback*)
Configure the accelerometer on the P3-ROC.

configure_driver(*config: mpf.core.platform.DriverConfig*, *number: str*, *platform_settings: dict*)
Create a P3-ROC driver.

Typically drivers are coils or flashers, but for the P3-ROC this is also used for matrix-based lights.

Parameters **config** – Dictionary of settings for the driver.

Returns A reference to the PROCDriver object which is the actual object you can use to pulse(), patter(), enable(), etc.

configure_i2c(*number: str*)
Configure I2C device on P3-Roc.

configure_switch(*number: str*, *config: mpf.core.platform.SwitchConfig*, *platform_config: dict*)
Configure a P3-ROC switch.

Parameters **config** – Dictionary of settings for the switch. In the case of the P3-ROC, it uses the following:

Returns: A configured switch object.

connect()
Connect to the P3-Roc.

get_hw_switch_states()
Read in and set the initial switch state.

The P-ROC uses the following values for hw switch states: 1 - closed (debounced) 2 - open (debounced) 3 - closed (not debounced) 4 - open (not debounced)

get_info_string()
Dump infos about boards.

classmethod scale_accelerometer_to_g(*raw_value*)
Convert internal representation to g.

tick()
Check the P3-ROC for any events (switch state changes).

Also tickles the watchdog and flushes any queued commands to the P3-ROC.

self.machine.hardware_platforms['p_roc']

class mpf.platforms.p_roc.**PRocHardwarePlatform**(*machine*)
Bases: mpf.platforms.p_roc_common.PROCBasePlatform, mpf.core.platform.DmdPlatform, mpf.core.platform.SegmentDisplayPlatform

Platform class for the P-ROC hardware controller.

Parameters **machine** – The MachineController instance.

machine
The MachineController instance.

Accessing the p_roc platform via code

Hardware platforms are stored in the `self.machine.hardware_platforms` dictionary, so the p_roc platform is available via `self.machine.hardware_platforms['p_roc']`.

Methods & Attributes

The p_roc platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

configure_dmd()
Configure a hardware DMD connected to a classic P-ROC.

configure_driver(*config: mpf.core.platform.DriverConfig*, *number: str*, *platform_settings: dict*)
Create a P-ROC driver.

Typically drivers are coils or flashers, but for the P-ROC this is also used for matrix-based lights.

Parameters **config** – Dictionary of settings for the driver.

Returns A reference to the PROCDriver object which is the actual object you can use to pulse(), patter(), enable(), etc.

configure_segment_display(*number: str*) → mpf.platforms.interfaces.segment_display_platform_interface.SegmentDi
Configure display.

configure_switch(*number: str*, *config: mpf.core.platform.SwitchConfig*, *platform_config: dict*)
Configure a P-ROC switch.

Parameters

- **number** – String number of the switch to configure.
- **config** – SwitchConfig settings.

Returns: A configured switch object.

get_hw_switch_states()
Read in and set the initial switch state.

The P-ROC uses the following values for hw switch states: 1 - closed (debounced) 2 - open (debounced) 3 - closed (not debounced) 4 - open (not debounced)

get_info_string()
Dump infos about boards.

tick()
Check the P-ROC for any events (switch state changes or notification that a DMD frame was updated).

Also tickles the watchdog and flushes any queued commands to the P-ROC.

self.machine.hardware_platforms['pololu_maestro']

class mpf.platforms.pololu_maestro.**PololuMaestroHardwarePlatform**(*machine*)
Bases: mpf.core.platform.ServoPlatform

Supports the Pololu Maestro servo controllers via PySerial.

Works with Micro Maestro 6, and Mini Maestro 12, 18, and 24.

Accessing the pololu_maestro platform via code

Hardware platforms are stored in the `self.machine.hardware_platforms` dictionary, so the pololu_maestro platform is available via `self.machine.hardware_platforms['pololu_maestro']`.

Methods & Attributes

The pololu_maestro platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

configure_servo(*number: str*)
Configure a servo device in platform.

Parameters **config** (*dict*) – Configuration of device

initialize()
Initialise platform.

stop()
Close serial.

self.machine.hardware_platforms['rpi']

class mpf.platforms.rpi.rpi.**RaspberryPiHardwarePlatform**(*machine*)
Bases: mpf.core.platform.SwitchPlatform, mpf.core.platform.DriverPlatform, mpf.core.platform.ServoPlatform, mpf.core.platform.I2cPlatform

Control the hardware of a Raspberry Pi.

Works locally and remotely via network.

Accessing the rpi platform via code

Hardware platforms are stored in the `self.machine.hardware_platforms` dictionary, so the rpi platform is available via `self.machine.hardware_platforms['rpi']`.

Methods & Attributes

The rpi platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

clear_hw_rule(*switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
Raise exception.

configure_driver(*config: mpf.core.platform.DriverConfig*, *number: str*, *platform_settings: dict*) → mpf.platforms.interfaces.driver_platform_interface.DriverPlatformInterface
Configure an output on the Raspberry Pi.

configure_i2c(*number: str*)
Configure I2c device.

configure_servo(*number: str*) → mpf.platforms.interfaces.servo_platform_interface.ServoPlatformInterface
Configure a servo.

configure_switch(*number: str*, *config: mpf.core.platform.SwitchConfig*, *platform_config: dict*) → mpf.platforms.interfaces.switch_platform_interface.SwitchPlatformInterface
Configure a switch with pull up.

get_hw_switch_states()
Return current switch states.

initialize()
Initialise platform.

send_command(*cmd*)
Add a command to the command queue.

set_pulse_on_hit_and_enable_and_release_and_disable_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *disable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
Raise exception.

set_pulse_on_hit_and_enable_and_release_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
Raise exception.

set_pulse_on_hit_and_release_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
Raise exception.

set_pulse_on_hit_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
Raise exception.

stop()
Stop platform.

self.machine.hardware_platforms['smart_virtual']

class mpf.platforms.smart_virtual.**SmartVirtualHardwarePlatform**(*machine*)
Bases: *mpf.platforms.virtual.VirtualHardwarePlatform*

Base class for the smart_virtual hardware platform.

Accessing the smart_virtual platform via code

Hardware platforms are stored in the `self.machine.hardware_platforms` dictionary, so the smart_virtual platform is available via `self.machine.hardware_platforms['smart_virtual']`.

Methods & Attributes

The smart_virtual platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

add_ball_to_device(*device*)
Add ball to device.

configure_driver(*config: mpf.core.platform.DriverConfig*, *number: str*, *platform_settings: dict*)
Configure driver.

start()
Initialise platform when all devices are ready.

self.machine.hardware_platforms['smartmatrix']

class mpf.platforms.smartmatrix.**SmartMatrixHardwarePlatform**(*machine*)
Bases: `mpf.core.platform.RgbDmdPlatform`

SmartMatrix RGB DMD.

Accessing the smartmatrix platform via code

Hardware platforms are stored in the `self.machine.hardware_platforms` dictionary, so the smartmatrix platform is available via `self.machine.hardware_platforms['smartmatrix']`.

Methods & Attributes

The smartmatrix platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

configure_rgb_dmd(*name: str*)
Configure rgb dmd.

initialize()
Initialise platform.

stop()
Stop platform.

self.machine.hardware_platforms['smbus2']

class mpf.platforms.smbus2.**Smbus2**(*machine*)
Bases: mpf.core.platform.I2cPlatform

I2C platform which uses the smbus interface on linux via the smbus2 python extension.

Accessing the smbus2 platform via code

Hardware platforms are stored in the `self.machine.hardware_platforms` dictionary, so the smbus2 platform is available via `self.machine.hardware_platforms['smbus2']`.

Methods & Attributes

The smbus2 platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

configure_i2c(*number: str*) → Generator[[int, None], mpf.platforms.smbus2.Smbus2I2cDevice]
Configure device on smbus2.

initialize()
Check if smbus2 extension has been imported.

self.machine.hardware_platforms['snux']

class mpf.platforms.snux.**SnuxHardwarePlatform**(*machine: mpf.core.machine.MachineController*)
Bases: mpf.core.platform.DriverPlatform

Overlay platform for the snux hardware board.

Accessing the snux platform via code

Hardware platforms are stored in the `self.machine.hardware_platforms` dictionary, so the snux platform is available via `self.machine.hardware_platforms['snux']`.

Methods & Attributes

The snux platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

a_side_busy
Return if A side cannot be switches off right away.

c_side_active
Return if C side cannot be switches off right away.

clear_hw_rule(*switch*, *coil*)
Clear a rule for a driver on the snux board.

configure_driver(*config: mpf.core.platform.DriverConfig*, *number: str*, *platform_settings: dict*)
Configure a driver on the snux board.

Parameters **config** – Driver config dict

driver_action(*driver, pulse_settings: Optional[mpf.platforms.interfaces.driver_platform_interface.PulseSettings], hold_settings: Optional[mpf.platforms.interfaces.driver_platform_interface.HoldSettings]*)
Add a driver action for a switched driver to the queue (for either the A-side or C-side queue).

Parameters

- **driver** – A reference to the original platform class Driver instance.
- **pulse_settings** – Settings for the pulse or None
- **hold_settings** – Settings for hold or None

This action will be serviced immediately if it can, or ASAP otherwise.

initialize()
Automatically called by the Platform class after all the core modules are loaded.

set_pulse_on_hit_and_enable_and_release_and_disable_rule(*enable_switch*, *disable_switch*, *coil*)
Configure a rule for a driver on the snux board.

Will pass the call onto the parent platform if the driver is not on A/C relay.

set_pulse_on_hit_and_enable_and_release_rule(*enable_switch*, *coil*)
Configure a rule for a driver on the snux board.

Will pass the call onto the parent platform if the driver is not on A/C relay.

set_pulse_on_hit_and_release_rule(*enable_switch*, *coil*)
Configure a rule for a driver on the snux board.

Will pass the call onto the parent platform if the driver is not on A/C relay.

set_pulse_on_hit_rule(*enable_switch*, *coil*)
Configure a rule on the snux board.

Will pass the call onto the parent platform if the driver is not on A/C relay.

stop()
Stop the overlay. Nothing to do here because stop is also called on parent platform.

tick()
Snux main loop.

Called based on the timer_tick event

validate_coil_section(*driver*, *config*)
Validate coil config for platform.

self.machine.hardware_platforms['spike']

class mpf.platforms.spike.spike.**SpikePlatform**(*machine*)
Bases: mpf.core.platform.SwitchPlatform, mpf.core.platform.LightsPlatform, mpf.core.platform.DriverPlatform, mpf.core.platform.DmdPlatform

Stern Spike Platform.

Accessing the spike platform via code

Hardware platforms are stored in the `self.machine.hardware_platforms` dictionary, so the spike platform is available via `self.machine.hardware_platforms['spike']`.

Methods & Attributes

The spike platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

clear_hw_rule(*switch*, *coil*)
Disable hardware rule for this coil.

configure_dmd()
Configure a DMD.

configure_driver(*config: mpf.core.platform.DriverConfig*, *number: str*, *platform_settings: dict*)
Configure a driver on Stern Spike.

configure_light(*number*, *subtype*, *platform_settings*) → mpf.platforms.spike.spike.SpikeLight
Configure a light on Stern Spike.

configure_switch(*number: str*, *config: mpf.core.platform.SwitchConfig*, *platform_config: dict*)
Configure switch on Stern Spike.

get_hw_switch_states()
Return current switch states.

initialize()
Initialise platform.

parse_light_number_to_channels(*number: str*, *subtype: str*)
Return a single light.

send_cmd_and_wait_for_response(*node*, *cmd*, *data*, *response_len*) → Generator[[int, None], Optional[bytearray]]
Send cmd and wait for response.

send_cmd_async(*node*, *cmd*, *data*)
Send cmd which does not require a response.

send_cmd_raw(*data*, *wait_ms=0*)
Send raw command.

send_cmd_sync(*node*, *cmd*, *data*)
Send cmd which does not require a response.

set_pulse_on_hit_and_enable_and_release_and_disable_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *disable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
Set pulse on hit and release rule to driver.

Used for high-power coil on dual-wound flippers. Example from WWE: Type: 8 Cmd: 65 Node: 8 Msg: 0x00 0xff 0x33 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x42 0x40 0x00 0x02 0x06 0x00 Len: 25

set_pulse_on_hit_and_enable_and_release_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
Set pulse on hit and enable and relase rule on driver.

Used for single coil flippers. Examples from WWE: Dual-wound flipper hold coil: Type: 8 Cmd: 65 Node: 8 Msg: 0x02 0xff 0x46 0x01 0xff 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x3a 0x00 0x42 0x40 0x00 0x00 0x01 0x00 Len: 25

Ring Slings (different flags): Type: 8 Cmd: 65 Node: 10 Msg: 0x00 0xff 0x19 0x00 0x14 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x80 0x00 0x4a 0x40 0x00 0x00 0x06 0x05 Len: 25

set_pulse_on_hit_and_release_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)

Set pulse on hit and release rule to driver.

I believe that param2 == 1 means that it will cancel the pulse when the switch is released.

Used for high-power coils on dual-wound flippers. Example from WWE: Type: 8 Cmd: 65 Node: 8 Msg: 0x03 0xff 0x46 0x01 0xff 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x43 0x40 0x00 0x00 0x01 0x00 Len: 25

set_pulse_on_hit_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)

Set pulse on hit rule on driver.

This is mostly used for popbumpers. Example from WWE: Type: 8 Cmd: 65 Node: 9 Msg: 0x00 0xa6 0x28 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x14 0x00 0x00 0x00 0x38 0x00 0x40 0x00 0x00 0x00 0x00 0x00 Len: 25

stop()

Stop hardware and close connections.

self.machine.hardware_platforms['trinamics_steprocker']

class mpf.platforms.trinamics_steprocker.**TrinamicsStepRocker**(*machine*)

Bases: mpf.core.platform.StepperPlatform

Supports the Trinamics Step Rocker via PySerial.

Works with Trinamics Step Rocker. TBD other 'TMCL' based steppers eval boards

Accessing the trinamics_steprocker platform via code

Hardware platforms are stored in the `self.machine.hardware_platforms` dictionary, so the trinamics_steprocker platform is available via `self.machine.hardware_platforms['trinamics_steprocker']`.

Methods & Attributes

The trinamics_steprocker platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

configure_stepper(*number: str*, *config: dict*) → mpf.platforms.trinamics_steprocker.TrinamicsTMCLStepper

Configure a smart stepper device in platform.

Parameters **config** (*dict*) – Configuration of device

classmethod get_stepper_config_section()

Return config validator name.

initialize()

Initialise trinamics steprocker platform.

stop()

Close serial.

self.machine.hardware_platforms['virtual']

class mpf.platforms.virtual.**VirtualHardwarePlatform**(*machine*)

Bases: mpf.core.platform.AccelerometerPlatform, mpf.core.platform.I2cPlatform, mpf.core.platform.ServoPlatform, mpf.core.platform.LightsPlatform, mpf.core.platform.SwitchPlatform, mpf.core.platform.DriverPlatform, mpf.core.platform.DmdPlatform, mpf.core.platform.RgbDmdPlatform, mpf.core.platform.SegmentDisplayPlatform, mpf.core.platform.StepperPlatform, mpf.core.platform.HardwareSoundPlatform

Base class for the virtual hardware platform.

Accessing the virtual platform via code

Hardware platforms are stored in the `self.machine.hardware_platforms` dictionary, so the virtual platform is available via `self.machine.hardware_platforms['virtual']`.

Methods & Attributes

The virtual platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

clear_hw_rule(*switch*, *coil*)
Clear hw rule.

configure_accelerometer(*number*, *config*, *callback*)
Configure accelerometer.

configure_dmd()
Configure DMD.

configure_driver(*config: mpf.core.platform.DriverConfig*, *number: str*, *platform_settings: dict*)
Configure driver.

configure_hardware_sound_system() → mpf.platforms.interfaces.hardware_sound_platform_interface.HardwareSou
Configure virtual hardware sound system.

configure_i2c(*number: str*) → I2cPlatformInterface
Configure virtual i2c device.

configure_light(*number*, *subtype*, *platform_settings*)
Configure light channel.

configure_rgb_dmd(*name: str*)
Configure DMD.

configure_segment_display(*number: str*) → mpf.platforms.interfaces.segment_display_platform_interface.SegmentDi
Configure segment display.

configure_servo(*number: str*)
Configure a servo device in platform.

configure_stepper(*number: str*, *config: dict*)
Configure a smart stepper / axis device in platform.

configure_switch(*number: str*, *config: mpf.core.platform.SwitchConfig*, *platform_config: dict*)
Configure switch.

get_hw_switch_states()
Return hw switch states.

initialize() → None
Initialise platform.

parse_light_number_to_channels(*number: str*, *subtype: str*)
Parse channel str to a list of channels.

set_pulse_on_hit_and_enable_and_release_and_disable_rule(*enable_switch*, *disable_switch*, *coil*)
Set rule.

set_pulse_on_hit_and_enable_and_release_rule(*enable_switch*, *coil*)
Set rule.

set_pulse_on_hit_and_release_rule(*enable_switch*, *coil*)
Set rule.

set_pulse_on_hit_rule(*enable_switch*, *coil*)
Set rule.

stop()
Stop platform.

validate_coil_section(*driver*, *config*)
Validate coil sections.

validate_stepper_section(*stepper*, *config*)
Validate stepper sections.

validate_switch_section(*switch*, *config*)
Validate switch sections.

self.machine.hardware_platforms['virtual_pinball']

class mpf.platforms.virtual_pinball.virtual_pinball.**VirtualPinballPlatform**(*machine*)
Bases: mpf.core.platform.LightsPlatform, mpf.core.platform.SwitchPlatform, mpf.core.platform.DriverPlatform

VPX platform.

Accessing the virtual_pinball platform via code

Hardware platforms are stored in the `self.machine.hardware_platforms` dictionary, so the virtual_pinball platform is available via `self.machine.hardware_platforms['virtual_pinball']`.

Methods & Attributes

The virtual_pinball platform has the following methods & attributes available. Note that methods & attributes inherited from base classes are not included here.

clear_hw_rule(*switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
Clear a hw rule.

configure_driver(*config: mpf.core.platform.DriverConfig*, *number: str*, *platform_settings: dict*) → mpf.platforms.interfaces.driver_platform_interface.DriverPlatformInterface
Configure VPX driver.

configure_light(*number: str*, *subtype: str*, *platform_settings: dict*) → mpf.platforms.interfaces.light_platform_interface.LightPlatformInterface
Configure a VPX light.

configure_switch(*number: str*, *config: mpf.core.platform.SwitchConfig*, *platform_config: dict*) → mpf.platforms.interfaces.switch_platform_interface.SwitchPlatformInterface
Configure VPX switch.

get_hw_switch_states()
Return initial switch state.

initialize()
Initialise platform.

parse_light_number_to_channels(*number: str*, *subtype: str*)
Parse channel str to a list of channels.

set_pulse_on_hit_and_enable_and_release_and_disable_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *disable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
Write rule.

set_pulse_on_hit_and_enable_and_release_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
Write rule.

set_pulse_on_hit_and_release_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
Write rule.

set_pulse_on_hit_rule(*enable_switch: mpf.core.platform.SwitchSettings*, *coil: mpf.core.platform.DriverSettings*)
Write rule.

vpx_changed_gi_strings()
Return changed lamps since last call.

vpx_changed_lamps()
Return changed lamps since last call.

vpx_changed_solenoids()
Return changed solenoids since last call.

vpx_get_mech(*number*)
Not implemented.

vpx_get_switch(*number*)
Return switch value.

vpx_mech(*number*)
Not implemented.

vpx_set_mech (*number*, *value*)
Not implemented.

vpx_set_switch (*number*, *value*)
Update switch from VPX.

vpx_start ()
Start machine.

vpx_switch (*number*)
Return switch value.

8.3.5 Config Players

Config players are available as machine attributes in the form of their player name plus `_player`, for example, `self.machine.light_player` or `self.machine.score_player`.

self.machine.blocking_player

class mpf.config_players.block_event_player.**BlockEventPlayer** (*machine*)
Bases: mpf.core.config_player.ConfigPlayer

Posts events based on config.

Accessing the blocking_player in code

The blocking_player is available via `self.machine.blocking_player`.

Methods & Attributes

The blocking_player has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

get_express_config (*value*)
Parse short config.

play (*settings*, *context*, *calling_context*, *priority=0*, ***kwargs*)
Block event.

validate_config_entry (*settings: dict*, *name: str*) → dict
Validate one entry of this player.

self.machine.coil_player

class mpf.config_players.coil_player.**CoilPlayer** (*machine*)
Bases: mpf.config_players.device_config_player.DeviceConfigPlayer

Triggers coils based on config.

Accessing the coil_player in code

The coil_player is available via `self.machine.coil_player`.

Methods & Attributes

The coil_player has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

clear_context(*context*)
Disable enabled coils.

get_express_config(*value: str*)
Parse short config version.

play(*settings*, *context: str*, *calling_context: str*, *priority: int = 0*, ***kwargs*)
Enable, Pulse or disable coils.

self.machine.event_player

class mpf.config_players.event_player.**EventPlayer**(*machine*)
Bases: mpf.config_players.flat_config_player.FlatConfigPlayer

Posts events based on config.

Accessing the event_player in code

The event_player is available via `self.machine.event_player`.

Methods & Attributes

The event_player has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

get_express_config(*value*)
Parse short config.

get_list_config(*value*)
Parse list.

play(*settings*, *context*, *calling_context*, *priority=0*, ***kwargs*)
Post (delayed) events.

self.machine.flasher_player

class mpf.config_players.flasher_player.**FlasherPlayer**(*machine*)
Bases: mpf.config_players.device_config_player.DeviceConfigPlayer

Triggers flashers based on config.

Accessing the flasher_player in code

The flasher_player is available via `self.machine.flasher_player`.

Methods & Attributes

The flasher_player has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

get_express_config(*value*)
Parse express config.

play(*settings*, *context*, *calling_context*, *priority=0*, ***kwargs*)
Flash flashers.

self.machine.hardware_sound_player_player

class mpf.config_players.hardware_sound_player.**HardwareSoundPlayer**(*machine*)
Bases: mpf.config_players.device_config_player.DeviceConfigPlayer

Plays sounds on an external sound card.

Accessing the hardware_sound_player_player in code

The hardware_sound_player_player is available via `self.machine.hardware_sound_player_player`.

Methods & Attributes

The hardware_sound_player_player has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

get_express_config(*value*)
Parse express config.

get_string_config(*string*)
Parse string config.

play(*settings*, *context*, *calling_context*, *priority=0*, ***kwargs*)
Play sound on external card.

self.machine.light_player

class mpf.config_players.light_player.**LightPlayer**(*machine*)
Bases: mpf.config_players.device_config_player.DeviceConfigPlayer

Sets lights based on config.

Accessing the light_player in code

The light_player is available via `self.machine.light_player`.

Methods & Attributes

The light_player has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

clear_context(*context*)
Remove all colors which were set in context.

get_express_config(*value*)
Parse express config.

handle_subscription_change(*value*, *settings*, *priority*, *context*)
Handle subscriptions.

play(*settings*, *context*, *calling_context*, *priority=0*, ***kwargs*)
Set light color based on config.

self.machine.queue_event_player

class mpf.config_players.queue_event_player.**QueueEventPlayer**(*machine*)
Bases: mpf.core.config_player.ConfigPlayer

Posts queue events based on config.

Accessing the queue_event_player in code

The queue_event_player is available via `self.machine.queue_event_player`.

Methods & Attributes

The queue_event_player has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

get_express_config(*value*)
No express config.

play(*settings*, *context*, *calling_context*, *priority=0*, ***kwargs*)
Post queue events.

validate_config_entry(*settings*, *name*)
Validate one entry of this player.

self.machine.queue_relay_player

class mpf.config_players.queue_relay_player.**QueueRelayPlayer**(*machine*)
Bases: mpf.core.config_player.ConfigPlayer

Blocks queue events and converts them to normal events.

Accessing the queue_relay_player in code

The queue_relay_player is available via `self.machine.queue_relay_player`.

Methods & Attributes

The queue_relay_player has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

clear_context(*context*)
Clear all queues and remove handlers.

get_express_config(*value*)
No express config.

play(*settings*, *context*, *calling_context*, *priority=0*, ***kwargs*)
Block queue event.

validate_config_entry(*settings*, *name*)
Validate one entry of this player.

self.machine.random_event_player

class mpf.config_players.random_event_player.**RandomEventPlayer**(*machine*)
Bases: mpf.core.config_player.ConfigPlayer

Plays a random event based on config.

Accessing the random_event_player in code

The random_event_player is available via `self.machine.random_event_player`.

Methods & Attributes

The random_event_player has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

get_express_config(*value*)
Parse express config.

get_list_config(*value*)
Parse list.

static is_entry_valid_outside_mode(*settings*) → bool
Return true if scope is not player.

play(*settings*, *context*, *calling_context*, *priority=0*, ***kwargs*)
Play a random event from list based on config.

validate_config_entry(*settings*, *name*)
Validate one entry of this player.

self.machine.segment_display_player_player

class mpf.config_players.segment_display_player.**SegmentDisplayPlayer**(*machine*)
Bases: mpf.config_players.device_config_player.DeviceConfigPlayer

Generates texts on segment displays.

Accessing the segment_display_player_player in code

The segment_display_player_player is available via `self.machine.segment_display_player_player`.

Methods & Attributes

The segment_display_player_player has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

clear_context(*context*)
Remove all texts.

get_express_config(*value*)
Parse express config.

play(*settings*, *context*, *calling_context*, *priority=0*, ***kwargs*)
Show text on display.

self.machine.show_player

class mpf.config_players.show_player.**ShowPlayer**(*machine*)
Bases: mpf.config_players.device_config_player.DeviceConfigPlayer

Plays, starts, stops, pauses, resumes or advances shows based on config.

Accessing the show_player in code

The show_player is available via `self.machine.show_player`.

Methods & Attributes

The show_player has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

clear_context(*context*)
Stop running shows from context.

get_express_config(*value*)
Parse express config.

handle_subscription_change(*value*, *settings*, *priority*, *context*)
Handle subscriptions.

play(*settings*, *context*, *calling_context*, *priority=0*, ***kwargs*)
Play, start, stop, pause, resume or advance show based on config.

self.machine.variable_player

class mpf.config_players.variable_player.**VariablePlayer**(*machine: mpf.core.machine.MachineController*)

Bases: mpf.core.config_player.ConfigPlayer

Posts events based on config.

Accessing the variable_player in code

The variable_player is available via `self.machine.variable_player`.

Methods & Attributes

The variable_player has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

clear_context(*context: str*) → None
Clear context.

get_express_config(*value: Any*) → dict
Parse express config.

get_list_config(*value: Any*)
Parse list.

static is_entry_valid_outside_mode(*settings: dict*) → bool
Return true if this entry may run without a game and player.

play(*settings: dict*, *context: str*, *calling_context: str*, *priority: int = 0*, ***kwargs*) → None
Variable name.

validate_config_entry(*settings: dict*, *name: str*) → dict
Validate one entry of this player.

8.3.6 Testing Class API

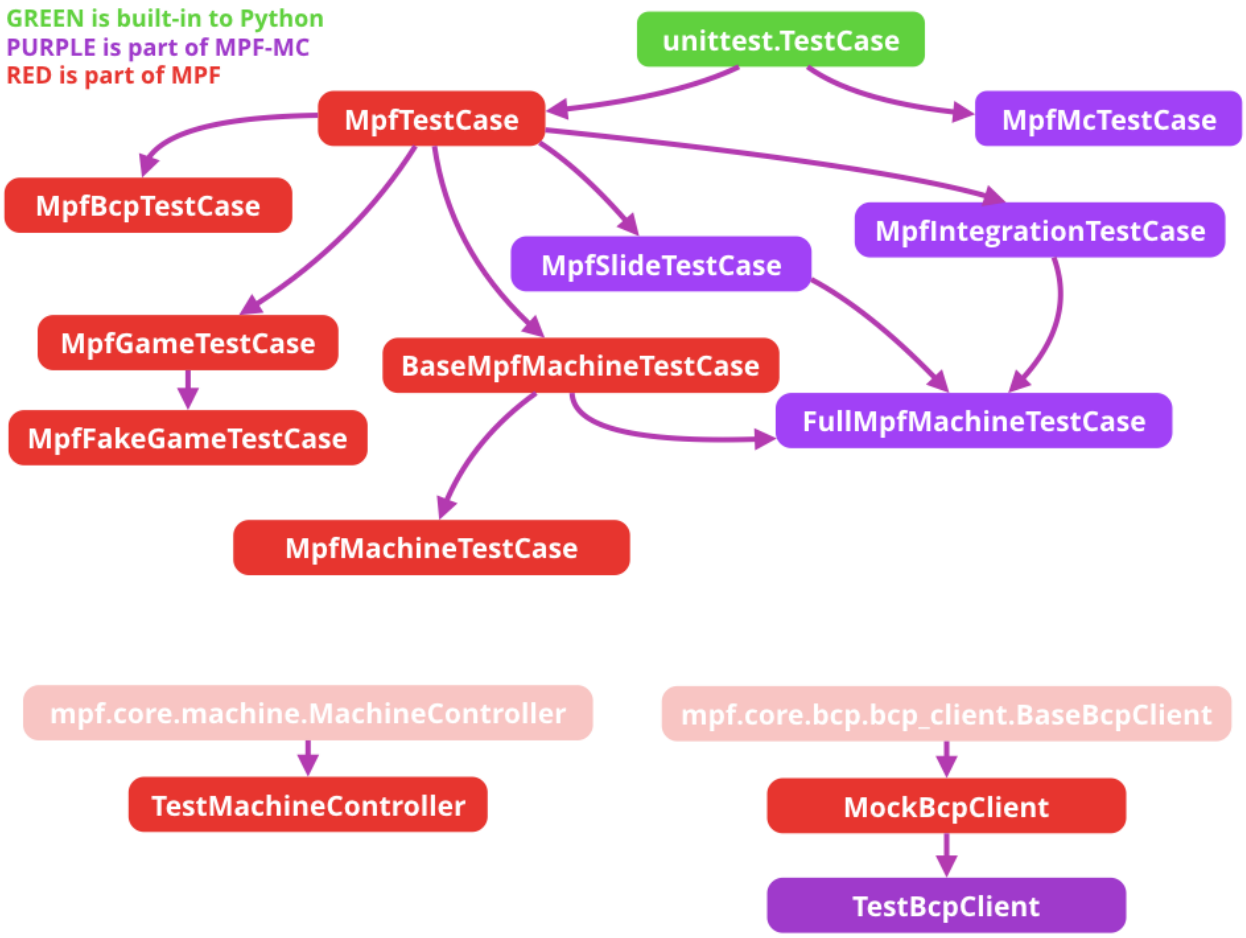
MPF includes several unit test classes which you can use to *write tests which test MPF* or to write tests for your own game.

These tests include several MPF-specific assertion methods for things like modes, players, balls, device states, etc., as well as logic which advances the time and mocks the BCP and hardware connections.

You can add commands in your tests to "advance" the time which the MPF tests can test quickly, so you can test a complete 3-minute game play session in a few hundred milliseconds of real world time.

It might be helpful to look at the real internal tests that MPF uses (which all use these test classes) to get a feel for how tests are written in MPF. They're available in the mpf/tests folder in the MPF repository. (They're installed locally when you install MPF.)

Here's a diagram which shows how all the MPF and MPF-MC test case classes relate to each other:



And the API reference for each:

MockBcpClient

class mpf.tests.MpfBcpTestCase.**MockBcpClient**(*machine*, *name*, *bcp*)

Bases: mpf.core.bcp.bcp_client.BaseBcpClient

A Mock BCP Client.

This is used in tests require BCP for testing but where you don't actually create a real BCP connection.

Methods & Attributes

The MockBcpClient has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

accept_connection(*receiver*, *sender*)

Handle incoming connection from remote client.

connect(*config*)

Actively connect client.

ignorable_runtime_exception(*msg: str*) → None
Handle ignorable runtime exception.

During development or tests raise an exception for easier debugging. Log an error during production.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

read_message()
Read one message from client.

send(*bcp_command*, *bcp_command_args*)
Send data to client.

stop()
Stop client connection.

warning_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the warning level.

These messages will always be shown in the console and the log file.

MpfBcpTestCase

class mpf.tests.MpfBcpTestCase.**MpfBcpTestCase**(*methodName='runTest'*)
Bases: *mpf.tests.MpfTestCase.MpfTestCase*

An MpfTestCase instance which uses the MockBcpClient.

Methods & Attributes

The MpfBcpTestCase has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

static add_to_config_validator(*machine*, *key*, *new_dict*)
Add config dict to validator.

advance_time_and_run(*delta=1.0*)
Advance the test clock and run anything that should run during that time.

Parameters **delta** – How much time to advance the test clock by (in seconds)

This method will cause anything scheduled during the time to run, including things like delays, timers, etc.

Advancing the clock will happen in multiple small steps if things are scheduled to happen during this advance. For example, you can advance the clock 10 seconds like this:

```
self.advance_time_and_run(10)
```

If there is a delay callback that is scheduled to happen in 2 seconds, then this method will advance the clock 2 seconds, process that delay, and then advance the remaining 8 seconds.

assertAlmostEqual(*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Fail if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is more than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

If the two objects compare equal then they will automatically compare almost equal.

assertColorAlmostEqual(*color1*, *color2*, *delta=6*)
Assert that two color are almost equal.

Parameters

- **color1** – The first color, as an RGBColor instance or 3-item iterable.
- **color2** – The second color, as an RGBColor instance or 3-item iterable.
- **delta** – How close the colors have to be. The deltas between red, green, and blue are added together and must be less or equal to this value for this assertion to succeed.

assertCountEqual(*first*, *second*, *msg=None*)
An unordered sequence comparison asserting that the same elements, regardless of order. If the same element occurs more than once, it verifies that the elements occur the same number of times.

self.assertEqual(Counter(list(first)), Counter(list(second)))

Example:

- [0, 1, 1] and [1, 0, 1] compare equal.
- [0, 0, 1] and [0, 1] compare unequal.

assertDictContainsSubset(*subset*, *dictionary*, *msg=None*)
Checks whether dictionary is a superset of subset.

assertEqual(*first*, *second*, *msg=None*)
Fail if the two objects are unequal as determined by the '==' operator.

assertEventCalled(*event_name*, *times=None*)
Assert that event was called.

Parameters

- **event_name** – String name of the event to check.
- **times** – An optional value to confirm the number of times the event was called. Default of *None* means this method will pass as long as the event has been called at least once.

If you want to reset the `times` count, you can mock the event again.

Note that the event must be mocked via `self.mock_event()` first in order to use this method.

For example:

```
self.mock_event('my_event')
self.assertEventNotCalled('my_event')  # This will pass

self.post_event('my_event')
self.assertEventCalled('my_event')     # This will pass
self.assertEventCalled('my_event', 1)  # This will pass

self.post_event('my_event')
self.assertEventCalled('my_event')     # This will pass
self.assertEventCalled('my_event', 2)  # This will pass
```

assertEventCalledWith(*event_name*, ***kwargs*)
Assert an event was called with certain kwargs.

Parameters

- **event_name** – String name of the event to check.
- ****kwargs** – Name/value parameters to check.

For example:

```
self.mock_event('jackpot')

self.post_event('jackpot', count=1, first_time=True)
self.assertEventCalled('jackpot')  # This will pass
self.assertEventCalledWith('jackpot', count=1, first_time=True)  # This will
↪also pass
self.assertEventCalledWith('jackpot', count=1, first_time=False)  # This will
↪fail
```

assertEventNotCalled(*event_name*)
Assert that event was not called.

Parameters **event_name** – String name of the event to check.

Note that the event must be mocked via `self.mock_event()` first in order to use this method.

assertFalse(*expr*, *msg=None*)
Check that the expression is false.

assertGreater(*a*, *b*, *msg=None*)
Just like self.assertTrue(a > b), but with a nicer default message.

assertGreaterEqual(*a*, *b*, *msg=None*)
Just like self.assertTrue(a >= b), but with a nicer default message.

assertIn(*member*, *container*, *msg=None*)
Just like self.assertTrue(a in b), but with a nicer default message.

assertIs(*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is b), but with a nicer default message.

assertIsInstance(*obj*, *cls*, *msg=None*)
Same as self.assertTrue(isinstance(obj, cls)), with a nicer default message.

assertIsNone(*obj*, *msg=None*)
Same as self.assertTrue(obj is None), with a nicer default message.

assertIsNot(*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is not b), but with a nicer default message.

assertIsNotNone(*obj*, *msg=None*)
Included for symmetry with assertIsNone.

assertLess(*a*, *b*, *msg=None*)
Just like self.assertTrue(a < b), but with a nicer default message.

assertLessEqual(*a*, *b*, *msg=None*)
Just like self.assertTrue(a <= b), but with a nicer default message.

assertListEqual(*list1*, *list2*, *msg=None*)
A list-specific equality assertion.

Parameters

- **list1** – The first list to compare.
- **list2** – The second list to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertLogs(*logger=None*, *level=None*)
Fail unless a log message of level *level* or higher is emitted on *logger_name* or its children. If omitted, *level* defaults to INFO and *logger* defaults to the root logger.

This method must be used as a context manager, and will yield a recording object with two attributes: *output* and *records*. At the end of the context manager, the *output* attribute will be a list of the matching formatted log messages and the *records* attribute will be a list of the corresponding LogRecord objects.

Example:

```
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

assertMultiLineEqual(*first*, *second*, *msg=None*)
Assert that two multi-line strings are equal.

assertNotAlmostEqual(*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Fail if the two objects are equal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is less than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

Objects that are equal automatically fail.

assertNotEqual(*first*, *second*, *msg=None*)
Fail if the two objects are equal as determined by the '!=' operator.

assertNotIn(*member*, *container*, *msg=None*)
Just like self.assertTrue(a not in b), but with a nicer default message.

assertNotIsInstance(*obj*, *cls*, *msg=None*)
Included for symmetry with assertIsInstance.

assertNotRegex(*text*, *unexpected_regex*, *msg=None*)
Fail the test if the text matches the regular expression.

assertNumBallsKnown(*balls*)
Assert that a certain number of balls are known in the machine.

assertRaises(*expected_exception*, **args*, ***kwargs*)
Fail unless an exception of class expected_exception is raised by the callable when invoked with specified positional and keyword arguments. If a different type of exception is raised, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.

If called with the callable and arguments omitted, will return a context object used like this:

```
with self.assertRaises(SomeException):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertRaises is used as a context object.

The context manager keeps a reference to the exception as the 'exception' attribute. This allows you to inspect the exception after the assertion:

```
with self.assertRaises(SomeException) as cm:
    do_something()
```

(continues on next page)

(continued from previous page)

```
the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

assertRaisesRegex(*expected_exception*, *expected_regex*, **args*, ***kwargs*)
Asserts that the message in a raised exception matches a regex.

Parameters

- **expected_exception** – Exception class expected to be raised.
- **expected_regex** – Regex (re pattern object or string) expected to be found in error message.
- **args** – Function to be called and extra positional args.
- **kwargs** – Extra kwargs.
- **msg** – Optional message used in case of failure. Can only be used when assertRaisesRegex is used as a context manager.

assertRegex(*text*, *expected_regex*, *msg=None*)
Fail the test unless the text matches the regular expression.

assertSequenceEqual(*seq1*, *seq2*, *msg=None*, *seq_type=None*)
An equality assertion for ordered sequences (like lists and tuples).

For the purposes of this function, a valid ordered sequence type is one which can be indexed, has a length, and has an equality operator.

Parameters

- **seq1** – The first sequence to compare.
- **seq2** – The second sequence to compare.
- **seq_type** – The expected datatype of the sequences, or None if no datatype should be enforced.
- **msg** – Optional message to use on failure instead of a list of differences.

assertSetEqual(*set1*, *set2*, *msg=None*)
A set-specific equality assertion.

Parameters

- **set1** – The first set to compare.
- **set2** – The second set to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertSetEqual uses ducktyping to support different types of sets, and is optimized for sets specifically (parameters must support a difference method).

assertTrue(*expr*, *msg=None*)
Check that the expression is true.

assertTupleEqual(*tuple1*, *tuple2*, *msg=None*)
A tuple-specific equality assertion.

Parameters

- **tuple1** – The first tuple to compare.
- **tuple2** – The second tuple to compare.

- **msg** – Optional message to use on failure instead of a list of differences.

assertWarns(*expected_warning*, **args*, ***kwargs*)

Fail unless a warning of class warnClass is triggered by the callable when invoked with specified positional and keyword arguments. If a different type of warning is triggered, it will not be handled: depending on the other warning filtering rules in effect, it might be silenced, printed out, or raised as an exception.

If called with the callable and arguments omitted, will return a context object used like this:

```
with self.assertWarns(SomeWarning):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertWarns is used as a context object.

The context manager keeps a reference to the first matching warning as the 'warning' attribute; similarly, the 'filename' and 'lineno' attributes give you information about the line of Python code from which the warning was triggered. This allows you to inspect the warning after the assertion:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()
the_warning = cm.warning
self.assertEqual(the_warning.some_attribute, 147)
```

assertWarnsRegex(*expected_warning*, *expected_regex*, **args*, ***kwargs*)

Asserts that the message in a triggered warning matches a regexp. Basic functioning is similar to assertWarns() with the addition that only warnings whose messages also match the regular expression are considered successful matches.

Parameters

- **expected_warning** – Warning class expected to be triggered.
- **expected_regex** – Regex (re pattern object or string) expected to be found in error message.
- **args** – Function to be called and extra positional args.
- **kwargs** – Extra kwargs.
- **msg** – Optional message used in case of failure. Can only be used when assertWarnsRegex is used as a context manager.

fail(*msg=None*)

Fail immediately, with the given message.

getConfigFile()

Return a string name of the machine config file to use for the tests in this class.

You should override this method in your own test class to point to the config file you need for your tests.

Returns A string name of the machine config file to use, complete with the `.yaml` file extension.

For example:

```
def getConfigFile(self):
    return 'my_config.yaml'
```

getMachinePath()

Return a string name of the path to the machine folder to use for the tests in this class.

You should override this method in your own test class to point to the machine folder root you need for your tests.

Returns A string name of the machine path to use

For example:

```
def getMachinePath(self):
    return 'tests/machine_files/my_test/'
```

Note that this path is relative to the MPF package root

get_enable_plugins()
Control whether tests in this class should load MPF plugins.

Returns: True or False

The default is False. To load plugins in your test class, add the following:

```
def get_enable_plugins(self):
    return True
```

get_platform()
Force this test class to use a certain platform.

Returns String name of the platform this test class will use.

If you don't include this method in your test class, the platform will be set to *virtual*. If you want to use the smart virtual platform, you would add the following to your test class:

```
def get_platform(self):
    return 'smart_virtual`
```

get_use_bcp()
Control whether tests in this class should use BCP.

Returns: True or False

The default is False. To use BCP in your test class, add the following:

```
def get_use_bcp(self):
    return True
```

hit_and_release_switch(*name*)
Momentarily activates and then deactivates a switch.

Parameters **name** – The name of the switch to hit.

This method immediately activates and deactivates a switch with no time in between.

hit_and_release_switches_simultaneously(*names*)
Momentarily activates and then deactivates multiple switches.

Switches are hit sequentially and then released sequentially. Events are only processed at the end of the sequence which is useful to reproduce race conditions when processing nearly simultaneous hits.

Parameters **names** – The names of the switches to hit and release.

hit_switch_and_run(*name*, *delta*)
Activates a switch and advances the time.

Parameters

- **name** – The name of the switch to activate.
- **delta** – The time (in seconds) to advance the clock.

Note that this method does not deactivate the switch once the time has been advanced, meaning the switch stays active. To make the switch inactive, use the *release_switch_and_run()*.

machine_run()
Process any delays, timers, or anything else scheduled.

Note this is the same as:

```
self.advance_time_and_run(0)
```

mock_event(*event_name*)
Configure an event to be mocked.

Parameters **event_name** – String name of the event to mock.

Mocking an event is an easy way to check if an event was called without configuring some kind of callback action in your tests.

Note that an event must be mocked here *before* it's posted in order for *assertEventNotCalled()* and *assertEventCalled()* to work.

Mocking an event will not "break" it. In other words, any other registered handlers for this event will also be called even if the event has been mocked.

For example:

```
self.mock_event('my_event')
self.assertEventNotCalled('my_event')  # This will be True
self.post_event('my_event')
self.assertEventCalled('my_event')  # This will also be True
```

post_event(*event_name*, *run_time=0*)
Post an MPF event and optionally advance the time.

Parameters

- **event_name** – String name of the event to post
- **run_time** – How much time (in seconds) the test should advance after this event has been posted.

For example, to post an event called "shot1_hit":

```
self.post_event('shot1_hit')
```

To post an event called "tilt" and then advance the time 1.5 seconds:

```
self.post_event('tilt', 1.5)
```

post_event_with_params(*event_name*, ***params*)
Post an MPF event with kwarg parameters.

Parameters

- **event_name** – String name of the event to post
- ****params** – One or more kwarg key/value pairs to post with the event.

For example, to post an event called "jackpot" with the parameters `count=1` and `first_time=True`, you would use:

```
self.post_event('jackpot', count=1, first_time=True)
```

release_switch_and_run(*name*, *delta*)
Deactivates a switch and advances the time.

Parameters

- **name** – The name of the switch to activate.
- **delta** – The time (in seconds) to advance the clock.

reset_mock_events()
Reset all mocked events.

This will reset the count of number of times called every mocked event is.

setUp()
Hook method for setting up the test fixture before exercising it.

set_num_balls_known(*balls*)
Set the ball controller's `num_balls_known` attribute.

This is needed for tests where you don't have any ball devices and other situations where you need the ball controller to think the machine has a certain amount of balls to run a test.

Example:

```
self.set_num_balls_known(3)
```

shortDescription()
Returns a one-line description of the test, or None if no description has been provided.

The default implementation of this method returns the first line of the specified test method's docstring.

skipTest(*reason*)
Skip this test.

start_mode(*mode*)
Start mode.

stop_mode(*mode*)
Stop mode.

tearDown()
Hook method for deconstructing the test fixture after testing it.

static unittest_verbosity()
Return the verbosity setting of the currently running unittest program, or 0 if none is running.

Returns: An integer value of the current verbosity setting.

MpfFakeGameTestCase

class mpf.tests.MpfFakeGameTestCase.**MpfFakeGameTestCase**(*methodName*)
Bases: *mpf.tests.MpfGameTestCase.MpfGameTestCase*

Test case for a game that does not require ball devices & start switches.

Often times you need to write a test that is able to start a game. However in order to start a game, MPF requires lots of things, like having proper ball devices and a start button and stuff like that.

This test overwrites the `start_game()` and `drain_ball()` methods of the `MpfGameTestCase` class so that you can start games and drain balls without actually having any ball devices configured.

Methods & Attributes

The MpfFakeGameTestCase has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

add_player()
Add a player to the current game.

This method hits and releases a switch called `s_start` and then verifies that the player count actually increased by 1.

You can call this method multiple times to add multiple players. For example, to start a game and then add 2 additional players (for 3 players total), you would use:

```
self.start_game()
self.add_player()
self.add_player()
```

static add_to_config_validator(*machine*, *key*, *new_dict*)
Add config dict to validator.

advance_time_and_run(*delta=1.0*)
Advance the test clock and run anything that should run during that time.

Parameters **delta** – How much time to advance the test clock by (in seconds)

This method will cause anything scheduled during the time to run, including things like delays, timers, etc.

Advancing the clock will happen in multiple small steps if things are scheduled to happen during this advance. For example, you can advance the clock 10 seconds like this:

```
self.advance_time_and_run(10)
```

If there is a delay callback that is scheduled to happen in 2 seconds, then this method will advance the clock 2 seconds, process that delay, and then advance the remaining 8 seconds.

assertAlmostEqual(*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Fail if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is more than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

If the two objects compare equal then they will automatically compare almost equal.

assertBallNumber(*number*)
Asserts that the current ball is a certain ball numebr.

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

Raises

- **Assertion error if there is no game in progress or if** –
- **the current ball is not the ball number passed.** –

The following code will check to make sure the game is on Ball 1:

```
self.assertBallNumber(1)
```

assertBallsInPlay(*balls*)
Asserts that a certain number of balls are in play.

Note that the number of balls in play is not necessarily the same as the number of balls on the playfield. (For example, a ball could be held in a ball device, or the machine could be in the process of adding a ball to the platfield.)

Parameters **balls** – The number of balls you want to assert are in play.

To assert that there are 3 balls in play (perhaps during a multiball), you would use:

```
self.assertBallsInPlay(3)
```

assertColorAlmostEqual(*color1*, *color2*, *delta=6*)
Assert that two color are almost equal.

Parameters

- **color1** – The first color, as an RGBColor instance or 3-item iterable.
- **color2** – The second color, as an RGBColor instance or 3-item iterable.
- **delta** – How close the colors have to be. The deltas between red, green, and blue are added together and must be less or equal to this value for this assertion to succeed.

assertCountEqual(*first*, *second*, *msg=None*)
An unordered sequence comparison asserting that the same elements, regardless of order. If the same element occurs more than once, it verifies that the elements occur the same number of times.

self.assertEqual(Counter(list(first)), Counter(list(second)))

Example:

- [0, 1, 1] and [1, 0, 1] compare equal.
- [0, 0, 1] and [0, 1] compare unequal.

assertDictContainsSubset(*subset*, *dictionary*, *msg=None*)
Checks whether dictionary is a superset of subset.

assertEqual(*first*, *second*, *msg=None*)
Fail if the two objects are unequal as determined by the '==' operator.

assertEventCalled(*event_name*, *times=None*)
Assert that event was called.

Parameters

- **event_name** – String name of the event to check.
- **times** – An optional value to confirm the number of times the event was called. Default of *None* means this method will pass as long as the event has been called at least once.

If you want to reset the `times` count, you can mock the event again.

Note that the event must be mocked via `self.mock_event()` first in order to use this method.

For example:

```
self.mock_event('my_event')
self.assertEventNotCalled('my_event')  # This will pass

self.post_event('my_event')
self.assertEventCalled('my_event')     # This will pass
self.assertEventCalled('my_event', 1)  # This will pass
```

(continues on next page)

(continued from previous page)

```
self.post_event('my_event')
self.assertEventCalled('my_event')      # This will pass
self.assertEventCalled('my_event', 2)   # This will pass
```

assertEventCalledWith(*event_name*, ***kwargs*)
Assert an event was called with certain kwargs.

Parameters

- **event_name** – String name of the event to check.
- ****kwargs** – Name/value parameters to check.

For example:

```
self.mock_event('jackpot')

self.post_event('jackpot', count=1, first_time=True)
self.assertEventCalled('jackpot')  # This will pass
self.assertEventCalledWith('jackpot', count=1, first_time=True)  # This will␣
↪also pass
self.assertEventCalledWith('jackpot', count=1, first_time=False)  # This will␣
↪fail
```

assertEventNotCalled(*event_name*)
Assert that event was not called.

Parameters **event_name** – String name of the event to check.

Note that the event must be mocked via `self.mock_event()` first in order to use this method.

assertFalse(*expr*, *msg=None*)
Check that the expression is false.

assertGameIsNotRunning()
Assert a game is not running.

Example:

```
self.assertGameIsNotRunning()
```

assertGameIsRunning()
Assert a game is running.

Example:

```
self.assertGameIsRunning()
```

assertGreater(*a*, *b*, *msg=None*)
Just like self.assertTrue(a > b), but with a nicer default message.

assertGreaterEqual(*a*, *b*, *msg=None*)
Just like self.assertTrue(a >= b), but with a nicer default message.

assertIn(*member*, *container*, *msg=None*)
Just like self.assertTrue(a in b), but with a nicer default message.

assertIs(*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is b), but with a nicer default message.

assertIsInstance(*obj*, *cls*, *msg=None*)
Same as self.assertTrue(isinstance(obj, cls)), with a nicer default message.

assertIsNone(*obj*, *msg=None*)
Same as self.assertTrue(obj is None), with a nicer default message.

assertIsNot(*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is not b), but with a nicer default message.

assertIsNotNone(*obj*, *msg=None*)
Included for symmetry with assertIsNone.

assertLess(*a*, *b*, *msg=None*)
Just like self.assertTrue(a < b), but with a nicer default message.

assertLessEqual(*a*, *b*, *msg=None*)
Just like self.assertTrue(a <= b), but with a nicer default message.

assertListEqual(*list1*, *list2*, *msg=None*)
A list-specific equality assertion.

Parameters

- **list1** – The first list to compare.
- **list2** – The second list to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertLogs(*logger=None*, *level=None*)
Fail unless a log message of level *level* or higher is emitted on *logger_name* or its children. If omitted, *level* defaults to INFO and *logger* defaults to the root logger.

This method must be used as a context manager, and will yield a recording object with two attributes: *output* and *records*. At the end of the context manager, the *output* attribute will be a list of the matching formatted log messages and the *records* attribute will be a list of the corresponding LogRecord objects.

Example:

```
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

assertMultiLineEqual(*first*, *second*, *msg=None*)
Assert that two multi-line strings are equal.

assertNotAlmostEqual(*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Fail if the two objects are equal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is less than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

Objects that are equal automatically fail.

assertNotEqual(*first*, *second*, *msg=None*)
Fail if the two objects are equal as determined by the '!=' operator.

assertNotIn(*member*, *container*, *msg=None*)
Just like self.assertTrue(a not in b), but with a nicer default message.

assertNotIsInstance(*obj*, *cls*, *msg=None*)
Included for symmetry with assertIsInstance.

assertNotRegex(*text*, *unexpected_regex*, *msg=None*)
Fail the test if the text matches the regular expression.

assertNumBallsKnown(*balls*)
Assert that a certain number of balls are known in the machine.

assertPlayerCount(*count*)
Asserts that count players exist.

Parameters **count** – The expected number of players.

For example, to assert that the to players are in the game:

```
self.assertPlayerCount(2)
```

assertPlayerNumber(*number*)
Asserts that the current player is a certain player number.

Parameters **number** – The player number you can to assert is the current player.

For example, to assert that the current player is Player 2, you would use:

```
self.assertPlayerNumber(2)
```

assertRaises(*expected_exception*, **args*, ***kwargs*)
Fail unless an exception of class expected_exception is raised by the callable when invoked with specified positional and keyword arguments. If a different type of exception is raised, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.

If called with the callable and arguments omitted, will return a context object used like this:

```
with self.assertRaises(SomeException):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertRaises is used as a context object.

The context manager keeps a reference to the exception as the 'exception' attribute. This allows you to inspect the exception after the assertion:

```
with self.assertRaises(SomeException) as cm:
    do_something()
the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

assertRaisesRegex(*expected_exception*, *expected_regex*, **args*, ***kwargs*)
Asserts that the message in a raised exception matches a regex.

Parameters

- **expected_exception** – Exception class expected to be raised.
- **expected_regex** – Regex (re pattern object or string) expected to be found in error message.
- **args** – Function to be called and extra positional args.
- **kwargs** – Extra kwargs.
- **msg** – Optional message used in case of failure. Can only be used when assertRaisesRegex is used as a context manager.

assertRegex(*text*, *expected_regex*, *msg=None*)
Fail the test unless the text matches the regular expression.

assertSequenceEqual(*seq1*, *seq2*, *msg=None*, *seq_type=None*)
An equality assertion for ordered sequences (like lists and tuples).

For the purposes of this function, a valid ordered sequence type is one which can be indexed, has a length, and has an equality operator.

Parameters

- **seq1** – The first sequence to compare.
- **seq2** – The second sequence to compare.
- **seq_type** – The expected datatype of the sequences, or None if no datatype should be enforced.
- **msg** – Optional message to use on failure instead of a list of differences.

assertSetEqual(*set1*, *set2*, *msg=None*)
A set-specific equality assertion.

Parameters

- **set1** – The first set to compare.
- **set2** – The second set to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertSetEqual uses ducktyping to support different types of sets, and is optimized for sets specifically (parameters must support a difference method).

assertTrue(*expr*, *msg=None*)
Check that the expression is true.

assertTupleEqual(*tuple1*, *tuple2*, *msg=None*)
A tuple-specific equality assertion.

Parameters

- **tuple1** – The first tuple to compare.
- **tuple2** – The second tuple to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertWarns(*expected_warning*, **args*, ***kwargs*)
Fail unless a warning of class warnClass is triggered by the callable when invoked with specified positional and keyword arguments. If a different type of warning is triggered, it will not be handled: depending on the other warning filtering rules in effect, it might be silenced, printed out, or raised as an exception.

If called with the callable and arguments omitted, will return a context object used like this:

```
with self.assertWarns(SomeWarning):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertWarns is used as a context object.

The context manager keeps a reference to the first matching warning as the 'warning' attribute; similarly, the 'filename' and 'lineno' attributes give you information about the line of Python code from which the warning was triggered. This allows you to inspect the warning after the assertion:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()
the_warning = cm.warning
self.assertEqual(the_warning.some_attribute, 147)
```

assertWarnsRegex(*expected_warning*, *expected_regex*, **args*, ***kwargs*)

Asserts that the message in a triggered warning matches a regexp. Basic functioning is similar to assertWarns() with the addition that only warnings whose messages also match the regular expression are considered successful matches.

Parameters

- **expected_warning** – Warning class expected to be triggered.
- **expected_regex** – Regex (re pattern object or string) expected to be found in error message.
- **args** – Function to be called and extra positional args.
- **kwargs** – Extra kwargs.
- **msg** – Optional message used in case of failure. Can only be used when assertWarnsRegex is used as a context manager.

drain_ball()

Drain all the balls in play.

Does not actually require any ball devices to be present in the config file.

fail(*msg=None*)

Fail immediately, with the given message.

fill_troughs()

Fill all ball devices tagged with `trough` with balls.

getConfigFile()

Return a string name of the machine config file to use for the tests in this class.

You should override this method in your own test class to point to the config file you need for your tests.

Returns A string name of the machine config file to use, complete with the `.yaml` file extension.

For example:

```
def getConfigFile(self):
    return 'my_config.yaml'
```

getMachinePath()

Return a string name of the path to the machine folder to use for the tests in this class.

You should override this method in your own test class to point to the machine folder root you need for your tests.

Returns A string name of the machine path to use

For example:

```
def getMachinePath(self):
    return 'tests/machine_files/my_test/'
```

Note that this path is relative to the MPF package root

get_enable_plugins()
Control whether tests in this class should load MPF plugins.

Returns: True or False

The default is False. To load plugins in your test class, add the following:

```
def get_enable_plugins(self):
    return True
```

get_platform()
Force this test class to use a certain platform.

Returns String name of the platform this test class will use.

If you don't include this method in your test class, the platform will be set to *virtual*. If you want to use the smart virtual platform, you would add the following to your test class:

```
def get_platform(self):
    return 'smart_virtual`
```

get_use_bcp()
Control whether tests in this class should use BCP.

Returns: True or False

The default is False. To use BCP in your test class, add the following:

```
def get_use_bcp(self):
    return True
```

hit_and_release_switch(*name*)
Momentarily activates and then deactivates a switch.

Parameters **name** – The name of the switch to hit.

This method immediately activates and deactivates a switch with no time in between.

hit_and_release_switches_simultaneously(*names*)
Momentarily activates and then deactivates multiple switches.

Switches are hit sequentially and then released sequentially. Events are only processed at the end of the sequence which is useful to reproduce race conditions when processing nearly simultaneous hits.

Parameters **names** – The names of the switches to hit and release.

hit_switch_and_run(*name*, *delta*)
Activates a switch and advances the time.

Parameters

- **name** – The name of the switch to activate.
- **delta** – The time (in seconds) to advance the clock.

Note that this method does not deactivate the switch once the time has been advanced, meaning the switch stays active. To make the switch inactive, use the *release_switch_and_run()*.

machine_run()
Process any delays, timers, or anything else scheduled.

Note this is the same as:

```
self.advance_time_and_run(0)
```

mock_event(*event_name*)
Configure an event to be mocked.

Parameters **event_name** – String name of the event to mock.

Mocking an event is an easy way to check if an event was called without configuring some kind of callback action in your tests.

Note that an event must be mocked here *before* it's posted in order for *assertEventNotCalled()* and *assertEventCalled()* to work.

Mocking an event will not "break" it. In other words, any other registered handlers for this event will also be called even if the event has been mocked.

For example:

```
self.mock_event('my_event')
self.assertEventNotCalled('my_event')  # This will be True
self.post_event('my_event')
self.assertEventCalled('my_event')  # This will also be True
```

post_event(*event_name*, *run_time=0*)
Post an MPF event and optionally advance the time.

Parameters

- **event_name** – String name of the event to post
- **run_time** – How much time (in seconds) the test should advance after this event has been posted.

For example, to post an event called "shot1_hit":

```
self.post_event('shot1_hit')
```

To post an event called "tilt" and then advance the time 1.5 seconds:

```
self.post_event('tilt', 1.5)
```

post_event_with_params(*event_name*, ***params*)
Post an MPF event with kwarg parameters.

Parameters

- **event_name** – String name of the event to post
- ****params** – One or more kwarg key/value pairs to post with the event.

For example, to post an event called "jackpot" with the parameters `count=1` and `first_time=True`, you would use:

```
self.post_event('jackpot', count=1, first_time=True)
```

release_switch_and_run(*name*, *delta*)
Deactivates a switch and advances the time.

Parameters

- **name** – The name of the switch to activate.
- **delta** – The time (in seconds) to advance the clock.

reset_mock_events()
Reset all mocked events.

This will reset the count of number of times called every mocked event is.

setUp()
Hook method for setting up the test fixture before exercising it.

set_num_balls_known(*balls*)
Set the ball controller's `num_balls_known` attribute.

This is needed for tests where you don't have any ball devices and other situations where you need the ball controller to think the machine has a certain amount of balls to run a test.

Example:

```
self.set_num_balls_known(3)
```

shortDescription()
Returns a one-line description of the test, or None if no description has been provided.

The default implementation of this method returns the first line of the specified test method's docstring.

skipTest(*reason*)
Skip this test.

start_game()
Start a game.

Does not require ball devices or a start button to be present in the config file. Sets the number of known balls to 3.

start_mode(*mode*)
Start mode.

start_two_player_game()
Start two player game.

stop_game(*stop_time=1*)
Stop the current game.

This method asserts that a game is running, then call's the game mode's `end_game()` method, then asserts that the game has successfully stopped.

Example:

```
self.stop_game()
```

stop_mode(*mode*)
Stop mode.

tearDown()
Hook method for deconstructing the test fixture after testing it.

static unittest_verbosity()
Return the verbosity setting of the currently running unittest program, or 0 if none is running.

Returns: An integer value of the current verbosity setting.

MpfGameTestCase

class mpf.tests.MpfGameTestCase.**MpfGameTestCase**(*methodName*)
Bases: *mpf.tests.MpfTestCase.MpfTestCase*

Test case for starting and running games.

This is based on `MpfTestCase` but adds methods and assertions related to running games (rather than just testing MPF components or devices).

Methods & Attributes

The MpfGameTestCase has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

add_player()
Add a player to the current game.

This method hits and releases a switch called `s_start` and then verifies that the player count actually increased by 1.

You can call this method multiple times to add multiple players. For example, to start a game and then add 2 additional players (for 3 players total), you would use:

```
self.start_game()
self.add_player()
self.add_player()
```

static add_to_config_validator(*machine*, *key*, *new_dict*)
Add config dict to validator.

advance_time_and_run(*delta=1.0*)
Advance the test clock and run anything that should run during that time.

Parameters **delta** – How much time to advance the test clock by (in seconds)

This method will cause anything scheduled during the time to run, including things like delays, timers, etc.

Advancing the clock will happen in multiple small steps if things are scheduled to happen during this advance. For example, you can advance the clock 10 seconds like this:

```
self.advance_time_and_run(10)
```

If there is a delay callback that is scheduled to happen in 2 seconds, then this method will advance the clock 2 seconds, process that delay, and then advance the remaining 8 seconds.

assertAlmostEqual(*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Fail if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is more than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

If the two objects compare equal then they will automatically compare almost equal.

assertBallNumber(*number*)
Asserts that the current ball is a certain ball numebr.

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

Raises

- **Assertion error if there is no game in progress or if** –
- **the current ball is not the ball number passed.** –

The following code will check to make sure the game is on Ball 1:

```
self.assertBallNumber(1)
```

assertBallsInPlay(*balls*)

Asserts that a certain number of balls are in play.

Note that the number of balls in play is not necessarily the same as the number of balls on the playfield. (For example, a ball could be held in a ball device, or the machine could be in the process of adding a ball to the platfield.)

Parameters **balls** – The number of balls you want to assert are in play.

To assert that there are 3 balls in play (perhaps during a multiball), you would use:

```
self.assertBallsInPlay(3)
```

assertColorAlmostEqual(*color1*, *color2*, *delta=6*)

Assert that two color are almost equal.

Parameters

- **color1** – The first color, as an RGBColor instance or 3-item iterable.
- **color2** – The second color, as an RGBColor instance or 3-item iterable.
- **delta** – How close the colors have to be. The deltas between red, green, and blue are added together and must be less or equal to this value for this assertion to succeed.

assertCountEqual(*first*, *second*, *msg=None*)

An unordered sequence comparison asserting that the same elements, regardless of order. If the same element occurs more than once, it verifies that the elements occur the same number of times.

self.assertEqual(Counter(list(first)), Counter(list(second)))

Example:

- [0, 1, 1] and [1, 0, 1] compare equal.
- [0, 0, 1] and [0, 1] compare unequal.

assertDictContainsSubset(*subset*, *dictionary*, *msg=None*)

Checks whether dictionary is a superset of subset.

assertEqual(*first*, *second*, *msg=None*)

Fail if the two objects are unequal as determined by the '==' operator.

assertEventCalled(*event_name*, *times=None*)

Assert that event was called.

Parameters

- **event_name** – String name of the event to check.
- **times** – An optional value to confirm the number of times the event was called. Default of *None* means this method will pass as long as the event has been called at least once.

If you want to reset the `times` count, you can mock the event again.

Note that the event must be mocked via `self.mock_event()` first in order to use this method.

For example:

```
self.mock_event('my_event')
self.assertEventNotCalled('my_event')  # This will pass

self.post_event('my_event')
self.assertEventCalled('my_event')     # This will pass
self.assertEventCalled('my_event', 1)  # This will pass

self.post_event('my_event')
self.assertEventCalled('my_event')     # This will pass
self.assertEventCalled('my_event', 2)  # This will pass
```

assertEventCalledWith(*event_name*, ***kwargs*)
Assert an event was called with certain kwargs.

Parameters

- **event_name** – String name of the event to check.
- ****kwargs** – Name/value parameters to check.

For example:

```
self.mock_event('jackpot')

self.post_event('jackpot', count=1, first_time=True)
self.assertEventCalled('jackpot')  # This will pass
self.assertEventCalledWith('jackpot', count=1, first_time=True)  # This will␣
↪also pass
self.assertEventCalledWith('jackpot', count=1, first_time=False)  # This will␣
↪fail
```

assertEventNotCalled(*event_name*)
Assert that event was not called.

Parameters **event_name** – String name of the event to check.

Note that the event must be mocked via `self.mock_event()` first in order to use this method.

assertFalse(*expr*, *msg=None*)
Check that the expression is false.

assertGameIsNotRunning()
Assert a game is not running.

Example:

```
self.assertGameIsNotRunning()
```

assertGameIsRunning()
Assert a game is running.

Example:

```
self.assertGameIsRunning()
```

assertGreater(*a*, *b*, *msg=None*)
Just like self.assertTrue(a > b), but with a nicer default message.

assertGreaterEqual(*a*, *b*, *msg=None*)
Just like self.assertTrue(a >= b), but with a nicer default message.

assertIn(*member*, *container*, *msg=None*)
Just like self.assertTrue(a in b), but with a nicer default message.

assertIs(*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is b), but with a nicer default message.

assertIsInstance(*obj*, *cls*, *msg=None*)
Same as self.assertTrue(isinstance(obj, cls)), with a nicer default message.

assertIsNone(*obj*, *msg=None*)
Same as self.assertTrue(obj is None), with a nicer default message.

assertIsNot(*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is not b), but with a nicer default message.

assertIsNotNone(*obj*, *msg=None*)
Included for symmetry with assertIsNone.

assertLess(*a*, *b*, *msg=None*)
Just like self.assertTrue(a < b), but with a nicer default message.

assertLessEqual(*a*, *b*, *msg=None*)
Just like self.assertTrue(a <= b), but with a nicer default message.

assertListEqual(*list1*, *list2*, *msg=None*)
A list-specific equality assertion.

Parameters

- **list1** – The first list to compare.
- **list2** – The second list to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertLogs(*logger=None*, *level=None*)
Fail unless a log message of level *level* or higher is emitted on *logger_name* or its children. If omitted, *level* defaults to INFO and *logger* defaults to the root logger.

This method must be used as a context manager, and will yield a recording object with two attributes: *output* and *records*. At the end of the context manager, the *output* attribute will be a list of the matching formatted log messages and the *records* attribute will be a list of the corresponding LogRecord objects.

Example:

```
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

assertMultiLineEqual(*first*, *second*, *msg=None*)
Assert that two multi-line strings are equal.

assertNotAlmostEqual(*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Fail if the two objects are equal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is less than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

Objects that are equal automatically fail.

assertNotEqual(*first*, *second*, *msg=None*)
Fail if the two objects are equal as determined by the '!=' operator.

assertNotIn(*member*, *container*, *msg=None*)
Just like self.assertTrue(a not in b), but with a nicer default message.

assertNotIsInstance(*obj*, *cls*, *msg=None*)
Included for symmetry with assertIsInstance.

assertNotRegex(*text*, *unexpected_regex*, *msg=None*)
Fail the test if the text matches the regular expression.

assertNumBallsKnown(*balls*)
Assert that a certain number of balls are known in the machine.

assertPlayerCount(*count*)
Asserts that count players exist.

Parameters **count** – The expected number of players.

For example, to assert that the to players are in the game:

```
self.assertPlayerCount(2)
```

assertPlayerNumber(*number*)
Asserts that the current player is a certain player number.

Parameters **number** – The player number you can to assert is the current player.

For example, to assert that the current player is Player 2, you would use:

```
self.assertPlayerNumber(2)
```

assertRaises(*expected_exception*, **args*, ***kwargs*)
Fail unless an exception of class expected_exception is raised by the callable when invoked with specified positional and keyword arguments. If a different type of exception is raised, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.

If called with the callable and arguments omitted, will return a context object used like this:

```
with self.assertRaises(SomeException):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertRaises is used as a context object.

The context manager keeps a reference to the exception as the 'exception' attribute. This allows you to inspect the exception after the assertion:

```
with self.assertRaises(SomeException) as cm:
    do_something()
the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

assertRaisesRegex(*expected_exception*, *expected_regex*, **args*, ***kwargs*)
Asserts that the message in a raised exception matches a regex.

Parameters

- **expected_exception** – Exception class expected to be raised.

- **expected_regex** – Regex (re pattern object or string) expected to be found in error message.
- **args** – Function to be called and extra positional args.
- **kwargs** – Extra kwargs.
- **msg** – Optional message used in case of failure. Can only be used when assertRaisesRegex is used as a context manager.

assertRegex(*text*, *expected_regex*, *msg=None*)
Fail the test unless the text matches the regular expression.

assertSequenceEqual(*seq1*, *seq2*, *msg=None*, *seq_type=None*)
An equality assertion for ordered sequences (like lists and tuples).

For the purposes of this function, a valid ordered sequence type is one which can be indexed, has a length, and has an equality operator.

Parameters

- **seq1** – The first sequence to compare.
- **seq2** – The second sequence to compare.
- **seq_type** – The expected datatype of the sequences, or None if no datatype should be enforced.
- **msg** – Optional message to use on failure instead of a list of differences.

assertSetEqual(*set1*, *set2*, *msg=None*)
A set-specific equality assertion.

Parameters

- **set1** – The first set to compare.
- **set2** – The second set to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertSetEqual uses ducktyping to support different types of sets, and is optimized for sets specifically (parameters must support a difference method).

assertTrue(*expr*, *msg=None*)
Check that the expression is true.

assertTupleEqual(*tuple1*, *tuple2*, *msg=None*)
A tuple-specific equality assertion.

Parameters

- **tuple1** – The first tuple to compare.
- **tuple2** – The second tuple to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertWarns(*expected_warning*, **args*, ***kwargs*)
Fail unless a warning of class warnClass is triggered by the callable when invoked with specified positional and keyword arguments. If a different type of warning is triggered, it will not be handled: depending on the other warning filtering rules in effect, it might be silenced, printed out, or raised as an exception.

If called with the callable and arguments omitted, will return a context object used like this:

```
with self.assertWarns(SomeWarning):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertWarns is used as a context object.

The context manager keeps a reference to the first matching warning as the 'warning' attribute; similarly, the 'filename' and 'lineno' attributes give you information about the line of Python code from which the warning was triggered. This allows you to inspect the warning after the assertion:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()
the_warning = cm.warning
self.assertEqual(the_warning.some_attribute, 147)
```

assertWarnsRegex(*expected_warning*, *expected_regex*, **args*, ***kwargs*)

Asserts that the message in a triggered warning matches a regexp. Basic functioning is similar to assertWarns() with the addition that only warnings whose messages also match the regular expression are considered successful matches.

Parameters

- **expected_warning** – Warning class expected to be triggered.
- **expected_regex** – Regex (re pattern object or string) expected to be found in error message.
- **args** – Function to be called and extra positional args.
- **kwargs** – Extra kwargs.
- **msg** – Optional message used in case of failure. Can only be used when assertWarnsRegex is used as a context manager.

drain_ball()

Drain a single ball.

If more than 1 ball is in play, this method will need to be called once for each ball in order to end the current ball.

For example, if you have a three-ball multiball and you want to drain all the balls (and end the ball), you would use:

```
self.drain_ball()
self.drain_ball()
self.drain_ball()
```

fail(*msg=None*)

Fail immediately, with the given message.

fill_troughs()

Fill all ball devices tagged with `trough` with balls.

getConfigFile()

Return a string name of the machine config file to use for the tests in this class.

You should override this method in your own test class to point to the config file you need for your tests.

Returns A string name of the machine config file to use, complete with the `.yaml` file extension.

For example:

```
def getConfigFile(self):
    return 'my_config.yaml'
```

getMachinePath()
Return a string name of the path to the machine folder to use for the tests in this class.

You should override this method in your own test class to point to the machine folder root you need for your tests.

Returns A string name of the machine path to use

For example:

```
def getMachinePath(self):
    return 'tests/machine_files/my_test/'
```

Note that this path is relative to the MPF package root

get_enable_plugins()
Control whether tests in this class should load MPF plugins.

Returns: True or False

The default is False. To load plugins in your test class, add the following:

```
def get_enable_plugins(self):
    return True
```

get_platform()
Force this test class to use a certain platform.

Returns String name of the platform this test class will use.

If you don't include this method in your test class, the platform will be set to *virtual*. If you want to use the smart virtual platform, you would add the following to your test class:

```
def get_platform(self):
    return 'smart_virtual`
```

get_use_bcp()
Control whether tests in this class should use BCP.

Returns: True or False

The default is False. To use BCP in your test class, add the following:

```
def get_use_bcp(self):
    return True
```

hit_and_release_switch(*name*)
Momentarily activates and then deactivates a switch.

Parameters **name** – The name of the switch to hit.

This method immediately activates and deactivates a switch with no time in between.

hit_and_release_switches_simultaneously(*names*)
Momentarily activates and then deactivates multiple switches.

Switches are hit sequentially and then released sequentially. Events are only processed at the end of the sequence which is useful to reproduce race conditions when processing nearly simultaneous hits.

Parameters **names** – The names of the switches to hit and release.

hit_switch_and_run(*name*, *delta*)
Activates a switch and advances the time.

Parameters

- **name** – The name of the switch to activate.
- **delta** – The time (in seconds) to advance the clock.

Note that this method does not deactivate the switch once the time has been advanced, meaning the switch stays active. To make the switch inactive, use the *release_switch_and_run()*.

machine_run()
Process any delays, timers, or anything else scheduled.

Note this is the same as:

```
self.advance_time_and_run(0)
```

mock_event(*event_name*)
Configure an event to be mocked.

Parameters **event_name** – String name of the event to mock.

Mocking an event is an easy way to check if an event was called without configuring some kind of callback action in your tests.

Note that an event must be mocked here *before* it's posted in order for *assertEventNotCalled()* and *assertEventCalled()* to work.

Mocking an event will not "break" it. In other words, any other registered handlers for this event will also be called even if the event has been mocked.

For example:

```
self.mock_event('my_event')
self.assertEventNotCalled('my_event')  # This will be True
self.post_event('my_event')
self.assertEventCalled('my_event')  # This will also be True
```

post_event(*event_name*, *run_time=0*)
Post an MPF event and optionally advance the time.

Parameters

- **event_name** – String name of the event to post
- **run_time** – How much time (in seconds) the test should advance after this event has been posted.

For example, to post an event called "shot1_hit":

```
self.post_event('shot1_hit')
```

To post an event called "tilt" and then advance the time 1.5 seconds:

```
self.post_event('tilt', 1.5)
```

post_event_with_params(*event_name*, ***params*)
Post an MPF event with kwarg parameters.

Parameters

- **event_name** – String name of the event to post

- ****params** – One or more kwarg key/value pairs to post with the event.

For example, to post an event called "jackpot" with the parameters `count=1` and `first_time=True`, you would use:

```
self.post_event('jackpot', count=1, first_time=True)
```

release_switch_and_run(*name*, *delta*)
Deactivates a switch and advances the time.

Parameters

- **name** – The name of the switch to activate.
- **delta** – The time (in seconds) to advance the clock.

reset_mock_events()
Reset all mocked events.

This will reset the count of number of times called every mocked event is.

setUp()
Hook method for setting up the test fixture before exercising it.

set_num_balls_known(*balls*)
Set the ball controller's `num_balls_known` attribute.

This is needed for tests where you don't have any ball devices and other situations where you need the ball controller to think the machine has a certain amount of balls to run a test.

Example:

```
self.set_num_balls_known(3)
```

shortDescription()
Returns a one-line description of the test, or None if no description has been provided.

The default implementation of this method returns the first line of the specified test method's docstring.

skipTest(*reason*)
Skip this test.

start_game()
Start a game.

This method checks to make sure a game is not running, then hits and releases the `s_start` switch, and finally checks to make sure a game actually started properly.

For example:

```
self.start_game()
```

start_mode(*mode*)
Start mode.

start_two_player_game()
Start two player game.

stop_game(*stop_time=1*)
Stop the current game.

This method asserts that a game is running, then call's the game mode's `end_game()` method, then asserts that the game has successfully stopped.

Example:

```
self.stop_game()
```

stop_mode(*mode*)
Stop mode.

tearDown()
Hook method for deconstructing the test fixture after testing it.

static unittest_verbosity()
Return the verbosity setting of the currently running unittest program, or 0 if none is running.

Returns: An integer value of the current verbosity setting.

MpfMachineTestCase

class mpf.tests.MpfMachineTestCase.**MpfMachineTestCase**(*methodName='runTest'*)
Bases: mpf.tests.MpfMachineTestCase.BaseMpfMachineTestCase

MPF only machine test case.

Methods & Attributes

The MpfMachineTestCase has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

static add_to_config_validator(*machine*, *key*, *new_dict*)
Add config dict to validator.

advance_time_and_run(*delta=1.0*)
Advance the test clock and run anything that should run during that time.

Parameters **delta** – How much time to advance the test clock by (in seconds)

This method will cause anything scheduled during the time to run, including things like delays, timers, etc.

Advancing the clock will happen in multiple small steps if things are scheduled to happen during this advance. For example, you can advance the clock 10 seconds like this:

```
self.advance_time_and_run(10)
```

If there is a delay callback that is scheduled to happen in 2 seconds, then this method will advance the clock 2 seconds, process that delay, and then advance the remaining 8 seconds.

assertAlmostEqual(*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Fail if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is more than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

If the two objects compare equal then they will automatically compare almost equal.

assertColorAlmostEqual(*color1*, *color2*, *delta=6*)
Assert that two color are almost equal.

Parameters

- **color1** – The first color, as an RGBColor instance or 3-item iterable.
- **color2** – The second color, as an RGBColor instance or 3-item iterable.
- **delta** – How close the colors have to be. The deltas between red, green, and blue are added together and must be less or equal to this value for this assertion to succeed.

assertCountEqual(*first*, *second*, *msg=None*)
An unordered sequence comparison asserting that the same elements, regardless of order. If the same element occurs more than once, it verifies that the elements occur the same number of times.

self.assertEqual(Counter(list(first)), Counter(list(second)))

Example:

- [0, 1, 1] and [1, 0, 1] compare equal.
- [0, 0, 1] and [0, 1] compare unequal.

assertDictContainsSubset(*subset*, *dictionary*, *msg=None*)
Checks whether dictionary is a superset of subset.

assertEqual(*first*, *second*, *msg=None*)
Fail if the two objects are unequal as determined by the '==' operator.

assertEventCalled(*event_name*, *times=None*)
Assert that event was called.

Parameters

- **event_name** – String name of the event to check.
- **times** – An optional value to confirm the number of times the event was called. Default of *None* means this method will pass as long as the event has been called at least once.

If you want to reset the `times` count, you can mock the event again.

Note that the event must be mocked via `self.mock_event()` first in order to use this method.

For example:

```
self.mock_event('my_event')
self.assertEventNotCalled('my_event')  # This will pass

self.post_event('my_event')
self.assertEventCalled('my_event')     # This will pass
self.assertEventCalled('my_event', 1)  # This will pass

self.post_event('my_event')
self.assertEventCalled('my_event')     # This will pass
self.assertEventCalled('my_event', 2)  # This will pass
```

assertEventCalledWith(*event_name*, ***kwargs*)
Assert an event was called with certain kwargs.

Parameters

- **event_name** – String name of the event to check.
- ****kwargs** – Name/value parameters to check.

For example:

```
self.mock_event('jackpot')

self.post_event('jackpot', count=1, first_time=True)
self.assertEventCalled('jackpot')  # This will pass
self.assertEventCalledWith('jackpot', count=1, first_time=True)  # This will␣
↪also pass
self.assertEventCalledWith('jackpot', count=1, first_time=False)  # This will␣
↪fail
```

assertEventNotCalled(*event_name*)
Assert that event was not called.

Parameters **event_name** – String name of the event to check.

Note that the event must be mocked via `self.mock_event()` first in order to use this method.

assertFalse(*expr*, *msg=None*)
Check that the expression is false.

assertGreater(*a*, *b*, *msg=None*)
Just like self.assertTrue(a > b), but with a nicer default message.

assertGreaterEqual(*a*, *b*, *msg=None*)
Just like self.assertTrue(a >= b), but with a nicer default message.

assertIn(*member*, *container*, *msg=None*)
Just like self.assertTrue(a in b), but with a nicer default message.

assertIs(*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is b), but with a nicer default message.

assertIsInstance(*obj*, *cls*, *msg=None*)
Same as self.assertTrue(isinstance(obj, cls)), with a nicer default message.

assertIsNone(*obj*, *msg=None*)
Same as self.assertTrue(obj is None), with a nicer default message.

assertIsNot(*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is not b), but with a nicer default message.

assertIsNotNone(*obj*, *msg=None*)
Included for symmetry with assertIsNone.

assertLess(*a*, *b*, *msg=None*)
Just like self.assertTrue(a < b), but with a nicer default message.

assertLessEqual(*a*, *b*, *msg=None*)
Just like self.assertTrue(a <= b), but with a nicer default message.

assertListEqual(*list1*, *list2*, *msg=None*)
A list-specific equality assertion.

Parameters

- **list1** – The first list to compare.
- **list2** – The second list to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertLogs(*logger=None*, *level=None*)
Fail unless a log message of level *level* or higher is emitted on *logger_name* or its children. If omitted, *level* defaults to INFO and *logger* defaults to the root logger.

This method must be used as a context manager, and will yield a recording object with two attributes: *output* and *records*. At the end of the context manager, the *output* attribute will be a list of the matching formatted log messages and the *records* attribute will be a list of the corresponding LogRecord objects.

Example:

```
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

assertMultiLineEqual (*first*, *second*, *msg=None*)
Assert that two multi-line strings are equal.

assertNotAlmostEqual (*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Fail if the two objects are equal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is less than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

Objects that are equal automatically fail.

assertNotEqual (*first*, *second*, *msg=None*)
Fail if the two objects are equal as determined by the '!=' operator.

assertNotIn (*member*, *container*, *msg=None*)
Just like self.assertTrue(a not in b), but with a nicer default message.

assertNotIsInstance (*obj*, *cls*, *msg=None*)
Included for symmetry with assertIsInstance.

assertNotRegex (*text*, *unexpected_regex*, *msg=None*)
Fail the test if the text matches the regular expression.

assertNumBallsKnown (*balls*)
Assert that a certain number of balls are known in the machine.

assertRaises (*expected_exception*, **args*, ***kwargs*)
Fail unless an exception of class expected_exception is raised by the callable when invoked with specified positional and keyword arguments. If a different type of exception is raised, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.

If called with the callable and arguments omitted, will return a context object used like this:

```
with self.assertRaises(SomeException):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertRaises is used as a context object.

The context manager keeps a reference to the exception as the 'exception' attribute. This allows you to inspect the exception after the assertion:

```
with self.assertRaises(SomeException) as cm:
    do_something()
the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

assertRaisesRegex (*expected_exception*, *expected_regex*, **args*, ***kwargs*)
Asserts that the message in a raised exception matches a regex.

Parameters

- **expected_exception** – Exception class expected to be raised.
- **expected_regex** – Regex (re pattern object or string) expected to be found in error message.
- **args** – Function to be called and extra positional args.
- **kwargs** – Extra kwargs.
- **msg** – Optional message used in case of failure. Can only be used when assertRaisesRegex is used as a context manager.

assertRegex(*text*, *expected_regex*, *msg=None*)
Fail the test unless the text matches the regular expression.

assertSequenceEqual(*seq1*, *seq2*, *msg=None*, *seq_type=None*)
An equality assertion for ordered sequences (like lists and tuples).

For the purposes of this function, a valid ordered sequence type is one which can be indexed, has a length, and has an equality operator.

Parameters

- **seq1** – The first sequence to compare.
- **seq2** – The second sequence to compare.
- **seq_type** – The expected datatype of the sequences, or None if no datatype should be enforced.
- **msg** – Optional message to use on failure instead of a list of differences.

assertSetEqual(*set1*, *set2*, *msg=None*)
A set-specific equality assertion.

Parameters

- **set1** – The first set to compare.
- **set2** – The second set to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertSetEqual uses ducktyping to support different types of sets, and is optimized for sets specifically (parameters must support a difference method).

assertTrue(*expr*, *msg=None*)
Check that the expression is true.

assertTupleEqual(*tuple1*, *tuple2*, *msg=None*)
A tuple-specific equality assertion.

Parameters

- **tuple1** – The first tuple to compare.
- **tuple2** – The second tuple to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertWarns(*expected_warning*, **args*, ***kwargs*)
Fail unless a warning of class warnClass is triggered by the callable when invoked with specified positional and keyword arguments. If a different type of warning is triggered, it will not be handled: depending on the other warning filtering rules in effect, it might be silenced, printed out, or raised as an exception.

If called with the callable and arguments omitted, will return a context object used like this:

```
with self.assertWarns(SomeWarning):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertWarns is used as a context object.

The context manager keeps a reference to the first matching warning as the 'warning' attribute; similarly, the 'filename' and 'lineno' attributes give you information about the line of Python code from which the warning was triggered. This allows you to inspect the warning after the assertion:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()
the_warning = cm.warning
self.assertEqual(the_warning.some_attribute, 147)
```

assertWarnsRegex(*expected_warning*, *expected_regex*, **args*, ***kwargs*)

Asserts that the message in a triggered warning matches a regexp. Basic functioning is similar to assertWarns() with the addition that only warnings whose messages also match the regular expression are considered successful matches.

Parameters

- **expected_warning** – Warning class expected to be triggered.
- **expected_regex** – Regex (re pattern object or string) expected to be found in error message.
- **args** – Function to be called and extra positional args.
- **kwargs** – Extra kwargs.
- **msg** – Optional message used in case of failure. Can only be used when assertWarnsRegex is used as a context manager.

fail(*msg=None*)

Fail immediately, with the given message.

getConfigFile()

Return a string name of the machine config file to use for the tests in this class.

You should override this method in your own test class to point to the config file you need for your tests.

Returns A string name of the machine config file to use, complete with the `.yaml` file extension.

For example:

```
def getConfigFile(self):
    return 'my_config.yaml'
```

getMachinePath()

Return a string name of the path to the machine folder to use for the tests in this class.

You should override this method in your own test class to point to the machine folder root you need for your tests.

Returns A string name of the machine path to use

For example:

```
def getMachinePath(self):
    return 'tests/machine_files/my_test/'
```

Note that this path is relative to the MPF package root

get_enable_plugins()
Control whether tests in this class should load MPF plugins.

Returns: True or False

The default is False. To load plugins in your test class, add the following:

```
def get_enable_plugins(self):
    return True
```

get_platform()
Force this test class to use a certain platform.

Returns String name of the platform this test class will use.

If you don't include this method in your test class, the platform will be set to *virtual*. If you want to use the smart virtual platform, you would add the following to your test class:

```
def get_platform(self):
    return 'smart_virtual`
```

get_use_bcp()
Control whether tests in this class should use BCP.

Returns: True or False

The default is False. To use BCP in your test class, add the following:

```
def get_use_bcp(self):
    return True
```

hit_and_release_switch(*name*)
Momentarily activates and then deactivates a switch.

Parameters **name** – The name of the switch to hit.

This method immediately activates and deactivates a switch with no time in between.

hit_and_release_switches_simultaneously(*names*)
Momentarily activates and then deactivates multiple switches.

Switches are hit sequentially and then released sequentially. Events are only processed at the end of the sequence which is useful to reproduce race conditions when processing nearly simultaneous hits.

Parameters **names** – The names of the switches to hit and release.

hit_switch_and_run(*name*, *delta*)
Activates a switch and advances the time.

Parameters

- **name** – The name of the switch to activate.
- **delta** – The time (in seconds) to advance the clock.

Note that this method does not deactivate the switch once the time has been advanced, meaning the switch stays active. To make the switch inactive, use the *release_switch_and_run()*.

machine_run()
Process any delays, timers, or anything else scheduled.

Note this is the same as:

```
self.advance_time_and_run(0)
```

mock_event(*event_name*)
Configure an event to be mocked.

Parameters **event_name** – String name of the event to mock.

Mocking an event is an easy way to check if an event was called without configuring some kind of callback action in your tests.

Note that an event must be mocked here *before* it's posted in order for *assertEventNotCalled()* and *assertEventCalled()* to work.

Mocking an event will not "break" it. In other words, any other registered handlers for this event will also be called even if the event has been mocked.

For example:

```
self.mock_event('my_event')
self.assertEventNotCalled('my_event')  # This will be True
self.post_event('my_event')
self.assertEventCalled('my_event')  # This will also be True
```

post_event(*event_name*, *run_time=0*)
Post an MPF event and optionally advance the time.

Parameters

- **event_name** – String name of the event to post
- **run_time** – How much time (in seconds) the test should advance after this event has been posted.

For example, to post an event called "shot1_hit":

```
self.post_event('shot1_hit')
```

To post an event called "tilt" and then advance the time 1.5 seconds:

```
self.post_event('tilt', 1.5)
```

post_event_with_params(*event_name*, ***params*)
Post an MPF event with kwarg parameters.

Parameters

- **event_name** – String name of the event to post
- ****params** – One or more kwarg key/value pairs to post with the event.

For example, to post an event called "jackpot" with the parameters `count=1` and `first_time=True`, you would use:

```
self.post_event('jackpot', count=1, first_time=True)
```

release_switch_and_run(*name*, *delta*)
Deactivates a switch and advances the time.

Parameters

- **name** – The name of the switch to activate.
- **delta** – The time (in seconds) to advance the clock.

reset_mock_events()
Reset all mocked events.

This will reset the count of number of times called every mocked event is.

setUp()
Hook method for setting up the test fixture before exercising it.

set_num_balls_known(*balls*)
Set the ball controller's num_balls_known attribute.

This is needed for tests where you don't have any ball devices and other situations where you need the ball controller to think the machine has a certain amount of balls to run a test.

Example:

```
self.set_num_balls_known(3)
```

shortDescription()
Returns a one-line description of the test, or None if no description has been provided.

The default implementation of this method returns the first line of the specified test method's docstring.

skipTest(*reason*)
Skip this test.

start_mode(*mode*)
Start mode.

stop_mode(*mode*)
Stop mode.

tearDown()
Hook method for deconstructing the test fixture after testing it.

static unittest_verbosity()
Return the verbosity setting of the currently running unittest program, or 0 if none is running.

Returns: An integer value of the current verbosity setting.

MpfTestCase

class mpf.tests.MpfTestCase.**MpfTestCase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Primary TestCase class used for all MPF unit tests.

Methods & Attributes

The MpfTestCase has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

static add_to_config_validator(*machine*, *key*, *new_dict*)
Add config dict to validator.

advance_time_and_run(*delta=1.0*)
Advance the test clock and run anything that should run during that time.

Parameters **delta** – How much time to advance the test clock by (in seconds)

This method will cause anything scheduled during the time to run, including things like delays, timers, etc.

Advancing the clock will happen in multiple small steps if things are scheduled to happen during this advance. For example, you can advance the clock 10 seconds like this:

```
self.advance_time_and_run(10)
```

If there is a delay callback that is scheduled to happen in 2 seconds, then this method will advance the clock 2 seconds, process that delay, and then advance the remaining 8 seconds.

assertAlmostEqual(*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Fail if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is more than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

If the two objects compare equal then they will automatically compare almost equal.

assertColorAlmostEqual(*color1*, *color2*, *delta=6*)
Assert that two color are almost equal.

Parameters

- **color1** – The first color, as an RGBColor instance or 3-item iterable.
- **color2** – The second color, as an RGBColor instance or 3-item iterable.
- **delta** – How close the colors have to be. The deltas between red, green, and blue are added together and must be less or equal to this value for this assertion to succeed.

assertCountEqual(*first*, *second*, *msg=None*)
An unordered sequence comparison asserting that the same elements, regardless of order. If the same element occurs more than once, it verifies that the elements occur the same number of times.

self.assertEqual(Counter(list(first)), Counter(list(second)))

Example:

- [0, 1, 1] and [1, 0, 1] compare equal.
- [0, 0, 1] and [0, 1] compare unequal.

assertDictContainsSubset(*subset*, *dictionary*, *msg=None*)
Checks whether dictionary is a superset of subset.

assertEqual(*first*, *second*, *msg=None*)
Fail if the two objects are unequal as determined by the '==' operator.

assertEventCalled(*event_name*, *times=None*)
Assert that event was called.

Parameters

- **event_name** – String name of the event to check.
- **times** – An optional value to confirm the number of times the event was called. Default of *None* means this method will pass as long as the event has been called at least once.

If you want to reset the `times` count, you can mock the event again.

Note that the event must be mocked via `self.mock_event()` first in order to use this method.

For example:

```
self.mock_event('my_event')
self.assertEventNotCalled('my_event')  # This will pass

self.post_event('my_event')
self.assertEventCalled('my_event')     # This will pass
self.assertEventCalled('my_event', 1)  # This will pass

self.post_event('my_event')
self.assertEventCalled('my_event')     # This will pass
self.assertEventCalled('my_event', 2)  # This will pass
```

assertEventCalledWith(*event_name*, ***kwargs*)
Assert an event was called with certain kwargs.

Parameters

- **event_name** – String name of the event to check.
- ****kwargs** – Name/value parameters to check.

For example:

```
self.mock_event('jackpot')

self.post_event('jackpot', count=1, first_time=True)
self.assertEventCalled('jackpot')  # This will pass
self.assertEventCalledWith('jackpot', count=1, first_time=True)  # This will␣
↪also pass
self.assertEventCalledWith('jackpot', count=1, first_time=False)  # This will␣
↪fail
```

assertEventNotCalled(*event_name*)
Assert that event was not called.

Parameters **event_name** – String name of the event to check.

Note that the event must be mocked via `self.mock_event()` first in order to use this method.

assertFalse(*expr*, *msg=None*)
Check that the expression is false.

assertGreater(*a*, *b*, *msg=None*)
Just like self.assertTrue(a > b), but with a nicer default message.

assertGreaterEqual(*a*, *b*, *msg=None*)
Just like self.assertTrue(a >= b), but with a nicer default message.

assertIn(*member*, *container*, *msg=None*)
Just like self.assertTrue(a in b), but with a nicer default message.

assertIs(*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is b), but with a nicer default message.

assertIsInstance(*obj*, *cls*, *msg=None*)
Same as self.assertTrue(isinstance(obj, cls)), with a nicer default message.

assertIsNone(*obj*, *msg=None*)
Same as self.assertTrue(obj is None), with a nicer default message.

assertIsNot(*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is not b), but with a nicer default message.

assertIsNotNone(*obj*, *msg=None*)
Included for symmetry with assertIsNone.

assertLess(*a*, *b*, *msg=None*)
Just like self.assertTrue(a < b), but with a nicer default message.

assertLessEqual(*a*, *b*, *msg=None*)
Just like self.assertTrue(a <= b), but with a nicer default message.

assertListEqual(*list1*, *list2*, *msg=None*)
A list-specific equality assertion.

Parameters

- **list1** – The first list to compare.
- **list2** – The second list to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertLogs(*logger=None*, *level=None*)
Fail unless a log message of level *level* or higher is emitted on *logger_name* or its children. If omitted, *level* defaults to INFO and *logger* defaults to the root logger.

This method must be used as a context manager, and will yield a recording object with two attributes: *output* and *records*. At the end of the context manager, the *output* attribute will be a list of the matching formatted log messages and the *records* attribute will be a list of the corresponding LogRecord objects.

Example:

```
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

assertMultiLineEqual(*first*, *second*, *msg=None*)
Assert that two multi-line strings are equal.

assertNotAlmostEqual(*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Fail if the two objects are equal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is less than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

Objects that are equal automatically fail.

assertNotEqual(*first*, *second*, *msg=None*)
Fail if the two objects are equal as determined by the '!=' operator.

assertNotIn(*member*, *container*, *msg=None*)
Just like self.assertTrue(a not in b), but with a nicer default message.

assertNotIsInstance(*obj*, *cls*, *msg=None*)
Included for symmetry with assertIsInstance.

assertNotRegex(*text*, *unexpected_regex*, *msg=None*)
Fail the test if the text matches the regular expression.

assertNumBallsKnown(*balls*)
Assert that a certain number of balls are known in the machine.

assertRaises(*expected_exception*, **args*, ***kwargs*)
Fail unless an exception of class expected_exception is raised by the callable when invoked with specified positional and keyword arguments. If a different type of exception is raised, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.

If called with the callable and arguments omitted, will return a context object used like this:

```
with self.assertRaises(SomeException):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertRaises is used as a context object.

The context manager keeps a reference to the exception as the 'exception' attribute. This allows you to inspect the exception after the assertion:

```
with self.assertRaises(SomeException) as cm:
    do_something()
the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

assertRaisesRegex(*expected_exception*, *expected_regex*, **args*, ***kwargs*)
Asserts that the message in a raised exception matches a regex.

Parameters

- **expected_exception** – Exception class expected to be raised.
- **expected_regex** – Regex (re pattern object or string) expected to be found in error message.
- **args** – Function to be called and extra positional args.
- **kwargs** – Extra kwargs.
- **msg** – Optional message used in case of failure. Can only be used when assertRaisesRegex is used as a context manager.

assertRegex(*text*, *expected_regex*, *msg=None*)
Fail the test unless the text matches the regular expression.

assertSequenceEqual(*seq1*, *seq2*, *msg=None*, *seq_type=None*)
An equality assertion for ordered sequences (like lists and tuples).

For the purposes of this function, a valid ordered sequence type is one which can be indexed, has a length, and has an equality operator.

Parameters

- **seq1** – The first sequence to compare.
- **seq2** – The second sequence to compare.
- **seq_type** – The expected datatype of the sequences, or None if no datatype should be enforced.
- **msg** – Optional message to use on failure instead of a list of differences.

assertSetEqual(*set1*, *set2*, *msg=None*)
A set-specific equality assertion.

Parameters

- **set1** – The first set to compare.
- **set2** – The second set to compare.

- **msg** – Optional message to use on failure instead of a list of differences.

assertSetEqual uses ducktyping to support different types of sets, and is optimized for sets specifically (parameters must support a difference method).

assertTrue(*expr*, *msg=None*)
Check that the expression is true.

assertTupleEqual(*tuple1*, *tuple2*, *msg=None*)
A tuple-specific equality assertion.

Parameters

- **tuple1** – The first tuple to compare.
- **tuple2** – The second tuple to compare.
- **msg** – Optional message to use on failure instead of a list of differences.

assertWarns(*expected_warning*, **args*, ***kwargs*)
Fail unless a warning of class warnClass is triggered by the callable when invoked with specified positional and keyword arguments. If a different type of warning is triggered, it will not be handled: depending on the other warning filtering rules in effect, it might be silenced, printed out, or raised as an exception.

If called with the callable and arguments omitted, will return a context object used like this:

```
with self.assertWarns(SomeWarning):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertWarns is used as a context object.

The context manager keeps a reference to the first matching warning as the 'warning' attribute; similarly, the 'filename' and 'lineno' attributes give you information about the line of Python code from which the warning was triggered. This allows you to inspect the warning after the assertion:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()
the_warning = cm.warning
self.assertEqual(the_warning.some_attribute, 147)
```

assertWarnsRegex(*expected_warning*, *expected_regex*, **args*, ***kwargs*)
Asserts that the message in a triggered warning matches a regexp. Basic functioning is similar to assertWarns() with the addition that only warnings whose messages also match the regular expression are considered successful matches.

Parameters

- **expected_warning** – Warning class expected to be triggered.
- **expected_regex** – Regex (re pattern object or string) expected to be found in error message.
- **args** – Function to be called and extra positional args.
- **kwargs** – Extra kwargs.
- **msg** – Optional message used in case of failure. Can only be used when assertWarnsRegex is used as a context manager.

fail(*msg=None*)
Fail immediately, with the given message.

getConfigFile()
Return a string name of the machine config file to use for the tests in this class.

You should override this method in your own test class to point to the config file you need for your tests.

Returns A string name of the machine config file to use, complete with the `.yaml` file extension.

For example:

```
def getConfigFile(self):
    return 'my_config.yaml'
```

getMachinePath()
Return a string name of the path to the machine folder to use for the tests in this class.

You should override this method in your own test class to point to the machine folder root you need for your tests.

Returns A string name of the machine path to use

For example:

```
def getMachinePath(self):
    return 'tests/machine_files/my_test/'
```

Note that this path is relative to the MPF package root

get_enable_plugins()
Control whether tests in this class should load MPF plugins.

Returns: True or False

The default is False. To load plugins in your test class, add the following:

```
def get_enable_plugins(self):
    return True
```

get_platform()
Force this test class to use a certain platform.

Returns String name of the platform this test class will use.

If you don't include this method in your test class, the platform will be set to *virtual*. If you want to use the smart virtual platform, you would add the following to your test class:

```
def get_platform(self):
    return 'smart_virtual`
```

get_use_bcp()
Control whether tests in this class should use BCP.

Returns: True or False

The default is False. To use BCP in your test class, add the following:

```
def get_use_bcp(self):
    return True
```

hit_and_release_switch(*name*)
Momentarily activates and then deactivates a switch.

Parameters **name** – The name of the switch to hit.

This method immediately activates and deactivates a switch with no time in between.

hit_and_release_switches_simultaneously(*names*)
Momentarily activates and then deactivates multiple switches.

Switches are hit sequentially and then released sequentially. Events are only processed at the end of the sequence which is useful to reproduce race conditions when processing nearly simultaneous hits.

Parameters **names** – The names of the switches to hit and release.

hit_switch_and_run(*name*, *delta*)
Activates a switch and advances the time.

Parameters

- **name** – The name of the switch to activate.
- **delta** – The time (in seconds) to advance the clock.

Note that this method does not deactivate the switch once the time has been advanced, meaning the switch stays active. To make the switch inactive, use the *release_switch_and_run()*.

machine_run()
Process any delays, timers, or anything else scheduled.

Note this is the same as:

```
self.advance_time_and_run(0)
```

mock_event(*event_name*)
Configure an event to be mocked.

Parameters **event_name** – String name of the event to mock.

Mocking an event is an easy way to check if an event was called without configuring some kind of callback action in your tests.

Note that an event must be mocked here *before* it's posted in order for *assertEventNotCalled()* and *assertEventCalled()* to work.

Mocking an event will not "break" it. In other words, any other registered handlers for this event will also be called even if the event has been mocked.

For example:

```
self.mock_event('my_event')
self.assertEventNotCalled('my_event')  # This will be True
self.post_event('my_event')
self.assertEventCalled('my_event')  # This will also be True
```

post_event(*event_name*, *run_time=0*)
Post an MPF event and optionally advance the time.

Parameters

- **event_name** – String name of the event to post
- **run_time** – How much time (in seconds) the test should advance after this event has been posted.

For example, to post an event called "shot1_hit":

```
self.post_event('shot1_hit')
```

To post an event called "tilt" and then advance the time 1.5 seconds:

```
self.post_event('tilt', 1.5)
```

post_event_with_params(*event_name*, ***params*)
Post an MPF event with kwarg parameters.

Parameters

- **event_name** – String name of the event to post
- ****params** – One or more kwarg key/value pairs to post with the event.

For example, to post an event called "jackpot" with the parameters `count=1` and `first_time=True`, you would use:

```
self.post_event('jackpot', count=1, first_time=True)
```

release_switch_and_run(*name*, *delta*)
Deactivates a switch and advances the time.

Parameters

- **name** – The name of the switch to activate.
- **delta** – The time (in seconds) to advance the clock.

reset_mock_events()
Reset all mocked events.

This will reset the count of number of times called every mocked event is.

setUp()
Hook method for setting up the test fixture before exercising it.

set_num_balls_known(*balls*)
Set the ball controller's `num_balls_known` attribute.

This is needed for tests where you don't have any ball devices and other situations where you need the ball controller to think the machine has a certain amount of balls to run a test.

Example:

```
self.set_num_balls_known(3)
```

shortDescription()
Returns a one-line description of the test, or None if no description has been provided.

The default implementation of this method returns the first line of the specified test method's docstring.

skipTest(*reason*)
Skip this test.

start_mode(*mode*)
Start mode.

stop_mode(*mode*)
Stop mode.

tearDown()
Hook method for deconstructing the test fixture after testing it.

static unittest_verbosity()
Return the verbosity setting of the currently running unittest program, or 0 if none is running.

Returns: An integer value of the current verbosity setting.

TestDataManager

class mpf.tests.TestDataManager.**TestDataManager**(*data*)
Bases: *mpf.core.data_manager.DataManager*

A patched version of the DataManager which is used in unit tests.

The main change is that the `save_all()` method doesn't actually write anything to disk so the tests don't fill up the disk with unneeded data.

Methods & Attributes

The TestDataManager has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

get_data(*section=None*)
Return the value of this DataManager's data.

Parameters **section** – Optional string name of a section (dictionary key) for the data you want returned. Default is None which returns the entire dictionary.

ignorable_runtime_exception(*msg: str*) → None
Handle ignorable runtime exception.

During development or tests raise an exception for easier debugging. Log an error during production.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

save_all(*data*)
Update all data.

warning_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the warning level.

These messages will always be shown in the console and the log file.

TestMachineController

class mpf.tests.MpfTestCase.**TestMachineController**(*mpf_path*, *machine_path*, *options*, *config_patches*, *config_defaults*, *clock*, *mock_data*, *enable_plugins=False*, *early_init=None*)
Bases: *mpf.core.machine.MachineController*

A patched version of the MachineController used in tests.

The TestMachineController has a few changes from the regular machine controller to facilitate running unit tests, including:

- Use the TestDataManager instead of the real one.
- Use a test clock which we can manually advance instead of the regular clock tied to real-world time.
- Only load plugins if `self._enable_plugins` is *True*.

- Merge any `test_config_patches` into the machine config.
- Disabled the config file caching to always load the config from disk.

Methods & Attributes

The TestMachineController has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

add_platform(*name: str*) → None
Make an additional hardware platform interface available to MPF.

Parameters **name** – String name of the platform to add. Must match the name of a platform file in the mpf/platforms folder (without the .py extension).

clear_boot_hold(*hold: str*) → None
Clear a boot hold.

configure_machine_var(*name: str*, *persist: bool*, *expire_secs: int = None*) → None
Create a new machine variable.

Parameters

- **name** – String name of the variable.
- **persist** – Boolean as to whether this variable should be saved to disk so it's available the next time MPF boots.
- **expire_secs** – Optional number of seconds you'd like this variable to persist on disk for. When MPF boots, if the expiration time of the variable is in the past, it will not be loaded. For example, this lets you write the number of credits on the machine to disk to persist even during power off, but you could set it so that those only stay persisted for an hour.

create_data_manager(*config_name*)
Create TestDataManager.

get_machine_var(*name: str*) → Any
Return the value of a machine variable.

Parameters **name** – String name of the variable you want to get that value for.

Returns The value of the variable if it exists, or None if the variable does not exist.

get_platform_sections(*platform_section: str*, *overwrite: str*) → SmartVirtualHardwarePlatform
Return platform section.

ignorable_runtime_exception(*msg: str*) → None
Handle ignorable runtime exception.

During development or tests raise an exception for easier debugging. Log an error during production.

init_done() → Generator[[int, None], None]
Finish init.

Called when init is done and all boot holds are cleared.

initialise() → Generator[[int, None], None]
Initialise machine.

initialise_core_and_hardware() → Generator[[int, None], None]
Load core modules and hardware.

initialise_mpf()
Initialise MPF.

is_machine_var(*name: str*) → bool
Return true if machine variable exists.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

register_boot_hold(*hold: str*) → None
Register a boot hold.

register_monitor(*monitor_class: str, monitor: Callable[..., Any]*) → None
Register a monitor.

Parameters

- **monitor_class** – String name of the monitor class for this monitor that's being registered.
- **monitor** – Callback to notify

MPF uses monitors to allow components to monitor certain internal elements of MPF.

For example, a player variable monitor could be setup to be notified of any changes to a player variable, or a switch monitor could be used to allow a plugin to be notified of any changes to any switches.

The MachineController's list of registered monitors doesn't actually do anything. Rather it's a dictionary of sets which the monitors themselves can reference when they need to do something. We just needed a central registry of monitors.

remove_machine_var(*name: str*) → None
Remove a machine variable by name.

If this variable persists to disk, it will remove it from there too.

Parameters **name** – String name of the variable you want to remove.

remove_machine_var_search(*startswith: str = ''*, *endswith: str = ''*) → None
Remove a machine variable by matching parts of its name.

Parameters

- **startswith** – Optional start of the variable name to match.
- **endswith** – Optional end of the variable name to match.

For example, if you pass startswit='player' and endswith='score', this method will match and remove player1_score, player2_score, etc.

reset() → Generator[[int, None], None]
Reset the machine.

This method is safe to call. It essentially sets up everything from scratch without reloading the config files and assets from disk. This method is called after a game ends and before attract mode begins.

run() → None
Start the main machine run loop.

set_default_platform(*name: str*) → None
Set the default platform.

It is used if a device class-specific or device-specific platform is not specified.

Parameters **name** – String name of the platform to set to default.

set_machine_var(*name: str*, *value: Any*) → None
Set the value of a machine variable.

Parameters

- **name** – String name of the variable you're setting the value for.
- **value** – The value you're setting. This can be any Type.

shutdown() → None
Shutdown the machine.

stop(***kwargs*) → None
Perform a graceful exit of MPF.

validate_machine_config_section(*section: str*) → None
Validate a config section.

verify_system_info()
Dump information about the Python installation to the log.

Information includes Python version, Python executable, platform, and core architecture.

warning_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the warning level.

These messages will always be shown in the console and the log file.

8.3.7 Miscellaneous Components

There are several other components and systems of MPF that don't fit into any of the other categories. Those are covered here.

Ball Search

class mpf.core.ball_search.**BallSearch**(*machine: mpf.core.machine.MachineController*, *playfield: Playfield*)
Bases: mpf.core.mpf_controller.MpfController

Implements Ball search for a playfield device.

In MPF, the ball search functionality is attached to each playfield device, rather than being done at the global level. (In other words, each playfield is responsible for making sure no balls get stuck on it, and it leverages an instance of this BallSearch class to handle it.)

Methods & Attributes

The Ball Search has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

block(***kwargs*)
Block ball search for this playfield.

Blocking will disable ball search if it's enabled or running, and will prevent ball search from enabling if it's disabled until `ball_search_unblock()` is called.

blocked = None
If True, ball search will be blocked and will not start.

cancel_ball_search(***kwargs*)
Cancel the current ball search and mark the ball as missing.

configure_logging(*logger: str*, *console_level: str = 'basic'*, *file_level: str = 'basic'*)
Configure logging.

Parameters

- **logger** – The string name of the logger to use.
- **console_level** – The level of logging for the console. Valid options are "none", "basic", or "full".
- **file_level** – The level of logging for the console. Valid options are "none", "basic", or "full".

debug_log(*msg: str*, **args*, ***kwargs*) → None
Log a message at the debug level.

Note that whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

disable(***kwargs*)
Disable ball search.

This method will also stop the ball search if it is running.

enable(***kwargs*)
Enable the ball search for this playfield.

Note that this method does *not* start the ball search process. Rather it just resets and starts the timeout timer, as well as resetting it when playfield switches are hit.

enabled = None
Is ball search enabled.

error_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the error level.

These messages will always be shown in the console and the log file.

give_up()
Give up the ball search.

This method is called when the ball search process Did not find the missing ball. It executes the failed action which depending on the specification of *ball_search_failed_action*, either adds a replacement ball, ends the game, or ends the current ball.

ignorable_runtime_exception(*msg: str*) → None
Handle ignorable runtime exception.

During development or tests raise an exception for easier debugging. Log an error during production.

info_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the info level.

Whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

iteration = None
Current iteration of the ball search, or `False` if ball search is not started.

phase = None
Current phase of the ball search, or `False` if ball search is not started.

playfield = None
The playfield device this ball search instance is attached to.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

register(*priority*, *callback*, *name*, *, *restore_callback=None*)
Register a callback for sequential ball search.

Callbacks are called by priority. Ball search only waits if the callback returns true.

Parameters

- **priority** – priority of this callback in the ball search procedure
- **callback** – callback to call. ball search will wait before the next callback, if it returns true
- **name** – string name which is used for debugging & the logs
- **restore_callback** – optional callback to restore state of the device after ball search ended

request_to_start_game(***kwargs*)
Handle result of the *request_to_start_game* event.

If ball search is running, this method will return *False* to prevent the game from starting while ball search is running.

This method also posts the *ball_search_prevents_game_start* event if ball search is started.

reset_timer()
Reset the timeout timer which starts ball search.

This method will also cancel an actively running (started) ball search.

This is called by the playfield anytime a playfield switch is hit.

start()
Start ball search the ball search process.

started = None
Is the ball search process started (running) now.

stop()
Stop an actively running ball search.

unblock(***kwargs*)
Unblock ball search for this playfield.

This will check to see if there are balls on the playfield, and if so, enable ball search.

warning_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the warning level.

These messages will always be shown in the console and the log file.

File Manager

class mpf.core.file_manager.**FileManager**
Bases: object

Manages file interfaces.

Methods & Attributes

The File Manager has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

static get_file_interface(*filename*)
Return a file interface.

classmethod init()
Initialise file interfaces.

static load(*filename*, *verify_version=False*, *halt_on_error=True*)
Load a file by name.

static locate_file(*filename*) → str
Find a file location.

Parameters **filename** – Filename to locate

Returns: Location of file

static save(*filename*, *data*)
Save data to file.

LogMixin

class mpf.core.logging.**LogMixin**
Bases: object

Mixin class to add smart logging functionality to modules.

Methods & Attributes

The LogMixin has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

configure_logging(*logger: str*, *console_level: str = 'basic'*, *file_level: str = 'basic'*)
Configure logging.

Parameters

- **logger** – The string name of the logger to use.
- **console_level** – The level of logging for the console. Valid options are "none", "basic", or "full".
- **file_level** – The level of logging for the console. Valid options are "none", "basic", or "full".

debug_log(*msg: str*, **args*, ***kwargs*) → None
Log a message at the debug level.

Note that whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

error_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the error level.

These messages will always be shown in the console and the log file.

ignorable_runtime_exception(*msg: str*) → None
Handle ignorable runtime exception.

During development or tests raise an exception for easier debugging. Log an error during production.

info_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the info level.

Whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

warning_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the warning level.

These messages will always be shown in the console and the log file.

Mode base class

class mpf.core.mode.**Mode**(*machine: MachineController*, *config*, *name: str*, *path*)
Bases: *mpf.core.logging.LogMixin*

Base class for a mode.

Methods & Attributes

The Mode base class has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

active
Return *True* if this mode is active.

add_mode_event_handler(*event: str*, *handler: Callable*, *priority: int = 0*, ***kwargs*)
Register an event handler which is automatically removed when this mode stops.

This method is similar to the Event Manager's add_handler() method, except this method automatically unregisters the handlers when the mode ends.

Parameters

- **event** – String name of the event you're adding a handler for. Since events are text strings, they don't have to be pre-defined.
- **handler** – The method that will be called when the event is fired.

- **priority** – An arbitrary integer value that defines what order the handlers will be called in. The default is 1, so if you have a handler that you want to be called first, add it here with a priority of 2. (Or 3 or 10 or 100000.) The numbers don't matter. They're called from highest to lowest. (i.e. priority 100 is called before priority 1.)
- ****kwargs** – Any any additional keyword/argument pairs entered here will be attached to the handler and called whenever that handler is called. Note these are in addition to kwargs that could be passed as part of the event post. If there's a conflict, the event-level ones will win.

Returns A GUID reference to the handler which you can use to later remove the handler via `remove_handler_by_key`. Though you don't need to remove the handler since the whole point of this method is they're automatically removed when the mode stops.

Note that if you do add a handler via this method and then remove it manually, that's ok too.

auto_stop_on_ball_end
Controls whether this mode is stopped when the ball ends, regardless of its stop_events settings.

configure_logging(*logger: str*, *console_level: str = 'basic'*, *file_level: str = 'basic'*)
Configure logging.

Parameters

- **logger** – The string name of the logger to use.
- **console_level** – The level of logging for the console. Valid options are "none", "basic", or "full".
- **file_level** – The level of logging for the console. Valid options are "none", "basic", or "full".

configure_mode_settings(*config: dict*) → None
Process this mode's configuration settings from a config dictionary.

create_mode_devices() → None
Create new devices that are specified in a mode config that haven't been created in the machine-wide.

debug_log(*msg: str*, **args*, ***kwargs*) → None
Log a message at the debug level.

Note that whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

delay
DelayManager instance for delays in this mode. Note that all delays scheduled here will be automatically canceled when the mode stops.

error_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the error level.

These messages will always be shown in the console and the log file.

static get_config_spec() → str
Return config spec for mode_settings.

ignorable_runtime_exception(*msg: str*) → None
Handle ignorable runtime exception.

During development or tests raise an exception for easier debugging. Log an error during production.

info_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the info level.

Whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

initialise_mode() → None
Initialise this mode.

is_game_mode
Return true if this is a game mode.

load_mode_devices() → None
Load config of mode devices.

mode_init() → None
User-overrideable method which will be called when this mode initializes as part of the MPF boot process.

mode_start(***kwargs*) → None
User-overrideable method which will be called whenever this mode starts (i.e. whenever it becomes active).

mode_stop(***kwargs*) → None
User-overrideable method which will be called whenever this mode stops.

mode_will_start(***kwargs*) → None
User-overrideable method which will be called whenever this mode starts (i.e. before it becomes active).

player
Reference to the current player object.

raise_config_error(*msg*, *error_no*, *, *context=None*)
Raise a ConfigFileError exception.

restart_on_next_ball
Controls whether this mode will restart on the next ball. This only works if the mode was running when the ball ended. It's tracked per- player in the 'restart_modes_on_next_ball' player variable.

start(*mode_priority=None*, *callback=None*, ***kwargs*) → None
Start this mode.

Parameters

- **mode_priority** – Integer value of what you want this mode to run at. If you don't specify one, it will use the "Mode: priority" setting from this mode's configuration file.
- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode starts in the mode_start method which will be called automatically.

stop(*callback: Any = None*, ***kwargs*) → bool
Stop this mode.

Parameters

- **callback** – Method which will be called once this mode has stopped. Will only be called when the mode is running (includes currently stopping)
- ****kwargs** – Catch-all since this mode might start from events with who-knows-what keyword arguments.

Warning: You can safely call this method, but do not override it in your mode code. If you want to write your own mode code by subclassing Mode, put whatever code you want to run when this mode stops in the mode_stop method which will be called automatically.

Returns true if the mode is running. Otherwise false.

warning_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the warning level.

These messages will always be shown in the console and the log file.

Players

class mpf.core.player.**Player**(*machine*, *index*)
Bases: object

Base class for a player in a game.

One instance of this class is automatically created for each player.

The game mode maintains a `player` attribute which always points to the current player and is available via `self.machine.game.player`.

It also contains a `player_list` attribute which is a list of the player instances (in order) which you can use to access the non-current player.

This Player class is responsible for tracking *player variables* which is a dictionary of key/value pairs maintained on a per-player basis. There are several ways they can be used:

First, player variables can be accessed as attributes of the player object directly. For example, to set a player variable *foo* for the current player, you could use:

```
self.machine.player.foo = 0
```

If that variable didn't exist, it will be automatically created.

You can get the value of player variables by accessing them directly. For example:

```
print(self.machine.player.foo)  # prints 0
```

If you attempt to access a player variable that doesn't exist, it will automatically be created with a value of `0`.

Every time a player variable is created or changed, an MPF event is posted in the form *player_* plus the variable name. For example, creating or changing the *foo* variable will cause an event called *player_foo* to be posted.

The player variable event will have four parameters posted along with it:

- `value` (the new value)
- `prev_value` (the old value before it was updated)
- `change` (the change in the value)
- `player_num` (the player number the variable belongs to)

For the `change` parameter, it will attempt to subtract the old value from the new value. If that works, it will return the result as the change. If it doesn't work (like if you're not storing numbers in this variable), then the change parameter will be *True* if the new value is different and *False* if the value didn't change.

For examples, the following three lines:

```
self.machine.player.score = 0
self.machine.player.score += 500
self.machine.player.score = 1200
```

... will cause the following three events to be posted:

`player_score` with Args: `value=0, change=0, prev_value=0` `player_score` with Args: `value=500, change=500, prev_value=0` `player_score` with Args: `value=1200, change=700, prev_value=500`

Methods & Attributes

The Players has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

enable_events(*enable=True*, *send_all_variables=True*)
Enable/disable player variable events.

Parameters

- **enable** – Flag to enable/disable player variable events
- **send_all_variables** – Flag indicating whether or not to send an event with the current value of every player variable.

is_player_var(*var_name*)
Check if player var exists.

Parameters **var_name** – String name of the player variable to test.

Returns: *True* if the variable exists and *False* if not.

monitor_enabled = False
Class attribute which specifies whether any monitors have been registered to track player variable changes.

send_all_variable_events()
Send a player variable event for the current value of all player variables.

RGBAColor

class mpf.core.rgba_color.**RGBAColor**(*color: Union[mpf.core.rgb_color.RGBColor, str, Tuple[int, int, int], Tuple[int, int, int, int], List[int]]*)
Bases: *mpf.core.rgb_color.RGBColor*

RGB Color with alpha channel.

Methods & Attributes

The RGBAColor has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

static add_color(*name: str, color: Union[RGBColor, str, List[int], Tuple[int, int, int]]*)
Add (or updates if it already exists) a color.

Note that this is not permanent, the list is reset when MPF restarts (though you can define your own custom colors in your config file's colors: section). You *can* use this function to dynamically change the values of colors in shows (they take place the next time an LED switches to that color).

Parameters

- **name** – String name of the color you want to add/update

- **color** – The color you want to set. You can pass the same types as the RGBColor class constructor, including a tuple or list of RGB ints (0-255 each), a hex string, an RGBColor instance, or a dictionart of red, green, blue key/value pairs.

static blend(*start_color*, *end_color*, *fraction*)
Blend two colors.

Parameters

- **start_color** – The start color
- **end_color** – The end color
- **fraction** – The fraction between 0 and 1 that is used to set the blend point between the two colors.

Returns: An RGBColor object that is a blend between the start and end colors

blue
Return the blue component of the RGB color representation.

green
Return the green component of the RGB color representation.

hex
Return a 6-char HEX representation of the color.

static hex_to_rgb(*_hex: str*, *default=None*) → Tuple[int, int, int]
Convert a HEX color representation to an RGB color representation.

Parameters

- **_hex** – The 3- or 6-char hexadecimal string representing the color value.
- **default** – The default value to return if _hex is invalid.

Returns: RGB representation of the input HEX value as a 3-item tuple with each item being an integer 0-255.

name
Return the color name or None.

Returns a string containing a standard color name or None if the current RGB color does not have a standard name.

static name_to_rgb(*name: str*, *default=(0, 0, 0)*) → Tuple[int, int, int]
Convert a standard color name to an RGB value (tuple).

If the name is not found, the default value is returned. :param name: A standard color name. :param default: The default value to return if the color name is not found. :return: RGB representation of the named color. :rtype: tuple

static random_rgb() → Tuple[int, int, int]
Generate a uniformly random RGB value.

Returns A tuple of three integers with values between 0 and 255 inclusive

red
Return the red component of the RGB color representation.

rgb
Return an RGB representation of the color.

static rgb_to_hex(*rgb: Tuple[int, int, int]*) → str
Convert an RGB color representation to a HEX color representation.

(r, g, b) :: r -> [0, 255] g -> [0, 255] b -> [0, 255]

Parameters **rgb** – A tuple of three numeric values corresponding to the red, green, and blue value.

Returns HEX representation of the input RGB value.

Return type str

rgba
Return an RGB representation of the color.

static string_to_rgb(*value: str*, *default=(0, 0, 0)*) → Tuple[int, int, int]
Convert a string which could be either a standard color name or a hex value to an RGB value (tuple).

If the name is not found and the supplied value is not a valid hex string it raises an error. :param value: A standard color name or hex value. :param default: The default value to return if the color name is not found and the supplied value is not a valid hex color string. :return: RGB representation of the named color. :rtype: tuple

RGBColor

class mpf.core.rgb_color.**RGBColor**(*color: Union[RGBColor, str, List[int], Tuple[int, int, int]] = None*)
Bases: object

One RGB Color.

Methods & Attributes

The RGBColor has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

static add_color(*name: str, color: Union[RGBColor, str, List[int], Tuple[int, int, int]]*)
Add (or updates if it already exists) a color.

Note that this is not permanent, the list is reset when MPF restarts (though you can define your own custom colors in your config file's colors: section). You *can* use this function to dynamically change the values of colors in shows (they take place the next time an LED switches to that color).

Parameters

- **name** – String name of the color you want to add/update
- **color** – The color you want to set. You can pass the same types as the RGBColor class constructor, including a tuple or list of RGB ints (0-255 each), a hex string, an RGBColor instance, or a dictionart of red, green, blue key/value pairs.

static blend(*start_color*, *end_color*, *fraction*)
Blend two colors.

Parameters

- **start_color** – The start color
- **end_color** – The end color

- **fraction** – The fraction between 0 and 1 that is used to set the blend point between the two colors.

Returns: An RGBColor object that is a blend between the start and end colors

blue
Return the blue component of the RGB color representation.

green
Return the green component of the RGB color representation.

hex
Return a 6-char HEX representation of the color.

static hex_to_rgb(*_hex: str*, *default=None*) → Tuple[int, int, int]
Convert a HEX color representation to an RGB color representation.

Parameters

- **_hex** – The 3- or 6-char hexadecimal string representing the color value.
- **default** – The default value to return if _hex is invalid.

Returns: RGB representation of the input HEX value as a 3-item tuple with each item being an integer 0-255.

name
Return the color name or None.

Returns a string containing a standard color name or None if the current RGB color does not have a standard name.

static name_to_rgb(*name: str*, *default=(0, 0, 0)*) → Tuple[int, int, int]
Convert a standard color name to an RGB value (tuple).

If the name is not found, the default value is returned. :param name: A standard color name. :param default: The default value to return if the color name is not found. :return: RGB representation of the named color. :rtype: tuple

static random_rgb() → Tuple[int, int, int]
Generate a uniformly random RGB value.

Returns A tuple of three integers with values between 0 and 255 inclusive

red
Return the red component of the RGB color representation.

rgb
Return an RGB representation of the color.

static rgb_to_hex(*rgb: Tuple[int, int, int]*) → str
Convert an RGB color representation to a HEX color representation.

(r, g, b) :: r -> [0, 255] g -> [0, 255] b -> [0, 255]

Parameters **rgb** – A tuple of three numeric values corresponding to the red, green, and blue value.

Returns HEX representation of the input RGB value.

Return type str

static string_to_rgb(*value: str*, *default=(0, 0, 0)*) → Tuple[int, int, int]
Convert a string which could be either a standard color name or a hex value to an RGB value (tuple).

If the name is not found and the supplied value is not a valid hex string it raises an error. :param value: A standard color name or hex value. :param default: The default value to return if the color name is not found and the supplied value is not a valid hex color string. :return: RGB representation of the named color. :rtype: tuple

Randomizer

class mpf.core.randomizer.**Randomizer**(*items*)
Bases: object

Generic list randomizer.

Methods & Attributes

The Randomizer has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

get_current()
Return current item.

get_next()
Return next item.

loop
Return loop property.

static pick_weighted_random(*items*)
Pick a random item.

Parameters **items** – Items to select from

Timers

Utility Functions

class mpf.core.utility_functions.**Util**
Bases: object

Utility functions for MPF.

Methods & Attributes

The Utility Functions has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

static any(*futures: Iterable[asyncio.futures.Future], loop, timeout=None*)
Return first future.

static bin_str_to_hex_str(*source_int_str: str*, *num_chars: int*) → str
Convert binary string to hex string.

static cancel_futures(*futures: Iterable[asyncio.futures.Future]*)
Cancel futures.

static chunker(*l*, *n*)
Yield successive n-sized chunks from l.

static convert_to_simply_type(*value*)
Convert value to a simple type.

static convert_to_type(*value*, *type_name*)
Convert value to type.

static db_to_gain(*db: float*) → float
Convert a value in decibels (-inf to 0.0) to a gain (0.0 to 1.0).

Parameters **db** – The decibel value (float) to convert to a gain

Returns Float

static dict_merge(*a*, *b*, *combine_lists=True*)
Recursively merge dictionaries.

Used to merge dictionaries of dictionaries, like when we're merging together the machine configuration files. This method is called recursively as it finds sub-dictionaries.

For example, in the traditional python dictionary update() methods, if a dictionary key exists in the original and merging-in dictionary, the new value will overwrite the old value.

Consider the following example:

Original dictionary: *config['foo']['bar'] = 1*

New dictionary we're merging in: *config['foo']['other_bar'] = 2*

Default python dictionary update() method would have the updated dictionary as this:

{'foo': {'other_bar': 2}}

This happens because the original dictionary which had the single key *bar* was overwritten by a new dictionary which has a single key *other_bar*.)

But really we want this:

{'foo': {'bar': 1, 'other_bar': 2}}

This code was based on this: https://www.xormedia.com/recursively-merge-dictionaries-in-python/

Parameters

- **a** (*dict*) – The first dictionary
- **b** (*dict*) – The second dictionary
- **combine_lists** (*bool*) – Controls whether lists should be combined (extended) or overwritten. Default is *True* which combines them.

Returns The merged dictionaries.

static ensure_future(*coro_or_future*, *loop*)
Wrap ensure_future.

static event_config_to_dict(*config*)
Convert event config to a dict.

static first (*futures: Iterable[asyncio.futures.Future], loop, timeout=None, cancel_others=True*)
Return first future and cancel others.

static get_from_dict (*dic*, *key_path*)
Get a value from a nested dict (or dict-like object) from an iterable of key paths.

Parameters

- **dic** – Nested dict of dicts to get the value from.
- **key_path** – iterable of key paths

Returns value

This code came from here: http://stackoverflow.com/questions/14692690/access-python-nested-dictionary-items-via-a-list-of-keys

static get_named_list_from_objects (*switches*) → List[str]
Return a list of names from a list of switch objects.

static hex_string_to_int (*inputstring: str*, *maxvalue: int = 255*) → int
Take a string input of hex numbers and an integer.

Parameters

- **inputstring** – A string of incoming hex colors, like ffff00.
- **maxvalue** – Integer of the max value you'd like to return. Default is 255. (This is the real value of why this method exists.)

Returns Integer representation of the hex string.

static hex_string_to_list (*input_string*, *output_length=3*)
Take a string input of hex numbers and return a list of integers.

This always groups the hex string in twos, so an input of ffff00 will be returned as [255, 255, 0]

Parameters

- **input_string** – A string of incoming hex colors, like ffff00.
- **output_length** – Integer value of the number of items you'd like in your returned list. Default is 3. This method will ignore extra characters if the input_string is too long, and it will pad the left with zeros if the input string is too short.

Returns List of integers, like [255, 255, 0]

Raises ValueError if the input string contains non-hex chars –

static int_to_hex_string (*source_int: int*) → str
Convert an int from 0-255 to a one-byte (2 chars) hex string, with uppercase characters.

static is_hex_string (*string: str*) → bool
Return true if string is hex.

static is_power2 (*num: int*) → bool
Check a number to see if it's a power of two.

Parameters num – The number to check

Returns: True or False

static keys_to_lower (*source_dict*)
Convert the keys of a dictionary to lowercase.

Parameters source_dict – The dictionary you want to convert.

Returns A dictionary with lowercase keys.

static list_of_lists(*incoming_string*)
Convert an incoming string or list into a list of lists.

static normalize_hex_string(*source_hex: str*, *num_chars: int = 2*) → str
Take an incoming hex value and convert it to uppercase and fills in leading zeros.

Parameters

- **source_hex** – Incoming source number. Can be any format.
- **num_chars** – Total number of characters that will be returned. Default is two.

Returns String, uppercase, zero padded to the num_chars.

Example usage: Send "c" as source_hex, returns "0C".

static power_to_on_off(*power: float*, *max_period: int = 20*) → Tuple[int, int]
Convert a float value to on/off times.

static pwm32_to_hex_string(*source_int: int*) → str
Convert a PWM32 value to hex.

static pwm32_to_int(*source_int: int*) → int
Convert a PWM32 value to int.

static pwm8_to_hex_string(*source_int: int*) → str
Convert an int to a PWM8 string.

static pwm8_to_int(*source_int: int*) → int
Convert a PWM8 value to int.

static race(*futures: Dict[asyncio.futures.Future, str], loop*)
Return key of first future and cancel others.

static set_in_dict(*dic*, *key_path*, *value*)
Set a value in a nested dict-like object based on an iterable of nested keys.

Parameters

- **dic** – Nested dict of dicts to set the value in.
- **key_path** – Iterable of the path to the key of the value to set.
- **value** – Value to set.

static string_to_class(*class_string: str*) → Callable[..., Any]
Convert a string like mpf.core.events.EventManager into a Python class.

Parameters **class_string** (*str*) – The input string

Returns A reference to the python class object

This function came from here: http://stackoverflow.com/questions/452969/does-python-have-an-equivalent-to-java-class-forname

static string_to_gain(*gain_string: str*) → float
Convert string to gain.

Decode a string containing either a gain value (0.0 to 1.0) or a decibel value (-inf to 0.0) into a gain value (0.0 to 1.0).

Parameters **gain_string** – The string to convert to a gain value

Returns Float containing a gain value (0.0 to 1.0)

static string_to_list(*string: Union[str, List[str], None]*) → List[str]
Convert a comma-separated and/or space-separated string into a Python list.

Parameters **string** – The string you'd like to convert.

Returns A python list object containing whatever was between commas and/or spaces in the string.

static string_to_lowercase_list(*string: str*) → List[str]
Convert a comma-separated and/or space-separated string into a Python list.

Each item in the list has been converted to lowercase.

Parameters **string** – The string you'd like to convert.

Returns A python list object containing whatever was between commas and/or spaces in the string, with each item converted to lowercase.

static string_to_ms(*time_string: str*) → int
Decode a string of real-world time into an int of milliseconds.

Example inputs:

200ms 2s None

If no "s" or "ms" is provided, this method assumes "milliseconds."

If time is 'None' or a string of 'None', this method returns 0.

Returns Integer. The examples listed above return 200, 2000 and 0, respectively

static string_to_secs(*time_string: str*) → float
Decode a string of real-world time into an float of seconds.

See 'string_to_ms' for a description of the time string.

data_manager

class mpf.core.data_manager.**DataManager**(*machine*, *name*, *min_wait_secs=1*)
Bases: mpf.core.mpf_controller.MpfController

Handles key value data loading and saving for the machine.

Methods & Attributes

The data_manager has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

configure_logging(*logger: str*, *console_level: str = 'basic'*, *file_level: str = 'basic'*)
Configure logging.

Parameters

- **logger** – The string name of the logger to use.
- **console_level** – The level of logging for the console. Valid options are "none", "basic", or "full".
- **file_level** – The level of logging for the console. Valid options are "none", "basic", or "full".

debug_log(*msg: str*, **args*, ***kwargs*) → None
Log a message at the debug level.

Note that whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

error_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the error level.

These messages will always be shown in the console and the log file.

get_data(*section=None*)
Return the value of this DataManager's data.

Parameters **section** – Optional string name of a section (dictionary key) for the data you want returned. Default is None which returns the entire dictionary.

ignorable_runtime_exception(*msg: str*) → None
Handle ignorable runtime exception.

During development or tests raise an exception for easier debugging. Log an error during production.

info_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the info level.

Whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

raise_config_error(*msg*, *error_no*, ***, *context=None*)
Raise a ConfigFileError exception.

save_all(*data*)
Update all data.

warning_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the warning level.

These messages will always be shown in the console and the log file.

delay_manager

class mpf.core.delays.**DelayManager**(*registry: mpf.core.delays.DelayManagerRegistry*)
Bases: mpf.core.mpf_controller.MpfController

Handles delays for one object.

By default, a machine-wide instance is created and available via `self.machine.delay`.

Individual modes also have Delay Managers which can be accessed in mode code via `self.delay`. (Delays in mode-based delay managers are automatically removed when the mode stops.)

Methods & Attributes

The delay_manager has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

add(*ms: int, callback: Callable[..., None], name: str = None, **kwargs*) → str
Add a delay.

Parameters

- **ms** – The number of milliseconds you want this delay to be for.
- **callback** – The method that is called when this delay ends.
- **name** – String name of this delay. This name is arbitrary and only used to identify the delay later if you want to remove or change it. If you don't provide it, a UUID4 name will be created.
- ****kwargs** – Any other (optional) kwarg pairs you pass will be passed along as kwargs to the callback method.

Returns String name or UUID4 of the delay which you can use to remove it later.

add_if_doesnt_exist(*ms: int, callback: Callable[..., None], name: str, **kwargs*) → str
Add a delay only if a delay with that name doesn't exist already.

Parameters

- **ms** – Int of the number of milliseconds you want this delay to be for.
- **callback** – The method that is called when this delay ends.
- **name** – String name of this delay. This name is arbitrary and only used to identify the delay later if you want to remove or change it.
- ****kwargs** – Any other (optional) kwarg pairs you pass will be passed along as kwargs to the callback method.

Returns String name of the delay which you can use to remove it later.

check(*delay: str*) → bool
Check to see if a delay exists.

Parameters **delay** – A string of the delay you're checking for.

Returns True if the delay exists. False otherwise.

clear() → None
Remove (clear) all the delays associated with this DelayManager.

configure_logging(*logger: str*, *console_level: str = 'basic'*, *file_level: str = 'basic'*)
Configure logging.

Parameters

- **logger** – The string name of the logger to use.
- **console_level** – The level of logging for the console. Valid options are "none", "basic", or "full".
- **file_level** – The level of logging for the console. Valid options are "none", "basic", or "full".

debug_log(*msg: str*, **args*, ***kwargs*) → None
Log a message at the debug level.

Note that whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

error_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the error level.

These messages will always be shown in the console and the log file.

ignorable_runtime_exception(*msg: str*) → None
Handle ignorable runtime exception.

During development or tests raise an exception for easier debugging. Log an error during production.

info_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the info level.

Whether this message shows up in the console or log file is controlled by the settings used with configure_logging().

raise_config_error(*msg*, *error_no*, ***, *context=None*)
Raise a ConfigFileError exception.

remove(*name: str*)
Remove a delay by name.

Removing a delay prevents the callback from being called and cancels the delay.

Parameters **name** – String name of the delay you want to remove. If there is no delay with this name, that's ok. Nothing happens.

reset(*ms: int, callback: Callable[..., None], name: str, **kwargs*) → str
Reset a delay.

Resetting will first delete the existing delay (if it exists) and then add new delay with the new settings. If the delay does not exist, that's ok, and this method is essentially the same as just adding a delay with this name.

Parameters

- **ms** – The number of milliseconds you want this delay to be for.
- **callback** – The method that is called when this delay ends.
- **name** – String name of this delay. This name is arbitrary and only used to identify the delay later if you want to remove or change it. If you don't provide it, a UUID4 name will be created.
- ****kwargs** – Any other (optional) kwarg pairs you pass will be passed along as kwargs to the callback method.

Returns String name or UUID4 of the delay which you can use to remove it later.

run_now(*name: str*)
Run a delay callback now instead of waiting until its time comes.

This will cancel the future running of the delay callback.

Parameters **name** – Name of the delay to run. If this name is not an active delay, that's fine. Nothing happens.

warning_log(*msg: str*, **args*, *context=None*, ***kwargs*) → None
Log a message at the warning level.

These messages will always be shown in the console and the log file.

delay_manager_registry

class mpf.core.delays.**DelayManagerRegistry**(*machine: MachineController*)
Bases: object

Keeps references to all DelayManager instances.

Methods & Attributes

The delay_manager_registry has the following methods & attributes available. Note that methods & attributes inherited from the base class are not included here.

add_delay_manager(*delay_manager: mpf.core.delays.DelayManager*) → None
Add a delay manager to the list.

Parameters **delay_manager** – The *DelayManager* instance you're adding to this registry.

8.4 Common functions to use in your code

- *Machine Variables in Code*
- *Player Variables in Code*

8.4.1 Machine Variables in Code

You can use machine variables by calling into the MPF machine.

```
# read machine variable
print(self.machine.get_machine_var("my_variable"))

# configure variable to persist to disk and expire after 1 day (optional)
# alternatively you can also use "machine_vars" in config to achieve the same
self.machine.configure_machine_var("my_variable", persist=True, expire_secs=86400)

# set a variable
self.machine.set_machine_var("my_variable", 17)
```

8.4.2 Player Variables in Code

Player variables are only accessible when a game is running. Be prepared that the current player may change in a multiplayer game.

Inside a (game) mode you can access the current player using `self.player`. Alternatively, you can use `self.machine.game.player` but be aware that both `self.machine.game` and `self.machine.game.player` may be `None`.

You can use player variables like this:

```
player = self.machine.game.player     # do not persist the player because it may change
                                      # alternatively use self.player in modes

if not player:
    return     # do something reasonable here but do not crash in the next line
```

(continues on next page)

(continued from previous page)

```
# read player variable
print(player["my_variable"])

# set a variable
player["my_variable"] = 17
```

8.5 Automated Testing

The MPF dev team are strong believers in automated testing, and we use a test-driven development (TDD) process for developing MPF itself. (At the time of this writing, there are over 800 unit tests for MPF and MPF-MC, each which contain dozens of individual tests.)

We have extended Python's built-in unittest TestCase class for MPF-specific tests, including mocking critical internal elements and adding assertion methods for MPF features.

You can run built-in tests to test MPF itself or extend them if you think you found a bug or if you're adding features to MPF. We have also built TestCase classes you can use to write unittests for your own machine. Read on for details:

8.5.1 How to run MPF unittests

Once MPF is installed, you can run some automated tests to make sure that everything is working. To do this, open a command prompt, and then type the following command and then press <enter>:

```
python3 -m unittest discover mpf/tests
```

When you do this, you should see a bunch of dots on the screen (one for each test that's run), and then when it's done, you should see a message showing how many tests were run and that they were successful. The whole process should take less a minute or so.

(If you see any messages about some tests taking more than 0.5s, that's ok.)

The important thing is that when the tests are done, you should have a message like this:

```
Ran 587 tests in 27.121s

OK

C:\>
```

Note that the number of tests is changing all the time, so it probably won't be exactly 587. And also the time they took to run will be different depending on how fast your computer is.

These tests are the actual tests that the developers of MPF use to test MPF itself. We wrote all these tests to make sure that updates and changes we add to MPF don't break things. :) So if these tests pass, you know your MPF installation is solid.

Remember though that MPF is actually two separate parts, the MPF game engine and the MPF media controller. The command you run just tested the game engine, so now let's test the media controller. To do this, run the following command (basically the same thing as last time but with an "mc" added to the end, like this):

```
python3 -m unittest discover mpfmc/tests
```

(Note that `mpfmc` does not have a dash in it, like it did when you installed it via *pip*.)

When you run the MPF-MC tests, you should see a graphical window pop up on the screen, and many of the tests will put graphics and words in that window. Also, some of the tests include audio, so if your speakers are on you should hear some sounds at some point.

These tests take significantly longer (maybe 8x) than the MPF tests, but when they're done, that graphical window should close, and you'll see all the dots in your command window and a note that all the tests were successful.

Notes about the MPF-MC tests:

- These tests create a window on the screen and then just re-use the same window for all tests (to save time). So don't worry if it looks like the window content is scaled weird or blurry or doesn't fill the entire window.
- Many of these tests are used to test internal workings of the media controller itself, so there will be lots of time when the pop up window is blank or appears frozen since the tests are testing non-visual things.
- The animation and transition tests include testing functionality to stop, restart, pause, and skip frames. So if things look "jerky" in the tests, don't worry, that doesn't mean your computer is slow, it's just how the tests work! :)

8.5.2 Writing Unit Tests for MPF

todo

8.5.3 Writing Custom Tests for your Machine

As we already mentioned, the creators of MPF are HUGE believers in the value of automated testing. To that end, MPF includes everything you need to write automated tests that test the logical functionality of your machine. These tests are extremely valuable *even if your game is just based on config files*.

For example, you can write a test that simulates starting a game, launching a ball, hitting a sequence of switches, and then verifying that a certain mode is running, or a light is the right color, or an achievement group is in the proper state, etc. Then you can advance the time to timeout a mode and verify that the mode as stopped, etc, etc.

When you first start building your MPF config, you might think, "What's the point?"... especially with some of the more simple tests. However your MPF config files will get complex pretty quickly, and often times you'll think you have some mode done and working perfectly, but then a month later you change something that seems unrelated which ends up breaking it. Unfortunately this usually happens without you knowing it, and by the time you realize that something broke, more times has passed and it's hard to figure out what broke what.

So this is where unit tests come in! :)

If you write simple unit tests that test each new thing you add to an MPF config file, then over time you'll end up with a huge library of tests for your game. If you get in the habit of running your tests often, then you'll know right away if a change that you made broke something. (And you'll also know when everything is ok when all your tests pass again!)

Tutorial for writing your own tests

We have a complete tutorial which walks you through writing tests for your own machine. This tutorial conveniently follows the general MPF tutorial at docs.missionpinball.org. Each step here matches the step with the same number there. (Just make sure you'll looking at the same version of the documentation in both places.)

In the general MPF tutorial, each step builds on the previous to add more features to the config files for the tutorial project. In the unit test tutorial (what you're reading here), each step shows you how to write the unit tests which test the new features you just added to the tutorial machine.

You can follow along and learn here:

Testing Tutorial Step 1. Installing MPF

Step 1 of the MPF Tutorial is just about getting MPF installed, so there's nothing to do yet for testing. Really we just include Step 1 here because if we didn't it would be confusing as to why the tutorial starts on Step 2. :)

Note that if you download the MPF Example repo, the tutorial folder in there which contains all the complete tutorial files for every step also includes test folders with all the complete tests.

Testing Tutorial Step 2. Create your machine folder

Step 2 of the MPF Tutorial is where you create your machine folder and get MPF up and running with an empty config. Since it ends with MPF running and the attract mode being active, we can actually write a test for it!

Here are the steps to take:

1. Create a "tests" folder in your machine folder

First, create a folder called `tests` in your machine folder. This would be alongside the other folders in there, which will be "config" (created in the MPF tutorial), as well as "logs" and "data" which were created automatically by MPF the first time it ran.

2. Add an empty "__init__.py" file

Next, inside your new *tests* folder, create a blank file called `__init__.py`. (That's two underscores, then the word "init", then two more underscores, then ".py".) This file should be totally blank. (It just needs to exist.) This file is needed to let the Python test runner find and load the tests from this folder.

3. Add a test file

Next you need to add a Python file which actually holds your tests. You can name this file whatever you want as long as it starts with "test". (The reason for starting it with "test" is also so that the Python test runner knows that this file contains tests, allowing it to automatically find and run tests from it.)

For now let's call it `test_step_2.py`.

Open that file and add the following lines to it: (If you are interested in what all this means, then read on below the file. Otherwise you can skip down to Step 4.)

```
"""Contains tests for Step 2 of the MPF tutorial."""

from mpf.tests.MpfMachineTestCase import MpfMachineTestCase


class TestTutorialMachine(MpfMachineTestCase):

    """Contains tests for the MPF machine config"""

    def test_step_2_mpf_startup(self):
        """Tests Step 2 of the tutorial"""

        # At this point, the machine config is blank, which means that other
        # than MPF starting and the attract mode running, nothing is really
        # happening. So let's just check that the attract mode is running and
        # that's it.

        self.assertModeRunning('attract')

        # asserts that a mode called 'attract' is running, and fails the test
        # if not.
```

So what's this file actually doing?

The import line just imports the base class we use for MPF machine tests. (More details on that is covered in the *Testing Class API page*).

Our specific class name `TestTutorialMachine` can be whatever you want. Again just make sure it starts with "Test" in order for the test runner to find out.

Our specific method is called `test_step_2_mpf_startup()`. (Also it has to start with "test"). When the tests are run each method represents a separate "run" of MPF. The test runner will start up MPF and get it all up and running, and then it will move through the code in the test method, then it will cleanly shut down MPF when it's done. If there are multiple test methods, then the test running will start and stop MPF multiple times. The key is that each test method is run against a "fresh" MPF copy.

These test methods will also load the machine config files (just like if the command `mpf` was run the regular way).

Anyway, in our test method, we have the only actual line that does anything:

```
self.assertModeRunning('attract')
```

This just tests ("asserts") that a mode called "attract" is running. There are all sorts of MPF-specific assertion methods which we'll cover in later steps of this tutorial.

4. Run your test

You can run your tests via the command prompt from your machine folder. (In other words, the same place where you run `mpf` to run your machine.)

The exact command to run is `python -m unittest`. This should produce output similar to the following:

```
C:\pinball\your_machine>python -m unittest
C:\Python34\lib\imp.py:32: PendingDeprecationWarning: the imp module is deprecated in␣
↪favour of importlib; see the module's documentation for alternative uses
  PendingDeprecationWarning)
.
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------------------------------
Ran 1 test in 0.734s

OK

C:\pinball\your_machine>
```

That warning about the deprecation can be ignored (if you even have it.. you might not). The important thing is the message towards the bottom: "Ran 1 test in 0.734s" and the "OK" below it. That means your test passed!

5. Check out a failed test

When you're writing unit tests, you'll end up dealing with failed tests a lot! So let's purposefully change the test so it fails. In this case, change the line which asserts a mode called "attract" is running to look for a mode called "foo" instead, like this:

```
self.assertModeRunning('foo')
```

Save the file and rerun the tests and you should see results like this:

```
C:\pinball\your_machine>python -m unittest
C:\Python34\lib\imp.py:32: PendingDeprecationWarning: the imp module is deprecated in␣
↪favour of importlib; see the module's documentation for alternative uses
  PendingDeprecationWarning)
F
======================================================================
FAIL: test_mpf_starts (tests.test_step_2.TestTutorialMachine)
Tests Step 2 of the tutorial
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\pinball\your_machine\tests\test_step_2.py", line 18, in test_mpf_starts
    self.assertModeRunning('foo')
  File "C:\Python34\lib\site-packages\mpf\tests\MpfTestCase.py", line 576, in␣
↪assertModeRunning
    raise AssertionError("Mode {} not known.".format(mode_name))
AssertionError: Mode foo not known.

----------------------------------------------------------------------
Ran 1 test in 0.594s

FAILED (failures=1)

C:\pinball\your_machine>
```

Note that we see the test run failed, with one failure, and that we can scroll up and see the specific name of the test that failed along with the line that failed, and information about the failure. (In this case it tells us that the mode "foo" is not known.)

So to get this test to work, you either need to change your MPF config to start a mode called "foo", or you need to change the test back to looking for a mode called "attract". :)

What if it didn't work?

If the unit tests don't work for you, there are a few things you can try.

If you get some kind of loading error or config error, make sure you're running `python -m unittest` from your machine folder (not from the "tests" folder).

If you get a message about 0 tests run, make sure you have that empty `__init__.py` in your tests folder.

And if you get some weird error that you can't figure out, then post a message to the MPF Google Group.

8.5.4 Fuzz Testing

todo

8.6 Extending MPF

These guides explain how to setup a dev environment for extending and adding to MPF itself, and how to add various components to MPF.

8.6.1 Setting up your MPF Dev Environment

If you want to work on the core MPF or MPF-MC code, you have to install MPF and MPF-MC a bit differently than the normal process.

Why? Because normally when you install MPF and MPF-MC via *pip*, they get installed as Python packages into your `Python/Lib/site-packages` folder, and that location is not too conducive to editing MPF source code since it's in a deep random location. Also, if you ever ran *pip* again to update your MPF installation, you would potentially overwrite any changes you made.

Instead, you need to install MPF and MPF-MC in "developer" (also known as "editable") mode. This mode will let you run MPF and MPF-MC from the folder of your choice, and will allow code changes or additions you make to be immediately available whenever you run MPF.

1. Install a git client

MPF is cross-platform and runs the same on Mac, Windows, or Linux. So any changes or additions you make should work on all platforms.

If you're on Windows or Mac, the easiest way to get a git client installed is to use the GitHub Desktop app. This app will also install the git command line tools.

2. Clone the MPF and/or MPF-MC repo(s)

Clone the mpf repository and its submodules :

```
git clone --recursive https://github.com/missionpinball/mpf.git
```

Same thing for the mpf-mc repository :

```
git clone --recursive https://github.com/missionpinball/mpf-mc.git
```

If you're using the GitHub Desktop app, you can also browse to the repos on GitHub and click the green "Clone or Download" button, and then click the "Open in Desktop" link. That will pop up a box that prompts you to pick a folder for the local codebase.

Then inside that folder, you'll end up with an `mpf` folder for MPF and `mpf-mc` folder for MPF-MC.

3. Install MPF / MPF-MC in "developer" mode

Create a "virtualenv" for your MPF development in a mpf-env directory (Note : if you don't have virtualenv installed, you can get it via pip by running `pip3 install virtualenv`.

Using virtualenv lets you keep all the other Python packages MPF needs (pyserial, pyyaml, kivy, etc.) together in a "virtual" environment that you'll use for MPF and helps keep everything in your Python environment cleaner in general.

Create a new virtualenv called "mpf-venv" (or whatever you want to name it) like this:

```
virtualenv -p python3 mpf-venv
```

Then enter the newly-created virtualenv:

```
source mpf-venv/bin/activate
```

Each time you'll work with your MPF development version you'll have to switch to this environment. Note: in this environment, thanks to the "-p python3" option of virtualenv, the version of Python and pip is 3.x automatically.

Next you'll install MPF and MPF-MC. This is pretty much like a regular install, except that you'll also use the `-e` command line option which means these packages will be installed in "editable" mode.

Install mpf and mpf-mc like this:

```
pip install -e mpf
pip install -e mpf-mc
```

You should now be done, and you can verify that everyething is installed properly via:

```
mpf --version
```

Note : you could also install mpf and mpf-mc in your global environment using `sudo pip3 install -e mpf` and `sudo pip3 install -e mpf-mc`, or in your user environment using `pip3 install --user -e mpf` and `pip3 install --user -e mpf-mc`.

4. Make your changes

Be sure to add your name to the `AUTHORS` file in the root of the MPF or MPF-MC repo!

5. Write / update unit tests

We make heavy use of unit tests to ensure that future changes don't break existing functionality. So write new unit tests to cover whatever you just wrote, and be sure to rerun all the unit tests to make sure your changes or additions didn't break anything else.

More information on creating and running MPF unit tests is *here*.

6. Submit a pull request

If your change fixes an open issue, reference that issue number in the comments, like "fixes #123".

8.6.2 Writing Plugins for MPF

todo

8.6.3 Developing your own hardware interface for MPF

todo

8.6.4 Annotating events for MPF docs

You usually write the following to post an event in code:

```
# this event is posted when something awesome happens
self.machine.events.post("your_awesome_event", reason=what_happened)
```

This will work. However, nobody will know about your shiny new event. Therefore, we want to document it for our users. Since it would be dublicate work to document the event in code and in the docs, we use a custom docblock annotation:

```
self.machine.events.post("your_awesome_event", reason=what_happened)
'''event: your_awesome_event

desc: This event is posted when something awesome happens. We suggest that
you play a loud sound and show some flashy slides when this happens.

args:
  reason: The reason for this awesomeness is stated here.
'''
```

The event will be automatically added to the event reference on the next update of the documentation.

8.7 BCP Protocol Specification

This document describes the Backbox Control Protocol, (or "BCP"), a simple, fast protocol for communications between an implementation of a pinball game controller and a multimedia controller.

Note: BCP is how the MPF core engine and the MPF media controller communicate.

BCP transmits semantically relevant information and attempts to isolate specific behaviors and identifiers on both sides. i.e., the pin controller is responsible for telling the media controller "start multiball mode". The pin controller

doesn't care what the media controller does with that information, and the media controller doesn't care what happened on the pin controller that caused the multiball mode to start.

BCP is versioned to prevent conflicts. Future versions of the BCP will be designed to be backward compatible to every degree possible. The reference implementation uses a raw TCP socket for communication. On localhost the latency is usually sub-millisecond and on LANs it is under 10 milliseconds. That means that the effect of messages is generally under 1/100th of a second, which should be considered instantaneous from the perspective of human perception.

It is important to note that this document specifies the details of the protocol itself, not necessarily the behaviors of any specific implementations it connects. Thus, there won't be details about fonts or sounds or images or videos or shaders here; those are up to specific implementation being driven.

Warning: Since the pin controller and media controller are both state machines synchronized through the use of commands, it is possible for the programmer to inadvertently set up infinite loops. These can be halted with the "reset" command or "hello" described below.

8.7.1 Background

While the BCP protocol was created as part of the MPF project, the intention is that BCP is an open protocol that could connect *any* pinball controller to *any* media controller.

8.7.2 Protocol Format

- Commands are human-readable text in a format similar to URLs, e.g. `command?parameter1=value&parameter2=value`
- Command characters are encoded with the utf-8 character encoding. This allows ad-hoc text for languages that use characters past ASCII-7 bit, such as Japanese Kanji.
- Command and parameter names are whitespace-trimmed on both ends by the recipient
- Commands are case-insensitive
- Parameters are optional. If present, a question mark separates the command from its parameters
- Parameters are in the format `name=value`
- Parameter names are case-insensitive
- Parameter values are case-sensitive
- Simple parameter values are prefixed with a string that indicates their data type: (`int:`, `float:`, `bool:`, `NoneType:`). For example, the integer 5 would appear in the command string as `int:5`.
- When a command includes one or more complex value types (list or dict) all parameters are encoded using JSON and the resulting encoded value is assigned to the `json:` parameter.
- Parameters are separated by an ampersand (`&`)
- Parameter names and their values are escaped using percent encoding as necessary; (details here).
- Commands are terminated by a line feed character (`\n`). Carriage return characters (`\r`) should be tolerated but are not significant.
- A blank line (no command) is ignored
- Commands beginning with a hash character (`#`) are ignored
- If a command passes unknown parameters, the recipient should ignore them.

- The pinball controller and the media controller must be resilient to network problems; if a connection is lost, it can simply re-open it to resume operation. There is no requirement to buffer unsendable commands to transmit on reconnection.
- Once initial handshaking has completed on the first connection, subsequent re-connects do not have to handshake again.
- An unrecognized command results in an error response with the message "unknown command"

In all commands referenced below, the `\n` terminator is implicit. Some characters in parameters such as spaces would really be encoded as `%20` (space) in operation, but are left unencoded here for clarity.

8.7.3 Initial Handshake

When a connection is initially established, the pinball controller transmits the following command:

```
hello?version=1.0
```

. . . where *1.0* is the version of the Backbox protocol it wants to speak. The media controller may reply with one of two responses:

```
hello?version=1.0
```

. . . indicating that it can speak the protocol version named, and reporting the version it speaks, or

```
error?message=unknown protocol version
```

. . . indicating that it cannot. How the pin controller handles this situation is implementation-dependent.

8.7.4 BCP commands

The following BCP commands have been defined (and implemented) in MPF:

ball_end (BCP command)

Indicates the ball has ended. Note that this does not necessarily mean that the next player's turn will start, as this player may have an extra ball which means they'll shoot again.

Origin

Pin controller

Parameters

None

Response

None

ball_start (BCP command)

Indicates a new ball has started. It passes the player number (1, 2, etc.) and the ball number as parameters. This command will be sent every time a ball starts, even if the same player is shooting again after an extra ball.

Origin

Pin controller

Parameters

player_num

Type: `int`

The player number.

ball

Type: `int`

The ball number.

Response

None

device (BCP command)

Origin

Pin controller or media controller

Parameters

type

Type: `string`

The type/class of device (ex: coil).

name

Type: `string`

The name of the device.

changes

Type: `tuple` (attribute name, old value, new value)

The change to the device state.

state

Type: varies (depending upon device type)

The device state.

Response

None

error (BCP command)

This is a command used to convey error messages back to the origin of a command.

Origin

Pin controller or media controller

Parameters

message

Type: `string`

The error message.

command

Type: `string`

The command that was invalid and caused the error.

Response

None

goodbye (BCP command)

Lets one side tell the other than it's shutting down.

Origin

Pin controller or media controller

Parameters

None

Response

None

hello (BCP command)

This is the initial handshake command upon first connection. It sends the BCP protocol version that the origin controller speaks.

Origin

Pin controller or media controller

Parameters

version

Type: `string`

The BCP communication specification version implemented in the controller (ex: 1.0).

controller_name

Type: `string`

The name of the controller (ex: Mission Pinball Framework).

controller_version

Type: `string`

The version of the controller (ex: 0.33.0).

Response

When received by the media controller, this command automatically triggers a hard "reset". If the pin controller is sending this command, the media controller will respond with either its own "hello" command, or the error "unknown protocol version." The pin controller should never respond to this command when it receives it from the media controller; that would trigger an infinite loop.

machine_variable (BCP command)

This is a generic "catch all" which sends machine variables to the media controller any time they change. Machine variables are like player variables, except they're maintained machine-wide instead of per-player or per-game. Since the pin controller will most likely track hundreds of variables (with many being internal things that the media controller doesn't care about), it' s recommended that the pin controller has a way to filter which machine variables are sent to the media controller.

Origin

Pin controller

Parameters

name

Type: `string`

This is the name of the machine variable.

value

Type: Varies depending upon the variable type.

This is the new value of the machine variable.

prev_value

Type: Varies depending upon the variable type.

This is the previous value of the machine variable.

change

Type: Varies depending upon the variable type.

If the machine variable just changed, this will be the amount of the change. If it's not possible to determine a numeric change (for example, if this machine variable is a string), then this *change* value will be set to the boolean *True*.

Response

None

mode_start (BCP command)

A game mode has just started. The mode is passed via the name parameter, and the mode's priority is passed as an integer via the priority.

Origin

Pin controller

Parameters

name

Type: `string`

The mode name.

priority

Type: `int`

The mode priority.

Response

None

mode_stop (BCP command)

Indicates the mode has stopped.

Origin

Pin controller

Parameters

name

Type: `string`

The mode name.

Response

None

monitor_start (BCP command)

New in version 0.33.

Request from the media controller to the pin controller to begin monitoring events in the specified category. Events will not be automatically sent to the media controller from the pin controller via BCP unless they are requested using the `monitor_start` or *register_trigger* commands.

Origin

Media controller

Parameters

category

Single string value, type: one of the following options: events, devices, machine_vars, player_vars, switches, modes, ball, or timer.

The value of `category` determines the category of events to begin monitoring. Options for `category` are:

- `events` - All events in the pin controller
- `devices` - All device state changes
- `machine_vars` - All machine variable changes
- `player_vars` - All player variable changes
- `switches` - All switch state changes
- `modes` - All mode events (start, stop)
- `core_events` - Core MPF events (ball handing, player turn, etc.)

Response

None

monitor_stop (BCP command)

New in version 0.33.

Request from the media controller to the pin controller to stop monitoring events in the specified category. Once a monitor has been started, events will continue to be automatically sent to the media controller from the pin controller via BCP until they are stopped using the `monitor_stop` or *remove_trigger* commands.

Origin

Media controller

Parameters

category

Single string value, type: one of the following options: events, devices, machine_vars, player_vars, switches, modes, ball, or timer.

The value of `category` determines the category of events to stop monitoring. Options for `category` are:

- `events` - All events in the pin controller
- `devices` - All device state changes
- `machine_vars` - All machine variable changes
- `player_vars` - All player variable changes
- `switches` - All switch state changes
- `modes` - All mode events (start, stop)
- `core_events` - Core MPF events (ball handing, player turn, etc.)

Response

None

player_added (BCP command)

A player has just been added, with the player number passed via the *player_num* parameter. Typically these commands only occur during Ball 1.

Origin

Pin controller

Parameters

player_num

Type: `int`

The player number just added.

Response

None

player_turn_start (BCP command)

A new player's turn has begun. If a player has an extra ball, this command will *not* be sent between balls. However, a new *ball_start* command will be sent when the same player's additional balls start.

Origin

Pin controller

Parameters

player_num

Type: `int`

The player number.

Response

None

player_variable (BCP command)

This is a generic "catch all" which sends player-specific variables to the media controller any time they change. Since the pin controller will most likely track hundreds of variables per player (with many being internal things that the media controller doesn't care about), it's recommended that the pin controller has a way to filter which player variables are sent to the media controller. Also note the parameter *player_num* indicates which player this variable is for (starting with 1 for the first player). While it's usually the case that the *player_variable* command will be sent for the player whose turn it is, that's not always the case. (For example, when a second player is added during the first player's ball, the second player's default variables will be initialized at 0 and a *player_variable* event for player 2 will be sent even though player 1 is up.

Origin

Pin controller

Parameters

name

Type: `string`

This is the name of the player variable.

player_num

Type: `int`

This is the player number the variable is for (starting with 1 for the first player).

value

Type: Varies depending upon the variable type.

This is the new value of the player variable.

prev_value

Type: Varies depending upon the variable type.

This is the previous value of the player variable.

change

Type: Varies depending upon the variable type.

If the player variable just changed, this will be the amount of the change. If it's not possible to determine a numeric change (for example, if this player variable is a string), then this *change* value will be set to the boolean *True*.

Response

None

register_trigger (BCP command)

Request from the media controller to the pin controller to register an event name as a trigger so it will be sent via BCP to the media controller whenever the event is posted in MPF.

Origin

Media controller

Parameters

event

Type: `string`

This is the name of the trigger event to register with the pin controller.

Response

None

remove_trigger (BCP command)

New in version 0.33.

Request from the media controller to the pin controller to cancel/deregister an event name as a trigger so it will no longer be sent via BCP to the media controller whenever the event is posted in MPF.

Origin

Media controller

Parameters

event

Type: `string`

This is the name of the trigger event to cancel/deregister with the pin controller.

Response

None

reset (BCP command)

This command notifies the media controller that the pin controller is in the process of performing a reset. If necessary, the media controller should perform its own reset process. The media controller *must* respond with a *reset_complete* command when finished.

Origin

Pin controller

Parameters

None

Response

reset_complete when reset process has finished

reset_complete (BCP command)

This command notifies the pin controller that reset process is now complete. It *must* be sent in response to receiving a *reset* command.

Origin

Media controller

Parameters

None

Response

None

switch (BCP command)

Indicates that the other side should process the changed state of a switch. When sent from the media controller to the pin controller, this is typically used to implement a virtual keyboard interface via the media controller (where the player can activate pinball machine switches via keyboard keys for testing). For example, for the media controller to tell the pin controller that the player just pushed the start button, the command would be:

```
switch?name=start&state=1
```

followed very quickly by

```
switch?name=start&state=0
```

When sent from the pin controller to the media controller, this is used to send switch inputs to things like video modes, high score name entry, and service menu navigation. Note that the pin controller should not send the state of every switch change at all times, as the media controller doesn't need it and that would add lots of unnecessary commands. Instead the pin controller should only send switches based on some mode of operation that needs them. (For example, when the video mode starts, the pin controller would start sending the switch states of the flipper buttons, and when the video mode ends, it would stop.)

Origin

Pin controller or media controller

Parameters

name

Type: `string`

This is the name of the switch.

state

Type: `int`

The new switch state: *1* for active, and *0* for inactive.

Response

None

trigger (BCP command)

This command allows the one side to trigger the other side to do something. For example, the pin controller might send trigger commands to tell the media controller to start shows, play sound effects, or update the display. The media controller might send a trigger to the pin controller to flash the strobes at the down beat of a music track or to pulse the knocker in concert with a replay show.

Origin

Pin controller or media controller

Parameters

name

Type: `string`

This is the name of the trigger.

Note: Trigger messages may contain any additional parameters as needed by the application.

Response

Varies

8.8 Method & Class Index

Index

A

a_side_busy (*mpf.platforms.snux.SnuxHardwarePlatform attribute*), 140
Accelerometer (*class in mpf.devices.accelerometer*), 53
accept_connection() (*mpf.tests.MpfBcpTestCase.MockBcpClient method*), 154
Accrual (*class in mpf.devices.logic_blocks*), 53
Achievement (*class in mpf.devices.achievement*), 55
AchievementGroup (*class in mpf.devices.achievement_group*), 55
activate() (*mpf.devices.diverter.Diverter method*), 67
active (*mpf.core.mode.Mode attribute*), 208
active (*mpf.modes.attract.code.attract.Attract attribute*), 102
active (*mpf.modes.bonus.code.bonus.Bonus attribute*), 105
active (*mpf.modes.carousel.code.carousel.Carousel attribute*), 107
active (*mpf.modes.credits.code.credits.Credits attribute*), 109
active (*mpf.modes.game.code.game.Game attribute*), 112
active (*mpf.modes.high_score.code.high_score.HighScore attribute*), 115
active (*mpf.modes.match.code.match.Match attribute*), 117
active (*mpf.modes.service.code.service.Service attribute*), 119
active (*mpf.modes.tilt.code.tilt.Tilt attribute*), 122
active_sequences (*mpf.devices.shot.Shot attribute*), 95
add() (*mpf.core.delays.DelayManager method*), 221
add() (*mpf.devices.timer.Timer method*), 100
add_a_ball() (*mpf.devices.multiball.Multiball method*), 82
add_async_handler() (*mpf.core.events.EventManager method*), 33
add_ball() (*mpf.devices.playfield.Playfield method*), 84
add_ball_to_device() (*mpf.platforms.smart_virtual.SmartVirtualHardwarePlatform method*), 139
add_captured_ball() (*mpf.core.ball_controller.BallController method*), 31
add_color() (*mpf.core.rgb_color.RGBColor static method*), 214
add_color() (*mpf.core.rgba_color.RGBAColor static method*), 212
add_credit() (*mpf.modes.credits.code.credits.Credits method*), 109
add_delay_manager() (*mpf.core.delays.DelayManagerRegistry method*), 224
add_handler() (*mpf.core.events.EventManager method*), 33
add_handler() (*mpf.devices.switch.Switch method*), 98
add_if_doesnt_exist() (*mpf.core.delays.DelayManager method*), 222
add_incoming_ball() (*mpf.devices.ball_device.ball_device.BallDevice method*), 58
add_incoming_ball() (*mpf.devices.playfield.Playfield method*), 85
add_missing_balls() (*mpf.devices.playfield.Playfield method*), 85
add_mode_event_handler() (*mpf.core.mode.Mode method*), 208
add_mode_event_handler() (*mpf.modes.attract.code.attract.Attract method*), 102

add_mode_event_handler() (*mpf.modes.bonus.code.bonus.Bonus method*), 105
add_mode_event_handler() (*mpf.modes.carousel.code.carousel.Carousel method*), 107
add_mode_event_handler() (*mpf.modes.credits.code.credits.Credits method*), 109
add_mode_event_handler() (*mpf.modes.game.code.game.Game method*), 112
add_mode_event_handler() (*mpf.modes.high_score.code.high_score.HighScore method*), 115
add_mode_event_handler() (*mpf.modes.match.code.match.Match method*), 117
add_mode_event_handler() (*mpf.modes.service.code.service.Service method*), 119
add_mode_event_handler() (*mpf.modes.tilt.code.tilt.Tilt method*), 122
add_monitor() (*mpf.core.switch_controller.SwitchController method*), 48
add_platform() (*mpf.core.machine.MachineController method*), 38
add_platform() (*mpf.tests.MpfTestCase.TestMachineController method*), 202
add_player() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 164
add_player() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 174
add_setting() (*mpf.core.settings_controller.SettingsController method*), 46
add_switch_handler() (*mpf.core.switch_controller.SwitchController method*), 48
add_switch_handler_obj() (*mpf.core.switch_controller.SwitchController method*), 48
add_text() (*mpf.devices.segment_display.SegmentDisplay method*), 90
add_to_bank() (*mpf.devices.drop_target.DropTarget method*), 70
add_to_config_validator() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase static method*), 155
add_to_config_validator() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase static method*), 164
add_to_config_validator() (*mpf.tests.MpfGameTestCase.MpfGameTestCase static method*), 174
add_to_config_validator() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase static method*), 184
add_to_config_validator() (*mpf.tests.MpfTestCase.MpfTestCase static method*), 192
add_to_group() (*mpf.devices.achievement.Achievement method*), 56
advance() (*mpf.devices.shot.Shot method*), 95
advance_time_and_run() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 155
advance_time_and_run() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 164
advance_time_and_run() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 174
advance_time_and_run() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 184
advance_time_and_run() (*mpf.tests.MpfTestCase.MpfTestCase method*), 192
any() (*mpf.core.utility_functions.Util static method*), 216
are_balls_collected() (*mpf.core.ball_controller.BallController method*), 31
assertAlmostEqual() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 155
assertAlmostEqual() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 164
assertAlmostEqual() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 174
assertAlmostEqual() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 184
assertAlmostEqual() (*mpf.tests.MpfTestCase.MpfTestCase method*), 193
assertBallNumber() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 164
assertBallNumber() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 174
assertBallsInPlay() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 164
assertBallsInPlay()

(*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 175
assertColorAlmostEqual() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 156
assertColorAlmostEqual() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 165
assertColorAlmostEqual() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 175
assertColorAlmostEqual() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 184
assertColorAlmostEqual() (*mpf.tests.MpfTestCase.MpfTestCase method*), 193
assertCountEqual() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 156
assertCountEqual() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 165
assertCountEqual() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 175
assertCountEqual() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 185
assertCountEqual() (*mpf.tests.MpfTestCase.MpfTestCase method*), 193
assertDictContainsSubset() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 156
assertDictContainsSubset() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 165
assertDictContainsSubset() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 175
assertDictContainsSubset() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 185
assertDictContainsSubset() (*mpf.tests.MpfTestCase.MpfTestCase method*), 193
assertEqual() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 156
assertEqual() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 165
assertEqual() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 175
assertEqual() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 185
assertEqual() (*mpf.tests.MpfTestCase.MpfTestCase method*), 193
assertEventCalled() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 156
assertEventCalled() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 165
assertEventCalled() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 175
assertEventCalled() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 185
assertEventCalled() (*mpf.tests.MpfTestCase.MpfTestCase method*), 193
assertEventCalledWith() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 156
assertEventCalledWith() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 166
assertEventCalledWith() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 176
assertEventCalledWith() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 185
assertEventCalledWith() (*mpf.tests.MpfTestCase.MpfTestCase method*), 194
assertEventNotCalled() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 157
assertEventNotCalled() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 166
assertEventNotCalled() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 176
assertEventNotCalled() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 186
assertEventNotCalled() (*mpf.tests.MpfTestCase.MpfTestCase method*), 194
assertFalse() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 157
assertFalse() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 166
assertFalse() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 176

assertFalse() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 186
assertFalse() (*mpf.tests.MpfTestCase.MpfTestCase method*), 194
assertGameIsNotRunning() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 166
assertGameIsNotRunning() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 176
assertGameIsRunning() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 166
assertGameIsRunning() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 176
assertGreater() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 157
assertGreater() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 166
assertGreater() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 176
assertGreater() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 186
assertGreater() (*mpf.tests.MpfTestCase.MpfTestCase method*), 194
assertGreaterEqual() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 157
assertGreaterEqual() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 166
assertGreaterEqual() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 177
assertGreaterEqual() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 186
assertGreaterEqual() (*mpf.tests.MpfTestCase.MpfTestCase method*), 194
assertIn() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 157
assertIn() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 166
assertIn() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 177
assertIn() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 186
assertIn() (*mpf.tests.MpfTestCase.MpfTestCase method*), 194
assertIs() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 157
assertIs() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 166
assertIs() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 177
assertIs() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 186
assertIs() (*mpf.tests.MpfTestCase.MpfTestCase method*), 194
assertIsInstance() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 157
assertIsInstance() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 166
assertIsInstance() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 177
assertIsInstance() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 186
assertIsInstance() (*mpf.tests.MpfTestCase.MpfTestCase method*), 194
assertIsNone() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 157
assertIsNone() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 167
assertIsNone() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 177
assertIsNone() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 186
assertIsNone() (*mpf.tests.MpfTestCase.MpfTestCase method*), 194
assertIsNot() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 157
assertIsNot() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 167
assertIsNot() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 177
assertIsNot() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 186
assertIsNot() (*mpf.tests.MpfTestCase.MpfTestCase method*), 194
assertIsNotNone() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 157
assertIsNotNone() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 167
assertIsNotNone() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 177
assertIsNotNone() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 186

assertIsNotNone() (*mpf.tests.MpfTestCase.MpfTestCase method*), 194
assertLess() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 157
assertLess() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 167
assertLess() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 177
assertLess() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 186
assertLess() (*mpf.tests.MpfTestCase.MpfTestCase method*), 195
assertLessEqual() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 157
assertLessEqual() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 167
assertLessEqual() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 177
assertLessEqual() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 186
assertLessEqual() (*mpf.tests.MpfTestCase.MpfTestCase method*), 195
assertListEqual() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 157
assertListEqual() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 167
assertListEqual() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 177
assertListEqual() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 186
assertListEqual() (*mpf.tests.MpfTestCase.MpfTestCase method*), 195
assertLogs() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 157
assertLogs() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 167
assertLogs() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 177
assertLogs() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 186
assertLogs() (*mpf.tests.MpfTestCase.MpfTestCase method*), 195
assertMultiLineEqual() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 158
assertMultiLineEqual() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 167
assertMultiLineEqual() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 177
assertMultiLineEqual() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 187
assertMultiLineEqual() (*mpf.tests.MpfTestCase.MpfTestCase method*), 195
assertNotAlmostEqual() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 158
assertNotAlmostEqual() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 167
assertNotAlmostEqual() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 177
assertNotAlmostEqual() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 187
assertNotAlmostEqual() (*mpf.tests.MpfTestCase.MpfTestCase method*), 195
assertNotEqual() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 158
assertNotEqual() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTe method*), 167
assertNotEqual() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 178
assertNotEqual() (*mpf.tests.MpfMachineTestCase.MpfMachineTestC method*), 187
assertNotEqual() (*mpf.tests.MpfTestCase.MpfTestCase method*), 195
assertNotIn() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 158
assertNotIn() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCa method*), 167
assertNotIn() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 178
assertNotIn() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 187
assertNotIn() (*mpf.tests.MpfTestCase.MpfTestCase method*), 195
assertNotIsInstance() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 158
assertNotIsInstance() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase*

method), 167
assertNotIsInstance() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 178
assertNotIsInstance() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 187
assertNotIsInstance() (*mpf.tests.MpfTestCase.MpfTestCase method*), 195
assertNotRegex() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 158
assertNotRegex() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 168
assertNotRegex() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 178
assertNotRegex() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 187
assertNotRegex() (*mpf.tests.MpfTestCase.MpfTestCase method*), 195
assertNumBallsKnown() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 158
assertNumBallsKnown() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 168
assertNumBallsKnown() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 178
assertNumBallsKnown() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 187
assertNumBallsKnown() (*mpf.tests.MpfTestCase.MpfTestCase method*), 195
assertPlayerCount() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 168
assertPlayerCount() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 178
assertPlayerNumber() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 168
assertPlayerNumber() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 178
assertRaises() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 158
assertRaises() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 168
assertRaises() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 178
assertRaises() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 187
assertRaises() (*mpf.tests.MpfTestCase.MpfTestCase method*), 195
assertRaisesRegex() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 159
assertRaisesRegex() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 168
assertRaisesRegex() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 178
assertRaisesRegex() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 187
assertRaisesRegex() (*mpf.tests.MpfTestCase.MpfTestCase method*), 196
assertRegex() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 159
assertRegex() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 168
assertRegex() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 179
assertRegex() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 188
assertRegex() (*mpf.tests.MpfTestCase.MpfTestCase method*), 196
assertSequenceEqual() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 159
assertSequenceEqual() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 169
assertSequenceEqual() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 179
assertSequenceEqual() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 188
assertSequenceEqual() (*mpf.tests.MpfTestCase.MpfTestCase method*), 196
assertSetEqual() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 159
assertSetEqual() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 169
assertSetEqual() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 179
assertSetEqual() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 188
assertSetEqual() (*mpf.tests.MpfTestCase.MpfTestCase method*), 196
assertWarns() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase*

method), 159
assertTrue() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 169
assertTrue() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 179
assertTrue() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 188
assertTrue() (*mpf.tests.MpfTestCase.MpfTestCase method*), 197
assertTupleEqual() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 159
assertTupleEqual() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 169
assertTupleEqual() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 179
assertTupleEqual() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 188
assertTupleEqual() (*mpf.tests.MpfTestCase.MpfTestCase method*), 197
assertWarns() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 160
assertWarns() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 169
assertWarns() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 179
assertWarns() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 188
assertWarns() (*mpf.tests.MpfTestCase.MpfTestCase method*), 197
assertWarnsRegex() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 160
assertWarnsRegex() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 170
assertWarnsRegex() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 180
assertWarnsRegex() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 189
assertWarnsRegex() (*mpf.tests.MpfTestCase.MpfTestCase method*), 197
AsyncioSyncAssetManager (*class in mpf.core.assets*), 29
Attract (*class in mpf.modes.attract.code.attract*), 102
audit() (*mpf.plugins.auditor.Auditor method*), 30
audit_event() (*mpf.plugins.auditor.Auditor method*), 30
auditor() (*mpf.plugins.auditor.Auditor method*), 30
auditor_shot() (*mpf.plugins.auditor.Auditor method*), 30
auditor_switch() (*mpf.plugins.auditor.Auditor method*), 30
Auditor (*class in mpf.plugins.auditor*), 29
auto_stop_on_ball_end (*mpf.core.mode.Mode attribute*), 209
AutofireCoil (*class in mpf.devices.autofire*), 57
available_balls (*mpf.devices.ball_device.ball_device.BallDevice attribute*), 58
award() (*mpf.devices.extra_ball.ExtraBall method*), 72
award() (*mpf.devices.extra_ball_group.ExtraBallGroup method*), 71
award_disabled() (*mpf.devices.extra_ball_group.ExtraBallGroup method*), 71
award_lit() (*mpf.devices.extra_ball_group.ExtraBallGroup method*), 72

B

ball_arrived() (*mpf.devices.playfield.Playfield method*), 85
ball_drained() (*mpf.modes.game.code.game.Game method*), 112
ball_ending() (*mpf.modes.game.code.game.Game method*), 112
ball_search (*mpf.devices.playfield.Playfield attribute*), 85
ball_search_block() (*mpf.devices.playfield.Playfield method*), 85
ball_search_disable() (*mpf.devices.playfield.Playfield method*), 85
ball_search_enable() (*mpf.devices.playfield.Playfield method*), 85
ball_search_unblock() (*mpf.devices.playfield.Playfield method*), 85
BallController (*class in mpf.core.ball_controller*), 31
BallDevice (*class in mpf.devices.ball_device.ball_device*), 58
BallHold (*class in mpf.devices.ball_hold*), 60
BallLock (*class in mpf.devices.ball_lock*), 61
balls (*mpf.devices.ball_device.ball_device.BallDevice attribute*), 58
balls (*mpf.devices.playfield.Playfield attribute*), 85
balls_in_play (*mpf.modes.game.code.game.Game attribute*), 112
BallSave (*class in mpf.devices.ball_save*), 62

BallSearch (*class in mpf.core.ball_search*), 204
Bcp (*class in mpf.core.bcp.bcp*), 31
bin_str_to_hex_str() (*mpf.core.utility_functions.Util static method*), 216
blend() (*mpf.core.rgb_color.RGBColor static method*), 214
blend() (*mpf.core.rgba_color.RGBAColor static method*), 213
block() (*mpf.core.ball_search.BallSearch method*), 204
blocked (*mpf.core.ball_search.BallSearch attribute*), 204
BlockEventPlayer (*class in mpf.config_players.block_event_player*), 147
blue (*mpf.core.rgb_color.RGBColor attribute*), 215
blue (*mpf.core.rgba_color.RGBAColor attribute*), 213
Bonus (*class in mpf.modes.bonus.code.bonus*), 104

C

c_side_active (*mpf.platforms.snux.SnuxHardwarePlatform attribute*), 140
cancel() (*mpf.devices.sequence_shot.SequenceShot method*), 90
cancel_ball_search() (*mpf.core.ball_search.BallSearch method*), 205
cancel_futures() (*mpf.core.utility_functions.Util static method*), 217
cancel_path_if_target_is() (*mpf.devices.ball_device.ball_device.BallDevice method*), 58
capacity (*mpf.devices.ball_device.ball_device.BallDevice attribute*), 58
Carousel (*class in mpf.modes.carousel.code.carousel*), 107
change_tick_interval() (*mpf.devices.timer.Timer method*), 100
check() (*mpf.core.delays.DelayManager method*), 222
check_hw_switches() (*mpf.devices.score_reel.ScoreReel method*), 89
chime() (*mpf.devices.score_reel_group.ScoreReelGroup class method*), 87
chunker() (*mpf.core.utility_functions.Util static method*), 217
clear() (*mpf.core.delays.DelayManager method*), 222
clear_all_credits() (*mpf.modes.credits.code.credits.Credits method*), 110
clear_boot_hold() (*mpf.core.machine.MachineController method*), 39
clear_boot_hold() (*mpf.tests.MpfTestCase.TestMachineController method*), 202
clear_context() (*mpf.config_players.coil_player.CoilPlayer method*), 148
clear_context() (*mpf.config_players.light_player.LightPlayer method*), 150
clear_context() (*mpf.config_players.queue_relay_player.QueueRelay method*), 151
clear_context() (*mpf.config_players.segment_display_player.Segmen method*), 152
clear_context() (*mpf.config_players.show_player.ShowPlayer method*), 152
clear_context() (*mpf.config_players.variable_player.VariablePlayer method*), 153
clear_hw_rule() (*mpf.core.platform_controller.PlatformController method*), 43
clear_hw_rule() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 125
clear_hw_rule() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 128
clear_hw_rule() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 132
clear_hw_rule() (*mpf.platforms.rpi.rpi.RaspberryPiHardwarePlatfor method*), 138
clear_hw_rule() (*mpf.platforms.snux.SnuxHardwarePlatform method*), 140
clear_hw_rule() (*mpf.platforms.spike.spike.SpikePlatform method*), 142
clear_hw_rule() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 144
clear_hw_rule() (*mpf.platforms.virtual_pinball.virtual_pinball.Virtua method*), 145
clear_stack() (*mpf.devices.light.Light method*), 78
CoilPlayer (*class in mpf.config_players.coil_player*), 147
collect_balls() (*mpf.core.ball_controller.BallController method*), 31
color() (*mpf.devices.light.Light method*), 78
color() (*mpf.devices.light_group.LightRing method*), 77
color() (*mpf.devices.light_group.LightStrip method*), 77
color_correct() (*mpf.devices.light.Light method*), 78
ComboSwitch (*class in mpf.devices.combo_switch*), 64
complete() (*mpf.devices.achievement.Achievement method*), 56
complete() (*mpf.devices.logic_blocks.Accrual method*), 54
complete() (*mpf.devices.logic_blocks.Counter method*), 65
complete() (*mpf.devices.logic_blocks.Sequence method*), 91

completed (*mpf.devices.logic_blocks.Accrual attribute*), 54
completed (*mpf.devices.logic_blocks.Counter attribute*), 65
completed (*mpf.devices.logic_blocks.Sequence attribute*), 91
configure_accelerometer() (*mpf.platforms.mma8451.MMA8451Platform method*), 130
configure_accelerometer() (*mpf.platforms.p3_roc.P3RocHardwarePlatform method*), 135
configure_accelerometer() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 144
configure_dmd() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 125
configure_dmd() (*mpf.platforms.p_roc.PRocHardwarePlatform method*), 136
configure_dmd() (*mpf.platforms.spike.spike.SpikePlatform method*), 142
configure_dmd() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 144
configure_driver() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 125
configure_driver() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 128
configure_driver() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 132
configure_driver() (*mpf.platforms.p3_roc.P3RocHardwarePlatform method*), 135
configure_driver() (*mpf.platforms.p_roc.PRocHardwarePlatform method*), 136
configure_driver() (*mpf.platforms.rpi.rpi.RaspberryPiHardwarePlatform method*), 138
configure_driver() (*mpf.platforms.smart_virtual.SmartVirtualHardwarePlatform method*), 139
configure_driver() (*mpf.platforms.snux.SnuxHardwarePlatform method*), 140
configure_driver() (*mpf.platforms.spike.spike.SpikePlatform method*), 142
configure_driver() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 144
configure_driver() (*mpf.platforms.virtual_pinball.virtual_pinball.VirtualPinballPlatf method*), 146
configure_hardware_sound_system() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 128
configure_hardware_sound_system() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 144
configure_i2c() (*mpf.platforms.p3_roc.P3RocHardwarePlatform method*), 135
configure_i2c() (*mpf.platforms.rpi.rpi.RaspberryPiHardwarePlatfor method*), 138
configure_i2c() (*mpf.platforms.smbus2.Smbus2 method*), 140
configure_i2c() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 144
configure_light() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 125
configure_light() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 128
configure_light() (*mpf.platforms.openpixel.OpenpixelHardwarePlatform method*), 131
configure_light() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 132
configure_light() (*mpf.platforms.spike.spike.SpikePlatform method*), 142
configure_light() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 144
configure_light() (*mpf.platforms.virtual_pinball.virtual_pinball.VirtualPinballPlatf method*), 146
configure_logging() (*mpf.core.ball_search.BallSearch method*), 205
configure_logging() (*mpf.core.data_manager.DataManager method*), 220
configure_logging() (*mpf.core.delays.DelayManager method*), 222
configure_logging() (*mpf.core.logging.LogMixin method*), 207
configure_logging() (*mpf.core.mode.Mode method*), 209
configure_logging() (*mpf.modes.attract.code.attract.Attract method*), 102
configure_logging()

(*mpf.modes.bonus.code.bonus.Bonus method*), 105
configure_logging() (*mpf.modes.carousel.code.carousel.Carousel method*), 107
configure_logging() (*mpf.modes.credits.code.credits.Credits method*), 110
configure_logging() (*mpf.modes.game.code.game.Game method*), 113
configure_logging() (*mpf.modes.high_score.code.high_score.HighScore method*), 115
configure_logging() (*mpf.modes.match.code.match.Match method*), 118
configure_logging() (*mpf.modes.service.code.service.Service method*), 120
configure_logging() (*mpf.modes.tilt.code.tilt.Tilt method*), 122
configure_machine_var() (*mpf.core.machine.MachineController method*), 39
configure_machine_var() (*mpf.tests.MpfTestCase.TestMachineController method*), 202
configure_mode_settings() (*mpf.core.mode.Mode method*), 209
configure_rgb_dmd() (*mpf.platforms.smartmatrix.SmartMatrixHardwarePlatform method*), 139
configure_rgb_dmd() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 144
configure_segment_display() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 128
configure_segment_display() (*mpf.platforms.mypinballs.mypinballs.MyPinballsHardwarePlatform method*), 130
configure_segment_display() (*mpf.platforms.p_roc.PRocHardwarePlatform method*), 136
configure_segment_display() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 144
configure_servo() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 125
configure_servo() (*mpf.platforms.i2c_servo_controller.I2CServoControllerHardwarePlatform method*), 128
configure_servo() (*mpf.platforms.pololu_maestro.PololuMaestroHardwarePlatform method*), 137
configure_servo() (*mpf.platforms.rpi.rpi.RaspberryPiHardwarePlatform method*), 138
configure_servo() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 144
configure_stepper() (*mpf.platforms.trinamics_steprocker.TrinamicsStepRocker method*), 143
configure_stepper() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 144
configure_switch() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 125
configure_switch() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 129
configure_switch() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 132
configure_switch() (*mpf.platforms.p3_roc.P3RocHardwarePlatform method*), 135
configure_switch() (*mpf.platforms.p_roc.PRocHardwarePlatform method*), 136
configure_switch() (*mpf.platforms.rpi.rpi.RaspberryPiHardwarePlatform method*), 138
configure_switch() (*mpf.platforms.spike.spike.SpikePlatform method*), 142
configure_switch() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 144
configure_switch() (*mpf.platforms.virtual_pinball.virtual_pinball.VirtualPinballPlatf method*), 146
connect() (*mpf.platforms.p3_roc.P3RocHardwarePlatform method*), 135
connect() (*mpf.tests.MpfBcpTestCase.MockBcpClient method*), 154
convert_number_from_config() (*mpf.platforms.fast.fast.FastHardwarePlatform static method*), 126
convert_to_simply_type() (*mpf.core.utility_functions.Util static method*), 217
convert_to_str() (*mpf.core.utility_functions.Util static method*), 217

count() (*mpf.devices.logic_blocks.Counter method*), 65
Counter (*class in mpf.devices.logic_blocks*), 65
create_data_manager() (*mpf.core.machine.MachineController method*), 39
create_data_manager() (*mpf.tests.MpfTestCase.TestMachineController method*), 202
create_devices() (*mpf.core.device_manager.DeviceManager method*), 32
create_machinewide_device_control_events() (*mpf.core.device_manager.DeviceManager method*), 32
create_mode_devices() (*mpf.core.mode.Mode method*), 209
create_mode_devices() (*mpf.core.mode_controller.ModeController method*), 41
create_mode_devices() (*mpf.modes.attract.code.attract.Attract method*), 103
create_mode_devices() (*mpf.modes.bonus.code.bonus.Bonus method*), 105
create_mode_devices() (*mpf.modes.carousel.code.carousel.Carousel method*), 108
create_mode_devices() (*mpf.modes.credits.code.credits.Credits method*), 110
create_mode_devices() (*mpf.modes.game.code.game.Game method*), 113
create_mode_devices() (*mpf.modes.high_score.code.high_score.HighScore method*), 116
create_mode_devices() (*mpf.modes.match.code.match.Match method*), 118
create_mode_devices() (*mpf.modes.service.code.service.Service method*), 120
create_mode_devices() (*mpf.modes.tilt.code.tilt.Tilt method*), 122
create_show_config() (*mpf.core.show_controller.ShowController method*), 47
Credits (*class in mpf.modes.credits.code.credits*), 109

D

DataManager (*class in mpf.core.data_manager*), 220
db_to_gain() (*mpf.core.utility_functions.Util static method*), 217
deactivate() (*mpf.devices.diverter.Diverter method*), 67
debug_log() (*mpf.core.ball_search.BallSearch method*), 205
debug_log() (*mpf.core.data_manager.DataManager method*), 220
debug_log() (*mpf.core.delays.DelayManager method*), 222
debug_log() (*mpf.core.logging.LogMixin method*), 207
debug_log() (*mpf.core.mode.Mode method*), 209
debug_log() (*mpf.modes.attract.code.attract.Attract method*), 103
debug_log() (*mpf.modes.bonus.code.bonus.Bonus method*), 105
debug_log() (*mpf.modes.carousel.code.carousel.Carousel method*), 108
debug_log() (*mpf.modes.credits.code.credits.Credits method*), 110
debug_log() (*mpf.modes.game.code.game.Game method*), 113
debug_log() (*mpf.modes.high_score.code.high_score.HighScore method*), 116
debug_log() (*mpf.modes.match.code.match.Match method*), 118
debug_log() (*mpf.modes.service.code.service.Service method*), 120
debug_log() (*mpf.modes.tilt.code.tilt.Tilt method*), 122
decrease_volume() (*mpf.devices.hardware_sound_system.HardwareSoundSystem method*), 75
delay (*mpf.core.mode.Mode attribute*), 209
delay (*mpf.devices.playfield.Playfield attribute*), 85
delayed_eject() (*mpf.devices.ball_save.BallSave method*), 62
DelayManager (*class in mpf.core.delays*), 221
DelayManagerRegistry (*class in mpf.core.delays*), 223
DeviceManager (*class in mpf.core.device_manager*), 32
dict_merge() (*mpf.core.utility_functions.Util static method*), 217
DigitalOutput (*class in mpf.devices.digital_output*), 66
disable() (*mpf.core.ball_search.BallSearch method*), 205
disable() (*mpf.devices.achievement.Achievement method*), 56
disable() (*mpf.devices.achievement_group.AchievementGroup method*), 55
disable() (*mpf.devices.autofire.AutofireCoil method*), 57
disable() (*mpf.devices.ball_hold.BallHold method*),

60
disable() (mpf.devices.ball_lock.BallLock method), 61
disable() (mpf.devices.ball_save.BallSave method), 62
disable() (mpf.devices.digital_output.DigitalOutput method), 66
disable() (mpf.devices.diverter.Diverter method), 67
disable() (mpf.devices.driver.Driver method), 63
disable() (mpf.devices.dual_wound_coil.DualWoundCoil method), 71
disable() (mpf.devices.flipper.Flipper method), 74
disable() (mpf.devices.kickback.Kickback method), 76
disable() (mpf.devices.logic_blocks.Accrual method), 54
disable() (mpf.devices.logic_blocks.Counter method), 65
disable() (mpf.devices.logic_blocks.Sequence method), 91
disable() (mpf.devices.magnet.Magnet method), 80
disable() (mpf.devices.multiball.Multiball method), 83
disable() (mpf.devices.multiball_lock.MultiballLock method), 82
disable() (mpf.devices.shot.Shot method), 95
disable() (mpf.devices.shot_group.ShotGroup method), 93
disable() (mpf.plugins.auditor.Auditor method), 30
disable_keep_up() (mpf.devices.drop_target.DropTarget method), 70
disable_rotation() (mpf.devices.shot_group.ShotGroup method), 93
Diverter (class in mpf.devices.diverter), 67
Dmd (class in mpf.devices.dmd), 68
does_event_exist() (mpf.core.events.EventManager method), 34
drain_ball() (mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method), 170
drain_ball() (mpf.tests.MpfGameTestCase.MpfGameTestCase method), 180
Driver (class in mpf.devices.driver), 63
driver_action() (mpf.platforms.snux.SnuxHardwarePlatform method), 140
DropTarget (class in mpf.devices.drop_target), 69
DropTargetBank (class in mpf.devices.drop_target), 68
DualWoundCoil (class in mpf.devices.dual_wound_coil), 70
dump() (mpf.core.mode_controller.ModeController method), 41
dump_ball_counts() (mpf.core.ball_controller.BallController method), 31

E

early_ball_save() (mpf.devices.ball_save.BallSave method), 62
eject() (mpf.devices.ball_device.ball_device.BallDevice method), 58
eject_all() (mpf.devices.ball_device.ball_device.BallDevice method), 58
enable() (mpf.core.ball_search.BallSearch method), 205
enable() (mpf.devices.achievement.Achievement method), 56
enable() (mpf.devices.achievement_group.AchievementGroup method), 55
enable() (mpf.devices.autofire.AutofireCoil method), 57
enable() (mpf.devices.ball_hold.BallHold method), 60
enable() (mpf.devices.ball_lock.BallLock method), 61
enable() (mpf.devices.ball_save.BallSave method), 63
enable() (mpf.devices.combo_switch.ComboSwitch method), 64
enable() (mpf.devices.digital_output.DigitalOutput method), 66
enable() (mpf.devices.diverter.Diverter method), 67
enable() (mpf.devices.driver.Driver method), 63
enable() (mpf.devices.drop_target.DropTargetBank method), 69
enable() (mpf.devices.dual_wound_coil.DualWoundCoil method), 71
enable() (mpf.devices.extra_ball.ExtraBall method), 72
enable() (mpf.devices.flipper.Flipper method), 74
enable() (mpf.devices.kickback.Kickback method), 76
enable() (mpf.devices.logic_blocks.Accrual method), 54
enable() (mpf.devices.logic_blocks.Counter method), 65
enable() (mpf.devices.logic_blocks.Sequence method), 91
enable() (mpf.devices.magnet.Magnet method), 80
enable() (mpf.devices.multiball.Multiball method), 83
enable() (mpf.devices.multiball_lock.MultiballLock method), 82
enable() (mpf.devices.sequence_shot.SequenceShot method), 90
enable() (mpf.devices.shot.Shot method), 95
enable() (mpf.devices.shot_group.ShotGroup method), 93
enable() (mpf.devices.shot_profile.ShotProfile method), 94

enable() (*mpf.devices.state_machine.StateMachine method*), 96
enable() (*mpf.devices.timed_switch.TimedSwitch method*), 99
enable() (*mpf.devices.timer.Timer method*), 100
enable() (*mpf.plugins.auditor.Auditor method*), 30
enable_credit_play() (*mpf.modes.credits.code.credits.Credits method*), 110
enable_events() (*mpf.core.player.Player method*), 212
enable_free_play() (*mpf.modes.credits.code.credits.Credits method*), 110
enable_keep_up() (*mpf.devices.drop_target.DropTarget method*), 70
enable_rotation() (*mpf.devices.shot_group.ShotGroup method*), 93
enabled (*mpf.core.ball_search.BallSearch attribute*), 205
enabled (*mpf.devices.achievement_group.AchievementGroup attribute*), 55
enabled (*mpf.devices.extra_ball.ExtraBall attribute*), 72
enabled (*mpf.devices.extra_ball_group.ExtraBallGroup attribute*), 72
enabled (*mpf.devices.logic_blocks.Accrual attribute*), 54
enabled (*mpf.devices.logic_blocks.Counter attribute*), 65
enabled (*mpf.devices.logic_blocks.Sequence attribute*), 91
enabled (*mpf.devices.shot.Shot attribute*), 95
enabled (*mpf.plugins.auditor.Auditor attribute*), 30
end_ball() (*mpf.modes.game.code.game.Game method*), 113
end_game() (*mpf.modes.game.code.game.Game method*), 113
ensure_future() (*mpf.core.utility_functions.Util static method*), 217
entrance() (*mpf.devices.ball_device.ball_device.BallDevice method*), 58
eom_resp() (*mpf.platforms.opp.opp.OppHardwarePlatform static method*), 132
error_log() (*mpf.core.ball_search.BallSearch method*), 205
error_log() (*mpf.core.data_manager.DataManager method*), 221
error_log() (*mpf.core.delays.DelayManager method*), 222
error_log() (*mpf.core.logging.LogMixin method*), 208
error_log() (*mpf.core.mode.Mode method*), 209
error_log() (*mpf.modes.attract.code.attract.Attract method*), 103
error_log() (*mpf.modes.bonus.code.bonus.Bonus method*), 105
error_log() (*mpf.modes.carousel.code.carousel.Carousel method*), 108
error_log() (*mpf.modes.credits.code.credits.Credits method*), 110
error_log() (*mpf.modes.game.code.game.Game method*), 113
error_log() (*mpf.modes.high_score.code.high_score.HighScore method*), 116
error_log() (*mpf.modes.match.code.match.Match method*), 118
error_log() (*mpf.modes.service.code.service.Service method*), 120
error_log() (*mpf.modes.tilt.code.tilt.Tilt method*), 122
event_config_to_dict() (*mpf.core.utility_functions.Util static method*), 217
EventManager (*class in mpf.core.events*), 33
EventPlayer (*class in mpf.config_players.event_player*), 148
expected_ball_received() (*mpf.devices.ball_device.ball_device.BallDevice method*), 58
expected_ball_received() (*mpf.devices.playfield.Playfield method*), 85
ExtraBall (*class in mpf.devices.extra_ball*), 72
ExtraBallGroup (*class in mpf.devices.extra_ball_group*), 71

F

fade_in_progress (*mpf.devices.light.Light attribute*), 78
FadecandyHardwarePlatform (*class in mpf.platforms.fadecandy*), 124
fail() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 160
fail() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 170
fail() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 180
fail() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 189
fail() (*mpf.tests.MpfTestCase.MpfTestCase method*), 197
FastHardwarePlatform (*class in mpf.platforms.fast.fast*), 125
FileManager (*class in mpf.core.file_manager*), 207
fill_troughs() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestC method*), 170

fill_troughs() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 180
find_available_ball_in_path() (*mpf.devices.ball_device.ball_device.BallDevice method*), 59
find_next_trough() (*mpf.devices.ball_device.ball_device.BallDevice method*), 59
find_one_available_ball() (*mpf.devices.ball_device.ball_device.BallDevice method*), 59
find_path_to_target() (*mpf.devices.ball_device.ball_device.BallDevice method*), 59
first() (*mpf.core.utility_functions.Util static method*), 217
FlasherPlayer (*class in mpf.config_players.flasher_player*), 148
fling_ball() (*mpf.devices.magnet.Magnet method*), 80
Flipper (*class in mpf.devices.flipper*), 73

G

Game (*class in mpf.modes.game.code.game*), 111
game_ending() (*mpf.modes.game.code.game.Game method*), 113
gamma_correct() (*mpf.devices.light.Light method*), 78
get_active_event_for_switch() (*mpf.core.switch_controller.SwitchController static method*), 48
get_additional_ball_capacity() (*mpf.devices.playfield.Playfield class method*), 85
get_and_verify_hold_power() (*mpf.devices.driver.Driver method*), 64
get_and_verify_pulse_ms() (*mpf.devices.driver.Driver method*), 64
get_and_verify_pulse_power() (*mpf.devices.driver.Driver method*), 64
get_coil_config_section() (*mpf.platforms.fast.fast.FastHardwarePlatform class method*), 126
get_coil_config_section() (*mpf.platforms.opp.opp.OppHardwarePlatform class method*), 132
get_coil_map() (*mpf.core.service_controller.ServiceController method*), 45
get_color() (*mpf.devices.light.Light method*), 78
get_color_below() (*mpf.devices.light.Light method*), 79
get_config_spec() (*mpf.core.mode.Mode static method*), 209
get_current() (*mpf.core.randomizer.Randomizer method*), 216
get_data() (*mpf.core.data_manager.DataManager method*), 221
get_data() (*mpf.tests.TestDataManager.TestDataManager method*), 201
get_device_control_events() (*mpf.core.device_manager.DeviceManager method*), 32
get_enable_plugins() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 161
get_enable_plugins() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 170
get_enable_plugins() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 181
get_enable_plugins() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 190
get_enable_plugins() (*mpf.tests.MpfTestCase.MpfTestCase method*), 198
get_event_and_condition_from_string() (*mpf.core.events.EventManager method*), 34
get_express_config() (*mpf.config_players.block_event_player.BlockEventPlayer method*), 147
get_express_config() (*mpf.config_players.coil_player.CoilPlayer method*), 148
get_express_config() (*mpf.config_players.event_player.EventPlayer method*), 148
get_express_config() (*mpf.config_players.flasher_player.FlasherPlayer method*), 149
get_express_config() (*mpf.config_players.hardware_sound_player.HardwareSoundPlay method*), 149
get_express_config() (*mpf.config_players.light_player.LightPlayer method*), 150
get_express_config() (*mpf.config_players.queue_event_player.QueueEventPlayer method*), 150
get_express_config() (*mpf.config_players.queue_relay_player.QueueRelayPlayer method*), 151
get_express_config() (*mpf.config_players.random_event_player.RandomEventPlayer method*), 151
get_express_config()

(*mpf.config_players.segment_display_player.SegmentDisplayPlayer method*), 152
get_express_config() (*mpf.config_players.show_player.ShowPlayer method*), 152
get_express_config() (*mpf.config_players.variable_player.VariablePlayer method*), 153
get_file_interface() (*mpf.core.file_manager.FileManager static method*), 207
get_from_dict() (*mpf.core.utility_functions.Util static method*), 218
get_gen2_cfg_resp() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 132
get_global_parameters() (*mpf.core.placeholder_manager.PlaceholderManager method*), 43
get_hw_numbers() (*mpf.devices.light.Light method*), 79
get_hw_switch_states() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 126
get_hw_switch_states() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 129
get_hw_switch_states() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 132
get_hw_switch_states() (*mpf.platforms.p3_roc.P3RocHardwarePlatform method*), 135
get_hw_switch_states() (*mpf.platforms.p_roc.PRocHardwarePlatform method*), 136
get_hw_switch_states() (*mpf.platforms.rpi.rpi.RaspberryPiHardwarePlatform method*), 138
get_hw_switch_states() (*mpf.platforms.spike.spike.SpikePlatform method*), 142
get_hw_switch_states() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 145
get_hw_switch_states() (*mpf.platforms.virtual_pinball.virtual_pinball.VirtualPinballHardwarePlatform method*), 146
get_info_string() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 126
get_info_string() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 132
get_info_string() (*mpf.platforms.p3_roc.P3RocHardwarePlatform method*), 135
get_info_string() (*mpf.platforms.p_roc.PRocHardwarePlatform method*), 137
get_level_xyz() (*mpf.devices.accelerometer.Accelerometer method*), 53
get_level_xz() (*mpf.devices.accelerometer.Accelerometer method*), 53
get_level_yz() (*mpf.devices.accelerometer.Accelerometer method*), 53
get_light_map() (*mpf.core.service_controller.ServiceController method*), 45
get_list_config() (*mpf.config_players.event_player.EventPlayer method*), 148
get_list_config() (*mpf.config_players.random_event_player.RandomEventPlayer method*), 151
get_list_config() (*mpf.config_players.variable_player.VariablePlayer method*), 153
get_machine_var() (*mpf.core.machine.MachineController method*), 39
get_machine_var() (*mpf.tests.MpfTestCase.TestMachineController method*), 202
get_monitorable_devices() (*mpf.core.device_manager.DeviceManager method*), 32
get_named_list_from_objects() (*mpf.core.utility_functions.Util static method*), 218
get_next() (*mpf.core.randomizer.Randomizer method*), 216
get_next_show_id() (*mpf.core.show_controller.ShowController method*), 47
get_next_timed_switch_event() (*mpf.core.switch_controller.SwitchController method*), 48
get_platform() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 161
get_platform() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 171
get_platform() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 181
get_platform() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 190
get_platform() (*mpf.tests.MpfTestCase.MpfTestCase method*), 198
get_platform_sections()

(*mpf.core.machine.MachineController method*), 39
get_platform_sections() (*mpf.tests.MpfTestCase.TestMachineController method*), 202
get_setting_machine_var() (*mpf.core.settings_controller.SettingsController method*), 46
get_setting_value() (*mpf.core.settings_controller.SettingsController method*), 46
get_setting_value_label() (*mpf.core.settings_controller.SettingsController method*), 46
get_settings() (*mpf.core.settings_controller.SettingsController method*), 46
get_start_value() (*mpf.devices.logic_blocks.Accrual method*), 54
get_start_value() (*mpf.devices.logic_blocks.Counter method*), 66
get_start_value() (*mpf.devices.logic_blocks.Sequence method*), 91
get_stepper_config_section() (*mpf.platforms.trinamics_steprocker.TrinamicsStepRocker class method*), 143
get_string_config() (*mpf.config_players.hardware_sound_player.HardwareSoundPlayer method*), 149
get_switch_config_section() (*mpf.platforms.fast.fast.FastHardwarePlatform class method*), 126
get_switch_map() (*mpf.core.service_controller.ServiceController method*), 45
get_token() (*mpf.devices.light_group.LightRing method*), 77
get_token() (*mpf.devices.light_group.LightStrip method*), 77
get_use_bcp() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 161
get_use_bcp() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 171
get_use_bcp() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 181
get_use_bcp() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 190
get_use_bcp() (*mpf.tests.MpfTestCase.MpfTestCase method*), 198
get_wait_time_for_pulse() (*mpf.devices.power_supply_unit.PowerSupplyUnit method*), 86
getConfigFile() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 160
getConfigFile() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 170
getConfigFile() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 180
getConfigFile() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 189
getConfigFile() (*mpf.tests.MpfTestCase.MpfTestCase method*), 197
getMachinePath() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 160
getMachinePath() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 170
getMachinePath() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 181
getMachinePath() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 189
getMachinePath() (*mpf.tests.MpfTestCase.MpfTestCase method*), 198
give_up() (*mpf.core.ball_search.BallSearch method*), 205
go_to_position() (*mpf.devices.motor.Motor method*), 81
go_to_position() (*mpf.devices.servo.Servo method*), 92
grab_ball() (*mpf.devices.magnet.Magnet method*), 80
green (*mpf.core.rgb_color.RGBColor attribute*), 215
green (*mpf.core.rgba_color.RGBAColor attribute*), 213
group (*mpf.devices.extra_ball.ExtraBall attribute*), 73

H

handle_mechanial_eject_during_idle() (*mpf.devices.ball_device.ball_device.BallDevice method*), 59
handle_subscription_change() (*mpf.config_players.light_player.LightPlayer method*), 150
handle_subscription_change() (*mpf.config_players.show_player.ShowPlayer method*), 152
handle_switch() (*mpf.plugins.osc.Osc method*), 42
HardwareSoundPlayer (*class in mpf.config_players.hardware_sound_player*), 149
HardwareSoundSystem (*class in mpf.devices.hardware_sound_system*), 75
hex (*mpf.core.rgb_color.RGBColor attribute*), 215
hex (*mpf.core.rgba_color.RGBAColor attribute*), 213
hex_string_to_int() (*mpf.core.utility_functions.Util static method*), 218
hex_string_to_list() (*mpf.core.utility_functions.Util static method*), 218

hex_to_rgb() (*mpf.core.rgb_color.RGBColor static method*), 215
hex_to_rgb() (*mpf.core.rgba_color.RGBAColor static method*), 213
HighScore (*class in mpf.modes.high_score.code.high_score*), 115
hit() (*mpf.devices.logic_blocks.Accrual method*), 54
hit() (*mpf.devices.logic_blocks.Sequence method*), 91
hit() (*mpf.devices.shot.Shot method*), 95
hit_and_release_switch() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 161
hit_and_release_switch() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 171
hit_and_release_switch() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 181
hit_and_release_switch() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 190
hit_and_release_switch() (*mpf.tests.MpfTestCase.MpfTestCase method*), 198
hit_and_release_switches_simultaneously() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 161
hit_and_release_switches_simultaneously() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 171
hit_and_release_switches_simultaneously() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 181
hit_and_release_switches_simultaneously() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 190
hit_and_release_switches_simultaneously() (*mpf.tests.MpfTestCase.MpfTestCase method*), 199
hit_switch_and_run() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 161
hit_switch_and_run() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 171
hit_switch_and_run() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 182
hit_switch_and_run() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 190
hit_switch_and_run() (*mpf.tests.MpfTestCase.MpfTestCase method*), 199
hold() (*mpf.devices.ball_device.ball_device.BallDevice method*), 59
hurry_up() (*mpf.modes.bonus.code.bonus.Bonus method*), 105
hw_state (*mpf.devices.switch.Switch attribute*), 98

I

I2CServoControllerHardwarePlatform (*class in mpf.platforms.i2c_servo_controller*), 127
ignorable_runtime_exception() (*mpf.core.ball_search.BallSearch method*), 205
ignorable_runtime_exception() (*mpf.core.data_manager.DataManager method*), 221
ignorable_runtime_exception() (*mpf.core.delays.DelayManager method*), 222
ignorable_runtime_exception() (*mpf.core.logging.LogMixin method*), 208
ignorable_runtime_exception() (*mpf.core.mode.Mode method*), 209
ignorable_runtime_exception() (*mpf.modes.attract.code.attract.Attract method*), 103
ignorable_runtime_exception() (*mpf.modes.bonus.code.bonus.Bonus method*), 106
ignorable_runtime_exception() (*mpf.modes.carousel.code.carousel.Carousel method*), 108
ignorable_runtime_exception() (*mpf.modes.credits.code.credits.Credits method*), 110
ignorable_runtime_exception() (*mpf.modes.game.code.game.Game method*), 113
ignorable_runtime_exception() (*mpf.modes.high_score.code.high_score.HighScore method*), 116
ignorable_runtime_exception() (*mpf.modes.match.code.match.Match method*), 118
ignorable_runtime_exception() (*mpf.modes.service.code.service.Service method*), 120
ignorable_runtime_exception() (*mpf.modes.tilt.code.tilt.Tilt method*), 123
ignorable_runtime_exception() (*mpf.tests.MpfBcpTestCase.MockBcpClient method*), 154
ignorable_runtime_exception() (*mpf.tests.MpfTestCase.TestMachineController*

method), 202
ignorable_runtime_exception() (*mpf.tests.TestDataManager.TestDataManager method*), 201
increase_volume() (*mpf.devices.hardware_sound_system.HardwareSoundSystem method*), 75
info_log() (*mpf.core.ball_search.BallSearch method*), 205
info_log() (*mpf.core.data_manager.DataManager method*), 221
info_log() (*mpf.core.delays.DelayManager method*), 223
info_log() (*mpf.core.logging.LogMixin method*), 208
info_log() (*mpf.core.mode.Mode method*), 209
info_log() (*mpf.modes.attract.code.attract.Attract method*), 103
info_log() (*mpf.modes.bonus.code.bonus.Bonus method*), 106
info_log() (*mpf.modes.carousel.code.carousel.Carousel method*), 108
info_log() (*mpf.modes.credits.code.credits.Credits method*), 110
info_log() (*mpf.modes.game.code.game.Game method*), 113
info_log() (*mpf.modes.high_score.code.high_score.HighScore method*), 116
info_log() (*mpf.modes.match.code.match.Match method*), 118
info_log() (*mpf.modes.service.code.service.Service method*), 120
info_log() (*mpf.modes.tilt.code.tilt.Tilt method*), 123
InfoLights (*class in mpf.plugins.info_lights*), 37
init() (*mpf.core.file_manager.FileManager class method*), 207
init_done() (*mpf.core.machine.MachineController method*), 39
init_done() (*mpf.tests.MpfTestCase.TestMachineController method*), 202
initialise() (*mpf.core.machine.MachineController method*), 39
initialise() (*mpf.tests.MpfTestCase.TestMachineController method*), 202
initialise_core_and_hardware() (*mpf.core.machine.MachineController method*), 39
initialise_core_and_hardware() (*mpf.tests.MpfTestCase.TestMachineController method*), 202
initialise_light_subsystem() (*mpf.core.light_controller.LightController method*), 38
initialise_mode() (*mpf.core.mode.Mode method*), 210
initialise_mode() (*mpf.modes.attract.code.attract.Attract method*), 103
initialise_mode() (*mpf.modes.bonus.code.bonus.Bonus method*), 106
initialise_mode() (*mpf.modes.carousel.code.carousel.Carousel method*), 108
initialise_mode() (*mpf.modes.credits.code.credits.Credits method*), 110
initialise_mode() (*mpf.modes.game.code.game.Game method*), 113
initialise_mode() (*mpf.modes.high_score.code.high_score.HighScore method*), 116
initialise_mode() (*mpf.modes.match.code.match.Match method*), 118
initialise_mode() (*mpf.modes.service.code.service.Service method*), 120
initialise_mode() (*mpf.modes.tilt.code.tilt.Tilt method*), 123
initialise_modes() (*mpf.core.mode_controller.ModeController method*), 41
initialise_mpf() (*mpf.core.machine.MachineController method*), 39
initialise_mpf() (*mpf.tests.MpfTestCase.TestMachineController method*), 202
initialize() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 126
initialize() (*mpf.platforms.i2c_servo_controller.I2CServoControllerI method*), 128
initialize() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 129
initialize() (*mpf.platforms.mma8451.MMA8451Platform method*), 130
initialize() (*mpf.platforms.mypinballs.mypinballs.MyPinballsHardw method*), 130
initialize() (*mpf.platforms.openpixel.OpenpixelHardwarePlatform method*), 131
initialize() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 132
initialize() (*mpf.platforms.pololu_maestro.PololuMaestroHardwareI method*), 137
initialize() (*mpf.platforms.rpi.rpi.RaspberryPiHardwarePlatform method*), 138
initialize() (*mpf.platforms.smartmatrix.SmartMatrixHardwarePlatfo method*), 139
initialize() (*mpf.platforms.smbus2.Smbus2*

method), 140
initialize() (*mpf.platforms.snux.SnuxHardwarePlatform method*), 141
initialize() (*mpf.platforms.spike.spike.SpikePlatform method*), 142
initialize() (*mpf.platforms.trinamics_steprocker.TrinamicsStepRocker method*), 143
initialize() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 145
initialize() (*mpf.platforms.virtual_pinball.virtual_pinball.VirtualPinballPlatform method*), 146
initialize_devices() (*mpf.core.device_manager.DeviceManager method*), 33
int_to_hex_string() (*mpf.core.utility_functions.Util static method*), 218
int_to_reel_list() (*mpf.devices.score_reel_group.ScoreReelGroup method*), 87
inv_resp() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 132
is_active() (*mpf.core.mode_controller.ModeController method*), 41
is_active() (*mpf.core.switch_controller.SwitchController method*), 48
is_entry_valid_outside_mode() (*mpf.config_players.random_event_player.RandomEventPlayer static method*), 151
is_entry_valid_outside_mode() (*mpf.config_players.variable_player.VariablePlayer static method*), 153
is_full() (*mpf.devices.ball_hold.BallHold method*), 60
is_full() (*mpf.devices.ball_lock.BallLock method*), 61
is_game_mode (*mpf.core.mode.Mode attribute*), 210
is_game_mode (*mpf.modes.attract.code.attract.Attract attribute*), 103
is_game_mode (*mpf.modes.bonus.code.bonus.Bonus attribute*), 106
is_game_mode (*mpf.modes.carousel.code.carousel.Carousel attribute*), 108
is_game_mode (*mpf.modes.credits.code.credits.Credits attribute*), 111
is_game_mode (*mpf.modes.game.code.game.Game attribute*), 113
is_game_mode (*mpf.modes.high_score.code.high_score.HighScore attribute*), 116
is_game_mode (*mpf.modes.match.code.match.Match attribute*), 118
is_game_mode (*mpf.modes.service.code.service.Service attribute*), 120
is_game_mode (*mpf.modes.tilt.code.tilt.Tilt attribute*), 123
is_hex_string() (*mpf.core.utility_functions.Util static method*), 218
is_in_service() (*mpf.core.service_controller.ServiceController method*), 45
is_inactive() (*mpf.core.switch_controller.SwitchController method*), 48
is_machine_var() (*mpf.core.machine.MachineController method*), 39
is_mock() (*mpf.tests.MpfTestCase.TestMachineController method*), 203
is_ok_to_award() (*mpf.devices.extra_ball.ExtraBall method*), 73
is_ok_to_light() (*mpf.devices.extra_ball.ExtraBall method*), 73
is_ok_to_light() (*mpf.devices.extra_ball_group.ExtraBallGroup method*), 72
is_player_var() (*mpf.core.player.Player method*), 212
is_playfield() (*mpf.devices.ball_device.ball_device.BallDevice class method*), 59
is_playfield() (*mpf.devices.playfield.Playfield class method*), 85
is_power2() (*mpf.core.utility_functions.Util static method*), 218
is_state() (*mpf.core.switch_controller.SwitchController method*), 49
is_virtually_full (*mpf.devices.multiball_lock.MultiballLock attribute*), 82
iteration (*mpf.core.ball_search.BallSearch attribute*), 205

J

jump() (*mpf.devices.shot.Shot method*), 95
jump() (*mpf.devices.timer.Timer method*), 100

K

keys_to_lower() (*mpf.core.utility_functions.Util static method*), 218
Kickback (*class in mpf.devices.kickback*), 75
knockdown() (*mpf.devices.drop_target.DropTarget method*), 70

L

Light (*class in mpf.devices.light*), 77
light() (*mpf.devices.extra_ball.ExtraBall method*), 73
light() (*mpf.devices.extra_ball_group.ExtraBallGroup method*), 72
light() (*mpf.devices.score_reel_group.ScoreReelGroup method*), 88
light_sync() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 132

LightController (*class in mpf.core.light_controller*), 38
LightPlayer (*class in mpf.config_players.light_player*), 149
LightRing (*class in mpf.devices.light_group*), 76
LightStrip (*class in mpf.devices.light_group*), 77
list_of_lists() (*mpf.core.utility_functions.Util static method*), 219
LisyHardwarePlatform (*class in mpf.platforms.lisy.lisy*), 128
load() (*mpf.core.file_manager.FileManager static method*), 207
load_asset() (*mpf.core.assets.AsyncioSyncAssetManager method*), 29
load_devices_config() (*mpf.core.device_manager.DeviceManager method*), 33
load_mode_devices() (*mpf.core.mode.Mode method*), 210
load_mode_devices() (*mpf.core.mode_controller.ModeController method*), 41
load_mode_devices() (*mpf.modes.attract.code.attract.Attract method*), 103
load_mode_devices() (*mpf.modes.bonus.code.bonus.Bonus method*), 106
load_mode_devices() (*mpf.modes.carousel.code.carousel.Carousel method*), 108
load_mode_devices() (*mpf.modes.credits.code.credits.Credits method*), 111
load_mode_devices() (*mpf.modes.game.code.game.Game method*), 114
load_mode_devices() (*mpf.modes.high_score.code.high_score.HighScore method*), 116
load_mode_devices() (*mpf.modes.match.code.match.Match method*), 118
load_mode_devices() (*mpf.modes.service.code.service.Service method*), 121
load_mode_devices() (*mpf.modes.tilt.code.tilt.Tilt method*), 123
load_modes() (*mpf.core.mode_controller.ModeController method*), 41
locate_file() (*mpf.core.file_manager.FileManager static method*), 207
locked_balls (*mpf.devices.multiball_lock.MultiballLock attribute*), 82
log_active_switches() (*mpf.core.switch_controller.SwitchController method*), 49
LogMixin (*class in mpf.core.logging*), 207
loop (*mpf.core.randomizer.Randomizer attribute*), 216
lost_ejected_ball() (*mpf.devices.ball_device.ball_device.BallDevice method*), 59
lost_idle_ball() (*mpf.devices.ball_device.ball_device.BallDevice method*), 59
lost_incoming_ball() (*mpf.devices.ball_device.ball_device.BallDevice method*), 59

M

machine (*mpf.platforms.p3_roc.P3RocHardwarePlatform attribute*), 135
machine (*mpf.platforms.p_roc.PRocHardwarePlatform attribute*), 136
machine_run() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 162
machine_run() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCa method*), 171
machine_run() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 182
machine_run() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 190
machine_run() (*mpf.tests.MpfTestCase.MpfTestCase method*), 199
MachineController (*class in mpf.core.machine*), 38
Magnet (*class in mpf.devices.magnet*), 80
mark_playfield_active_from_device_action() (*mpf.devices.playfield.Playfield method*), 86
Match (*class in mpf.modes.match.code.match*), 117
member_state_changed() (*mpf.devices.achievement_group.AchievementGroup method*), 55
member_target_change() (*mpf.devices.drop_target.DropTargetBank method*), 69
MMA8451Platform (*class in mpf.platforms.mma8451*), 130
mock_event() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 162
mock_event() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCas method*), 172
mock_event() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 182
mock_event() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 191
mock_event() (*mpf.tests.MpfTestCase.MpfTestCase method*), 199
MockBcpClient (*class in mpf.tests.MpfBcpTestCase*), 154

Mode (*class in mpf.core.mode*), 208
mode_init() (*mpf.core.mode.Mode method*), 210
mode_start() (*mpf.core.mode.Mode method*), 210
mode_stop() (*mpf.core.mode.Mode method*), 210
mode_will_start() (*mpf.core.mode.Mode method*), 210
mode_will_start() (*mpf.modes.attract.code.attract.Attract method*), 103
mode_will_start() (*mpf.modes.bonus.code.bonus.Bonus method*), 106
mode_will_start() (*mpf.modes.carousel.code.carousel.Carousel method*), 108
mode_will_start() (*mpf.modes.credits.code.credits.Credits method*), 111
mode_will_start() (*mpf.modes.game.code.game.Game method*), 114
mode_will_start() (*mpf.modes.high_score.code.high_score.HighScore method*), 116
mode_will_start() (*mpf.modes.match.code.match.Match method*), 118
mode_will_start() (*mpf.modes.service.code.service.Service method*), 121
mode_will_start() (*mpf.modes.tilt.code.tilt.Tilt method*), 123
ModeController (*class in mpf.core.mode_controller*), 41
monitor_enabled (*mpf.core.player.Player attribute*), 212
monitor_enabled (*mpf.devices.shot.Shot attribute*), 95
monitor_lights() (*mpf.core.light_controller.LightController method*), 38
Motor (*class in mpf.devices.motor*), 81
MpfBcpTestCase (*class in mpf.tests.MpfBcpTestCase*), 155
MpfFakeGameTestCase (*class in mpf.tests.MpfFakeGameTestCase*), 163
MpfGameTestCase (*class in mpf.tests.MpfGameTestCase*), 174
MpfMachineTestCase (*class in mpf.tests.MpfMachineTestCase*), 184
MpfTestCase (*class in mpf.tests.MpfTestCase*), 192
ms_since_change() (*mpf.core.switch_controller.SwitchController method*), 49
Multiball (*class in mpf.devices.multiball*), 82
MultiballLock (*class in mpf.devices.multiball_lock*), 81
MyPinballsHardwarePlatform (*class in mpf.platforms.mypinballs.mypinballs*), 130

N

name (*mpf.core.rgb_color.RGBColor attribute*), 215
name (*mpf.core.rgba_color.RGBAColor attribute*), 213
name_to_rgb() (*mpf.core.rgb_color.RGBColor static method*), 215
name_to_rgb() (*mpf.core.rgba_color.RGBAColor static method*), 213
normalize_hex_string() (*mpf.core.utility_functions.Util static method*), 219
notify_about_instant_pulse() (*mpf.devices.power_supply_unit.PowerSupplyUnit method*), 86
notify_device_changes() (*mpf.core.device_manager.DeviceManager method*), 33

O

off() (*mpf.devices.light.Light method*), 79
on() (*mpf.devices.light.Light method*), 79
OpenpixelHardwarePlatform (*class in mpf.platforms.openpixel*), 131
OppHardwarePlatform (*class in mpf.platforms.opp.opp*), 131
Osc (*class in mpf.plugins.osc*), 42

P

P3RocHardwarePlatform (*class in mpf.platforms.p3_roc*), 135
parse_light_number_to_channels() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 126
parse_light_number_to_channels() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 129
parse_light_number_to_channels() (*mpf.platforms.openpixel.OpenpixelHardwarePlatform method*), 131
parse_light_number_to_channels() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 133
parse_light_number_to_channels() (*mpf.platforms.spike.spike.SpikePlatform method*), 142
parse_light_number_to_channels() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 145

parse_light_number_to_channels() (*mpf.platforms.virtual_pinball.virtual_pinball.VirtualPinballPlatform method*), 146
pause() (*mpf.devices.timer.Timer method*), 100
persist_enabled (*mpf.devices.shot.Shot attribute*), 95
phase (*mpf.core.ball_search.BallSearch attribute*), 205
pick_weighted_random() (*mpf.core.randomizer.Randomizer static method*), 216
PlaceholderManager (*class in mpf.core.placeholder_manager*), 43
PlatformController (*class in mpf.core.platform_controller*), 43
PlatformController.DriverRuleSettings (*class in mpf.core.platform_controller*), 45
PlatformController.HoldRuleSettings (*class in mpf.core.platform_controller*), 44
PlatformController.PulseRuleSettings (*class in mpf.core.platform_controller*), 44, 45
play() (*mpf.config_players.block_event_player.BlockEventPlayer method*), 147
play() (*mpf.config_players.coil_player.CoilPlayer method*), 148
play() (*mpf.config_players.event_player.EventPlayer method*), 148
play() (*mpf.config_players.flasher_player.FlasherPlayer method*), 149
play() (*mpf.config_players.hardware_sound_player.HardwareSoundPlayer method*), 149
play() (*mpf.config_players.light_player.LightPlayer method*), 150
play() (*mpf.config_players.queue_event_player.QueueEventPlayer method*), 150
play() (*mpf.config_players.queue_relay_player.QueueRelayPlayer method*), 151
play() (*mpf.config_players.random_event_player.RandomEventPlayer method*), 151
play() (*mpf.config_players.segment_display_player.SegmentDisplayPlayer method*), 152
play() (*mpf.config_players.show_player.ShowPlayer method*), 152
play() (*mpf.config_players.variable_player.VariablePlayer method*), 153
play() (*mpf.devices.hardware_sound_system.HardwareSoundSystem method*), 75
play_file() (*mpf.devices.hardware_sound_system.HardwareSoundSystem method*), 75
play_show_with_config() (*mpf.core.show_controller.ShowController method*), 47
Player (*class in mpf.core.player*), 211
player (*mpf.core.mode.Mode attribute*), 210
player (*mpf.devices.extra_ball.ExtraBall attribute*), 73
Playfield (*class in mpf.devices.playfield*), 84
playfield (*mpf.core.ball_search.BallSearch attribute*), 206
PlayfieldTransfer (*class in mpf.devices.playfield_transfer*), 83
PololuMaestroHardwarePlatform (*class in mpf.platforms.pololu_maestro*), 137
post() (*mpf.core.events.EventManager method*), 34
post_async() (*mpf.core.events.EventManager method*), 35
post_boolean() (*mpf.core.events.EventManager method*), 35
post_event() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 162
post_event() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 172
post_event() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 182
post_event() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 191
post_event() (*mpf.tests.MpfTestCase.MpfTestCase method*), 199
post_event_with_params() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 162
post_event_with_params() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 172
post_event_with_params() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 182
post_event_with_params() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 191
post_event_with_params() (*mpf.tests.MpfTestCase.MpfTestCase method*), 200
post_queue() (*mpf.core.events.EventManager method*), 35
post_queue_async() (*mpf.core.events.EventManager method*), 36
post_relay() (*mpf.core.events.EventManager method*), 36
post_relay_async() (*mpf.core.events.EventManager method*), 36
power_to_on_off() (*mpf.core.utility_functions.Util static method*), 219
PowerSupplyUnit (*class in mpf.devices.power_supply_unit*), 86
process_event_queue() (*mpf.core.events.EventManager method*), 36

process_received_message() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 126
process_received_message() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 133
process_switch() (*mpf.core.switch_controller.SwitchController method*), 49
process_switch_by_num() (*mpf.core.switch_controller.SwitchController method*), 49
process_switch_obj() (*mpf.core.switch_controller.SwitchController method*), 50
PRocHardwarePlatform (*class in mpf.platforms.p_roc*), 136
profile (*mpf.devices.shot.Shot attribute*), 96
profile_name (*mpf.devices.shot.Shot attribute*), 96
pulse() (*mpf.devices.driver.Driver method*), 64
pulse() (*mpf.devices.dual_wound_coil.DualWoundCoil method*), 71
pwm32_to_hex_string() (*mpf.core.utility_functions.Util static method*), 219
pwm32_to_int() (*mpf.core.utility_functions.Util static method*), 219
pwm8_to_hex_string() (*mpf.core.utility_functions.Util static method*), 219
pwm8_to_int() (*mpf.core.utility_functions.Util static method*), 219

Q

QueueEventPlayer (*class in mpf.config_players.queue_event_player*), 150
QueueRelayPlayer (*class in mpf.config_players.queue_relay_player*), 150

R

race() (*mpf.core.utility_functions.Util static method*), 219
raise_config_error() (*mpf.core.ball_search.BallSearch method*), 206
raise_config_error() (*mpf.core.data_manager.DataManager method*), 221
raise_config_error() (*mpf.core.delays.DelayManager method*), 223
raise_config_error() (*mpf.core.logging.LogMixin method*), 208
raise_config_error() (*mpf.core.mode.Mode method*), 210
raise_config_error() (*mpf.devices.accelerometer.Accelerometer method*), 53
raise_config_error() (*mpf.devices.achievement.Achievement method*), 56
raise_config_error() (*mpf.devices.achievement_group.AchievementGroup method*), 55
raise_config_error() (*mpf.devices.autofire.AutofireCoil method*), 57
raise_config_error() (*mpf.devices.ball_device.ball_device.BallDevice method*), 59
raise_config_error() (*mpf.devices.ball_hold.BallHold method*), 60
raise_config_error() (*mpf.devices.ball_lock.BallLock method*), 61
raise_config_error() (*mpf.devices.ball_save.BallSave method*), 63
raise_config_error() (*mpf.devices.combo_switch.ComboSwitch method*), 65
raise_config_error() (*mpf.devices.digital_output.DigitalOutput method*), 66
raise_config_error() (*mpf.devices.diverter.Diverter method*), 68
raise_config_error() (*mpf.devices.dmd.Dmd method*), 68
raise_config_error() (*mpf.devices.driver.Driver method*), 64
raise_config_error() (*mpf.devices.drop_target.DropTarget method*), 70
raise_config_error() (*mpf.devices.drop_target.DropTargetBank method*), 69
raise_config_error() (*mpf.devices.dual_wound_coil.DualWoundCoil method*), 71
raise_config_error() (*mpf.devices.extra_ball.ExtraBall method*), 73
raise_config_error() (*mpf.devices.extra_ball_group.ExtraBallGroup method*), 72
raise_config_error()

(*mpf.devices.flipper.Flipper method*), 74
raise_config_error() (*mpf.devices.hardware_sound_system.HardwareSoundSystem method*), 75
raise_config_error() (*mpf.devices.kickback.Kickback method*), 76
raise_config_error() (*mpf.devices.light.Light method*), 79
raise_config_error() (*mpf.devices.light_group.LightRing method*), 77
raise_config_error() (*mpf.devices.light_group.LightStrip method*), 77
raise_config_error() (*mpf.devices.logic_blocks.Accrual method*), 54
raise_config_error() (*mpf.devices.logic_blocks.Counter method*), 66
raise_config_error() (*mpf.devices.logic_blocks.Sequence method*), 91
raise_config_error() (*mpf.devices.magnet.Magnet method*), 80
raise_config_error() (*mpf.devices.motor.Motor method*), 81
raise_config_error() (*mpf.devices.multiball.Multiball method*), 83
raise_config_error() (*mpf.devices.multiball_lock.MultiballLock method*), 82
raise_config_error() (*mpf.devices.playfield.Playfield method*), 86
raise_config_error() (*mpf.devices.playfield_transfer.PlayfieldTransfer method*), 83
raise_config_error() (*mpf.devices.power_supply_unit.PowerSupplyUnit method*), 86
raise_config_error() (*mpf.devices.rgb_dmd.RgbDmd method*), 87
raise_config_error() (*mpf.devices.score_reel.ScoreReel method*), 89
raise_config_error() (*mpf.devices.score_reel_group.ScoreReelGroup method*), 88
raise_config_error() (*mpf.devices.segment_display.SegmentDisplay method*), 90
raise_config_error() (*mpf.devices.sequence_shot.SequenceShot method*), 90
raise_config_error() (*mpf.devices.servo.Servo method*), 92
raise_config_error() (*mpf.devices.shot.Shot method*), 96
raise_config_error() (*mpf.devices.shot_group.ShotGroup method*), 93
raise_config_error() (*mpf.devices.shot_profile.ShotProfile method*), 94
raise_config_error() (*mpf.devices.state_machine.StateMachine method*), 96
raise_config_error() (*mpf.devices.stepper.Stepper method*), 97
raise_config_error() (*mpf.devices.switch.Switch method*), 98
raise_config_error() (*mpf.devices.timed_switch.TimedSwitch method*), 99
raise_config_error() (*mpf.devices.timer.Timer method*), 100
raise_config_error() (*mpf.modes.attract.code.attract.Attract method*), 103
raise_config_error() (*mpf.modes.bonus.code.bonus.Bonus method*), 106
raise_config_error() (*mpf.modes.carousel.code.carousel.Carousel method*), 108
raise_config_error() (*mpf.modes.credits.code.credits.Credits method*), 111
raise_config_error() (*mpf.modes.game.code.game.Game method*), 114
raise_config_error() (*mpf.modes.high_score.code.high_score.HighScore method*), 116
raise_config_error() (*mpf.modes.match.code.match.Match method*), 118
raise_config_error() (*mpf.modes.service.code.service.Service method*), 121
raise_config_error() (*mpf.modes.tilt.code.tilt.Tilt method*), 123
raise_config_error() (*mpf.tests.MpfBcpTestCase.MockBcpClient method*), 155
raise_config_error() (*mpf.tests.MpfTestCase.TestMachineController*

method), 203
raise_config_error() (*mpf.tests.TestDataManager.TestDataManager method*), 201
random_rgb() (*mpf.core.rgb_color.RGBColor static method*), 215
random_rgb() (*mpf.core.rgba_color.RGBAColor static method*), 213
RandomEventPlayer (*class in mpf.config_players.random_event_player*), 151
Randomizer (*class in mpf.core.randomizer*), 216
RaspberryPiHardwarePlatform (*class in mpf.platforms.rpi.rpi*), 137
read_byte() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 129
read_gen2_inp_resp() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 133
read_gen2_inp_resp_initial() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 133
read_matrix_inp_resp() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 133
read_matrix_inp_resp_initial() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 133
read_message() (*mpf.tests.MpfBcpTestCase.MockBcpClient method*), 155
read_string() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 129
readuntil() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 129
receive_local_closed() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 126
receive_local_open() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 126
receive_nw_closed() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 126
receive_nw_open() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 126
receive_sa() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 126
red (*mpf.core.rgb_color.RGBColor attribute*), 215
red (*mpf.core.rgba_color.RGBAColor attribute*), 213
register() (*mpf.core.ball_search.BallSearch method*), 206
register_boot_hold() (*mpf.core.machine.MachineController method*), 39
register_boot_hold() (*mpf.tests.MpfTestCase.TestMachineController method*), 203
register_io_board() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 126
register_load_method() (*mpf.core.mode_controller.ModeController method*), 41
register_monitor() (*mpf.core.machine.MachineController method*), 39
register_monitor() (*mpf.tests.MpfTestCase.TestMachineController method*), 203
register_monitorable_device() (*mpf.core.device_manager.DeviceManager method*), 33
register_processor_connection() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 127
register_processor_connection() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 133
register_show() (*mpf.core.show_controller.ShowController method*), 47
register_start_method() (*mpf.core.mode_controller.ModeController method*), 42
register_stop_method() (*mpf.core.mode_controller.ModeController method*), 42
register_switch() (*mpf.core.switch_controller.SwitchController method*), 50
release_all() (*mpf.devices.ball_hold.BallHold method*), 60
release_all_balls() (*mpf.devices.ball_lock.BallLock method*), 62
release_ball() (*mpf.devices.magnet.Magnet method*), 80
release_balls() (*mpf.devices.ball_hold.BallHold method*), 60
release_balls() (*mpf.devices.ball_lock.BallLock method*), 62
release_one() (*mpf.devices.ball_hold.BallHold method*), 61
release_one() (*mpf.devices.ball_lock.BallLock method*), 62
release_one_if_full() (*mpf.devices.ball_hold.BallHold method*), 61

release_one_if_full() (*mpf.devices.ball_lock.BallLock method*), 62
release_switch_and_run() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 162
release_switch_and_run() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 172
release_switch_and_run() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 183
release_switch_and_run() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 191
release_switch_and_run() (*mpf.tests.MpfTestCase.MpfTestCase method*), 200
remaining_space_in_hold() (*mpf.devices.ball_hold.BallHold method*), 61
remaining_space_in_lock() (*mpf.devices.ball_lock.BallLock method*), 62
remaining_virtual_space_in_lock (*mpf.devices.multiball_lock.MultiballLock attribute*), 82
remove() (*mpf.core.delays.DelayManager method*), 223
remove_all_handlers_for_event() (*mpf.core.events.EventManager method*), 36
remove_from_bank() (*mpf.devices.drop_target.DropTarget method*), 70
remove_from_group() (*mpf.devices.achievement.Achievement method*), 56
remove_from_stack_by_key() (*mpf.devices.light.Light method*), 79
remove_handler() (*mpf.core.events.EventManager method*), 36
remove_handler() (*mpf.devices.switch.Switch method*), 98
remove_handler_by_event() (*mpf.core.events.EventManager method*), 36
remove_handler_by_key() (*mpf.core.events.EventManager method*), 36
remove_handlers_by_keys() (*mpf.core.events.EventManager method*), 37
remove_incoming_ball() (*mpf.devices.ball_device.ball_device.BallDevice method*), 59
remove_incoming_ball() (*mpf.devices.playfield.Playfield method*), 86
remove_machine_var() (*mpf.core.machine.MachineController method*), 40
remove_machine_var() (*mpf.tests.MpfTestCase.TestMachineController method*), 203
remove_machine_var_search() (*mpf.core.machine.MachineController method*), 40
remove_machine_var_search() (*mpf.tests.MpfTestCase.TestMachineController method*), 203
remove_monitor() (*mpf.core.switch_controller.SwitchController method*), 50
remove_start_method() (*mpf.core.mode_controller.ModeController method*), 42
remove_stop_method() (*mpf.core.mode_controller.ModeController method*), 42
remove_switch_handler() (*mpf.core.switch_controller.SwitchController method*), 50
remove_switch_handler_by_key() (*mpf.core.switch_controller.SwitchController method*), 50
remove_switch_handler_obj() (*mpf.core.switch_controller.SwitchController method*), 50
remove_text_by_key() (*mpf.devices.segment_display.SegmentDisplay method*), 90
replace_handler() (*mpf.core.events.EventManager method*), 37
replace_or_advance_show() (*mpf.core.show_controller.ShowController method*), 47
request_ball() (*mpf.devices.ball_device.ball_device.BallDevice method*), 59
request_player_add() (*mpf.modes.game.code.game.Game method*), 114
request_to_start_game() (*mpf.core.ball_controller.BallController method*), 31
request_to_start_game() (*mpf.core.ball_search.BallSearch method*), 206

requested_balls (*mpf.devices.ball_device.ball_device.BallDevice attribute*), 59
reset() (*mpf.core.delays.DelayManager method*), 223
reset() (*mpf.core.machine.MachineController method*), 40
reset() (*mpf.devices.achievement.Achievement method*), 56
reset() (*mpf.devices.ball_hold.BallHold method*), 61
reset() (*mpf.devices.ball_lock.BallLock method*), 62
reset() (*mpf.devices.diverter.Diverter method*), 68
reset() (*mpf.devices.drop_target.DropTarget method*), 70
reset() (*mpf.devices.drop_target.DropTargetBank method*), 69
reset() (*mpf.devices.logic_blocks.Accrual method*), 54
reset() (*mpf.devices.logic_blocks.Counter method*), 66
reset() (*mpf.devices.logic_blocks.Sequence method*), 91
reset() (*mpf.devices.magnet.Magnet method*), 80
reset() (*mpf.devices.motor.Motor method*), 81
reset() (*mpf.devices.multiball.Multiball method*), 83
reset() (*mpf.devices.servo.Servo method*), 92
reset() (*mpf.devices.shot.Shot method*), 96
reset() (*mpf.devices.shot_group.ShotGroup method*), 93
reset() (*mpf.devices.stepper.Stepper method*), 97
reset() (*mpf.devices.timer.Timer method*), 100
reset() (*mpf.tests.MpfTestCase.TestMachineController method*), 203
reset_all_counts() (*mpf.devices.multiball_lock.MultiballLock method*), 82
reset_count_for_current_player() (*mpf.devices.multiball_lock.MultiballLock method*), 82
reset_mock_events() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 163
reset_mock_events() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 172
reset_mock_events() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 183
reset_mock_events() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 191
reset_mock_events() (*mpf.tests.MpfTestCase.MpfTestCase method*), 200
reset_timer() (*mpf.core.ball_search.BallSearch method*), 206
resetWarnings() (*mpf.modes.tilt.code.tilt.Tilt method*), 123
restart() (*mpf.devices.logic_blocks.Accrual method*), 54
restart() (*mpf.devices.logic_blocks.Counter method*), 66
restart() (*mpf.devices.logic_blocks.Sequence method*), 91
restart() (*mpf.devices.shot.Shot method*), 96
restart() (*mpf.devices.shot_group.ShotGroup method*), 93
restart() (*mpf.devices.timer.Timer method*), 100
restart_on_next_ball (*mpf.core.mode.Mode attribute*), 210
result_of_start_request() (*mpf.modes.attract.code.attract.Attract method*), 103
rgb (*mpf.core.rgb_color.RGBColor attribute*), 215
rgb (*mpf.core.rgba_color.RGBAColor attribute*), 213
rgb_to_hex() (*mpf.core.rgb_color.RGBColor static method*), 215
rgb_to_hex() (*mpf.core.rgba_color.RGBAColor static method*), 213
rgba (*mpf.core.rgba_color.RGBAColor attribute*), 214
RGBAColor (*class in mpf.core.rgba_color*), 212
RGBColor (*class in mpf.core.rgb_color*), 214
RgbDmd (*class in mpf.devices.rgb_dmd*), 86
rotate() (*mpf.devices.shot_group.ShotGroup method*), 93
rotate_left() (*mpf.devices.achievement_group.AchievementGroup method*), 55
rotate_left() (*mpf.devices.shot_group.ShotGroup method*), 94
rotate_right() (*mpf.devices.achievement_group.AchievementGroup method*), 55
rotate_right() (*mpf.devices.shot_group.ShotGroup method*), 94
run() (*mpf.core.machine.MachineController method*), 40
run() (*mpf.tests.MpfTestCase.TestMachineController method*), 203
runNow() (*mpf.core.delays.DelayManager method*), 223

S

save() (*mpf.core.file_manager.FileManager static method*), 207
save_all() (*mpf.core.data_manager.DataManager method*), 221
save_all() (*mpf.tests.TestDataManager.TestDataManager method*), 201
scale_accelerometer_to_g() (*mpf.platforms.p3_roc.P3RocHardwarePlatform class method*), 135

schedule_deactivation() (*mpf.devices.diverter.Diverter method*), 68
ScoreReel (*class in mpf.devices.score_reel*), 88
ScoreReelGroup (*class in mpf.devices.score_reel_group*), 87
SegmentDisplay (*class in mpf.devices.segment_display*), 89
SegmentDisplayPlayer (*class in mpf.config_players.segment_display_player*), 152
select() (*mpf.devices.achievement.Achievement method*), 56
select_random_achievement() (*mpf.devices.achievement_group.AchievementGroup method*), 55
send() (*mpf.core.bcp.bcp.Bcp method*), 32
send() (*mpf.tests.MpfBcpTestCase.MockBcpClient method*), 155
send_all_variable_events() (*mpf.core.player.Player method*), 212
send_byte() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 129
send_cmd() (*mpf.platforms.mypinballs.mypinballs.MyPinballsHardwarePlatform method*), 130
send_cmd_and_wait_for_response() (*mpf.platforms.spike.spike.SpikePlatform method*), 142
send_cmd_async() (*mpf.platforms.spike.spike.SpikePlatform method*), 142
send_cmd_raw() (*mpf.platforms.spike.spike.SpikePlatform method*), 142
send_cmd_sync() (*mpf.platforms.spike.spike.SpikePlatform method*), 142
send_command() (*mpf.platforms.rpi.rpi.RaspberryPiHardwarePlatform method*), 138
send_string() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 129
send_to_processor() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 133
Sequence (*class in mpf.devices.logic_blocks*), 91
SequenceShot (*class in mpf.devices.sequence_shot*), 90
Service (*class in mpf.modes.service.code.service*), 119
ServiceController (*class in mpf.core.service_controller*), 45
Servo (*class in mpf.devices.servo*), 92
set_acceleration_limit() (*mpf.devices.servo.Servo method*), 92
set_default_platform() (*mpf.core.machine.MachineController method*), 40
set_default_platform() (*mpf.tests.MpfTestCase.TestMachineController method*), 203
set_destination_value() (*mpf.devices.score_reel.ScoreReel method*), 89
set_eject_state() (*mpf.devices.ball_device.ball_device.BallDevice method*), 59
set_flashing() (*mpf.devices.segment_display.SegmentDisplay method*), 90
set_in_dict() (*mpf.core.utility_functions.Util static method*), 219
set_machine_var() (*mpf.core.machine.MachineController method*), 40
set_machine_var() (*mpf.tests.MpfTestCase.TestMachineController method*), 203
set_mode_state() (*mpf.core.mode_controller.ModeController method*), 42
set_num_balls_known() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 163
set_num_balls_known() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 173
set_num_balls_known() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 183
set_num_balls_known() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 192
set_num_balls_known() (*mpf.tests.MpfTestCase.MpfTestCase method*), 200
set_pulse_on_hit_and_enable_and_release_and_disable() (*mpf.core.platform_controller.PlatformController method*), 43
set_pulse_on_hit_and_enable_and_release_and_disable() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 127
set_pulse_on_hit_and_enable_and_release_and_disable() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 129
set_pulse_on_hit_and_enable_and_release_and_disable() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 134
set_pulse_on_hit_and_enable_and_release_and_disable() (*mpf.platforms.rpi.rpi.RaspberryPiHardwarePlatform method*), 138
set_pulse_on_hit_and_enable_and_release_and_disable() (*mpf.platforms.snux.SnuxHardwarePlatform method*), 141
set_pulse_on_hit_and_enable_and_release_and_disable() (*mpf.platforms.spike.spike.SpikePlatform method*), 142

set_pulse_on_hit_and_enable_and_release_and_disable_rule() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 145
set_pulse_on_hit_and_enable_and_release_and_disable_rule() (*mpf.platforms.virtual_pinball.virtual_pinball.VirtualPinballPlatform method*), 146
set_pulse_on_hit_and_enable_and_release_rule() (*mpf.core.platform_controller.PlatformController method*), 44
set_pulse_on_hit_and_enable_and_release_rule() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 127
set_pulse_on_hit_and_enable_and_release_rule() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 129
set_pulse_on_hit_and_enable_and_release_rule() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 134
set_pulse_on_hit_and_enable_and_release_rule() (*mpf.platforms.rpi.rpi.RaspberryPiHardwarePlatform method*), 138
set_pulse_on_hit_and_enable_and_release_rule() (*mpf.platforms.snux.SnuxHardwarePlatform method*), 141
set_pulse_on_hit_and_enable_and_release_rule() (*mpf.platforms.spike.spike.SpikePlatform method*), 142
set_pulse_on_hit_and_enable_and_release_rule() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 145
set_pulse_on_hit_and_enable_and_release_rule() (*mpf.platforms.virtual_pinball.virtual_pinball.VirtualPinballPlatform method*), 146
set_pulse_on_hit_and_release_rule() (*mpf.core.platform_controller.PlatformController method*), 44
set_pulse_on_hit_and_release_rule() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 127
set_pulse_on_hit_and_release_rule() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 129
set_pulse_on_hit_and_release_rule() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 134
set_pulse_on_hit_and_release_rule() (*mpf.platforms.rpi.rpi.RaspberryPiHardwarePlatform method*), 138
set_pulse_on_hit_and_release_rule() (*mpf.platforms.snux.SnuxHardwarePlatform method*), 141
set_pulse_on_hit_and_release_rule() (*mpf.platforms.spike.spike.SpikePlatform method*), 143
set_pulse_on_hit_and_release_rule() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 145
set_pulse_on_hit_and_release_rule() (*mpf.platforms.virtual_pinball.virtual_pinball.VirtualPinballPlatform method*), 146
set_pulse_on_hit_rule() (*mpf.core.platform_controller.PlatformController method*), 45
set_pulse_on_hit_rule() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 127
set_pulse_on_hit_rule() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 129
set_pulse_on_hit_rule() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 134
set_pulse_on_hit_rule() (*mpf.platforms.rpi.rpi.RaspberryPiHardwarePlatform method*), 138
set_pulse_on_hit_rule() (*mpf.platforms.snux.SnuxHardwarePlatform method*), 141
set_pulse_on_hit_rule() (*mpf.platforms.spike.spike.SpikePlatform method*), 143
set_pulse_on_hit_rule() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 145
set_pulse_on_hit_rule() (*mpf.platforms.virtual_pinball.virtual_pinball.VirtualPinballPlatform method*), 146
set_setting_value() (*mpf.core.settings_controller.SettingsController method*), 46
set_speed_limit() (*mpf.devices.servo.Servo method*), 92
set_state() (*mpf.core.switch_controller.SwitchController method*), 50
set_tick_interval() (*mpf.devices.timer.Timer method*), 101
set_value() (*mpf.devices.score_reel_group.ScoreReelGroup method*), 88
set_volume() (*mpf.devices.hardware_sound_system.HardwareSoundSystem method*), 75
SettingsController (*class in mpf.core.settings_controller*), 46
setUp() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 163
setUp() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 173
setUp() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 183

setUp() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 192
setUp() (*mpf.tests.MpfTestCase.MpfTestCase method*), 200
setup_eject_chain() (*mpf.devices.ball_device.ball_device.BallDevice method*), 59
setup_eject_chain_next_hop() (*mpf.devices.ball_device.ball_device.BallDevice method*), 59
setup_player_controlled_eject() (*mpf.devices.ball_device.ball_device.BallDevice method*), 59
shortDescription() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 163
shortDescription() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 173
shortDescription() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 183
shortDescription() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 192
shortDescription() (*mpf.tests.MpfTestCase.MpfTestCase method*), 200
Shot (*class in mpf.devices.shot*), 95
ShotGroup (*class in mpf.devices.shot_group*), 92
ShotProfile (*class in mpf.devices.shot_profile*), 94
ShowController (*class in mpf.core.show_controller*), 46
ShowPlayer (*class in mpf.config_players.show_player*), 152
shutdown() (*mpf.core.machine.MachineController method*), 40
shutdown() (*mpf.tests.MpfTestCase.TestMachineController method*), 204
skipTest() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 163
skipTest() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 173
skipTest() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 183
skipTest() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 192
skipTest() (*mpf.tests.MpfTestCase.MpfTestCase method*), 200
slam_tilt() (*mpf.modes.tilt.code.tilt.Tilt method*), 123
SmartMatrixHardwarePlatform (*class in mpf.platforms.smartmatrix*), 139
SmartVirtualHardwarePlatform (*class in mpf.platforms.smart_virtual*), 139
Smbus2 (*class in mpf.platforms.smbus2*), 140
SnuxHardwarePlatform (*class in mpf.platforms.snux*), 140
SpikePlatform (*class in mpf.platforms.spike.spike*), 141
stack (*mpf.devices.light.Light attribute*), 79
start() (*mpf.core.ball_search.BallSearch method*), 206
start() (*mpf.core.mode.Mode method*), 210
start() (*mpf.devices.achievement.Achievement method*), 56
start() (*mpf.devices.multiball.Multiball method*), 83
start() (*mpf.devices.timer.Timer method*), 101
start() (*mpf.modes.attract.code.attract.Attract method*), 103
start() (*mpf.modes.bonus.code.bonus.Bonus method*), 106
start() (*mpf.modes.carousel.code.carousel.Carousel method*), 108
start() (*mpf.modes.credits.code.credits.Credits method*), 111
start() (*mpf.modes.game.code.game.Game method*), 114
start() (*mpf.modes.high_score.code.high_score.HighScore method*), 116
start() (*mpf.modes.match.code.match.Match method*), 118
start() (*mpf.modes.service.code.service.Service method*), 121
start() (*mpf.modes.tilt.code.tilt.Tilt method*), 123
start() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 127
start() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 129
start() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 134
start() (*mpf.platforms.smart_virtual.SmartVirtualHardwarePlatform method*), 139
start_button_pressed() (*mpf.modes.attract.code.attract.Attract method*), 104
start_button_released() (*mpf.modes.attract.code.attract.Attract method*), 104
start_game() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 173
start_game() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 183
start_mode() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 163
start_mode() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 173
start_mode() (*mpf.tests.MpfGameTestCase.MpfGameTestCase*

method), 183
start_mode() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 192
start_mode() (*mpf.tests.MpfTestCase.MpfTestCase method*), 200
start_or_add_a_ball() (*mpf.devices.multiball.Multiball method*), 83
start_selected() (*mpf.devices.achievement_group.AchievementGroup method*), 55
start_service() (*mpf.core.service_controller.ServiceController method*), 46
start_two_player_game() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 173
start_two_player_game() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 183
started (*mpf.core.ball_search.BallSearch attribute*), 206
state (*mpf.devices.achievement.Achievement attribute*), 56
state (*mpf.devices.ball_device.ball_device.BallDevice attribute*), 59
state (*mpf.devices.combo_switch.ComboSwitch attribute*), 65
state (*mpf.devices.shot.Shot attribute*), 96
state (*mpf.devices.state_machine.StateMachine attribute*), 96
state (*mpf.devices.switch.Switch attribute*), 98
state_name (*mpf.devices.shot.Shot attribute*), 96
StateMachine (*class in mpf.devices.state_machine*), 96
Stepper (*class in mpf.devices.stepper*), 97
stop() (*mpf.core.ball_search.BallSearch method*), 206
stop() (*mpf.core.machine.MachineController method*), 40
stop() (*mpf.core.mode.Mode method*), 210
stop() (*mpf.core.text_ui.TextUi method*), 52
stop() (*mpf.devices.achievement.Achievement method*), 56
stop() (*mpf.devices.multiball.Multiball method*), 83
stop() (*mpf.devices.score_reel.ScoreReel method*), 89
stop() (*mpf.devices.stepper.Stepper method*), 97
stop() (*mpf.devices.timer.Timer method*), 101
stop() (*mpf.modes.attract.code.attract.Attract method*), 104
stop() (*mpf.modes.bonus.code.bonus.Bonus method*), 106
stop() (*mpf.modes.carousel.code.carousel.Carousel method*), 108
stop() (*mpf.modes.credits.code.credits.Credits method*), 111
stop() (*mpf.modes.game.code.game.Game method*), 114
stop() (*mpf.modes.high_score.code.high_score.HighScore method*), 116
stop() (*mpf.modes.match.code.match.Match method*), 119
stop() (*mpf.modes.service.code.service.Service method*), 121
stop() (*mpf.modes.tilt.code.tilt.Tilt method*), 123
stop() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 127
stop() (*mpf.platforms.i2c_servo_controller.I2CServoControllerHardware method*), 128
stop() (*mpf.platforms.lisy.lisy.LisyHardwarePlatform method*), 129
stop() (*mpf.platforms.mma8451.MMA8451Platform method*), 130
stop() (*mpf.platforms.mypinballs.mypinballs.MyPinballsHardwarePlatfo method*), 131
stop() (*mpf.platforms.openpixel.OpenpixelHardwarePlatform method*), 131
stop() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 134
stop() (*mpf.platforms.pololu_maestro.PololuMaestroHardwarePlatform method*), 137
stop() (*mpf.platforms.rpi.rpi.RaspberryPiHardwarePlatform method*), 138
stop() (*mpf.platforms.smartmatrix.SmartMatrixHardwarePlatform method*), 139
stop() (*mpf.platforms.snux.SnuxHardwarePlatform method*), 141
stop() (*mpf.platforms.spike.spike.SpikePlatform method*), 143
stop() (*mpf.platforms.trinamics_steprocker.TrinamicsStepRocker method*), 143
stop() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 145
stop() (*mpf.tests.MpfBcpTestCase.MockBcpClient method*), 155
stop() (*mpf.tests.MpfTestCase.TestMachineController method*), 204
stop_all_sounds() (*mpf.devices.hardware_sound_system.HardwareSoundSystem method*), 75
stop_device() (*mpf.devices.ball_device.ball_device.BallDevice method*), 60
stop_devices() (*mpf.core.device_manager.DeviceManager method*), 33
stop_game() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 173
stop_game() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 183
stop_ignoring_hits() (*mpf.devices.logic_blocks.Counter method*), 66
stop_mode() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase*

method), 163
stop_mode() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 173
stop_mode() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 184
stop_mode() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 192
stop_mode() (*mpf.tests.MpfTestCase.MpfTestCase method*), 200
stop_service() (*mpf.core.service_controller.ServiceController method*), 46
string_to_class() (*mpf.core.utility_functions.Util static method*), 219
string_to_gain() (*mpf.core.utility_functions.Util static method*), 219
string_to_list() (*mpf.core.utility_functions.Util static method*), 219
string_to_lowercase_list() (*mpf.core.utility_functions.Util static method*), 220
string_to_ms() (*mpf.core.utility_functions.Util static method*), 220
string_to_rgb() (*mpf.core.rgb_color.RGBColor static method*), 215
string_to_rgb() (*mpf.core.rgba_color.RGBAColor static method*), 214
string_to_secs() (*mpf.core.utility_functions.Util static method*), 220
subtract() (*mpf.devices.timer.Timer method*), 101
sw_flip() (*mpf.devices.flipper.Flipper method*), 74
sw_release() (*mpf.devices.flipper.Flipper method*), 74
Switch (*class in mpf.devices.switch*), 97
SwitchController (*class in mpf.core.switch_controller*), 47
SwitchPlayer (*class in mpf.plugins.switch_player*), 51

T

tearDown() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase method*), 163
tearDown() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase method*), 173
tearDown() (*mpf.tests.MpfGameTestCase.MpfGameTestCase method*), 184
tearDown() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase method*), 192
tearDown() (*mpf.tests.MpfTestCase.MpfTestCase method*), 200
TestDataManager (*class in mpf.tests.TestDataManager*), 201
TestMachineController (*class in mpf.tests.MpfTestCase*), 201
text_to_speech() (*mpf.devices.hardware_sound_system.HardwareSoundSystem method*), 75
TextUi (*class in mpf.core.text_ui*), 52
tick() (*mpf.platforms.p3_roc.P3RocHardwarePlatform method*), 135
tick() (*mpf.platforms.p_roc.PRocHardwarePlatform method*), 137
tick() (*mpf.platforms.snux.SnuxHardwarePlatform method*), 141
tick() (*mpf.devices.timer.Timer attribute*), 101
Tilt (*class in mpf.modes.tilt.code.tilt*), 121
tilt() (*mpf.modes.tilt.code.tilt.Tilt method*), 124
tilt_settle_ms_remaining() (*mpf.modes.tilt.code.tilt.Tilt method*), 124
tilt_warning() (*mpf.modes.tilt.code.tilt.Tilt method*), 124
TimedSwitch (*class in mpf.devices.timed_switch*), 99
Timer (*class in mpf.devices.timer*), 99
timer_complete() (*mpf.devices.timer.Timer method*), 101
timer_start() (*mpf.devices.ball_save.BallSave method*), 63
toggle_credit_play() (*mpf.modes.credits.code.credits.Credits method*), 111
transfer() (*mpf.devices.playfield_transfer.PlayfieldTransfer method*), 83
TrinamicsStepRocker (*class in mpf.platforms.trinamics_steprocker*), 143

U

unblock() (*mpf.core.ball_search.BallSearch method*), 206
unexpected_ball_received() (*mpf.devices.ball_device.ball_device.BallDevice method*), 60
unexpected_ball_received() (*mpf.devices.playfield.Playfield method*), 86
unittest_verbosity() (*mpf.tests.MpfBcpTestCase.MpfBcpTestCase static method*), 163
unittest_verbosity() (*mpf.tests.MpfFakeGameTestCase.MpfFakeGameTestCase static method*), 173
unittest_verbosity() (*mpf.tests.MpfGameTestCase.MpfGameTestCase static method*), 184
unittest_verbosity() (*mpf.tests.MpfMachineTestCase.MpfMachineTestCase static method*), 192
unittest_verbosity() (*mpf.tests.MpfTestCase.MpfTestCase static method*), 200

unlight() (*mpf.devices.score_reel_group.ScoreReelGroup method*), 88
update() (*mpf.devices.dmd.Dmd method*), 68
update() (*mpf.devices.rgb_dmd.RgbDmd method*), 87
update_acceleration() (*mpf.devices.accelerometer.Accelerometer method*), 53
update_firmware() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 127
update_incand() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 134
update_leds() (*mpf.platforms.fast.fast.FastHardwarePlatform method*), 127
update_switches_from_hw() (*mpf.core.switch_controller.SwitchController method*), 51
Util (*class in mpf.core.utility_functions*), 216

V

validate_coil_section() (*mpf.platforms.snux.SnuxHardwarePlatform method*), 141
validate_coil_section() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 145
validate_config_entry() (*mpf.config_players.block_event_player.BlockEventPlayer method*), 147
validate_config_entry() (*mpf.config_players.queue_event_player.QueueEventPlayer method*), 150
validate_config_entry() (*mpf.config_players.queue_relay_player.QueueRelayPlayer method*), 151
validate_config_entry() (*mpf.config_players.random_event_player.RandomEventPlayer method*), 151
validate_config_entry() (*mpf.config_players.variable_player.VariablePlayer method*), 153
validate_machine_config_section() (*mpf.core.machine.MachineController method*), 40
validate_machine_config_section() (*mpf.tests.MpfTestCase.TestMachineController method*), 204
validate_stepper_section() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 145
validate_switch_section() (*mpf.platforms.virtual.VirtualHardwarePlatform method*), 145
value (*mpf.devices.logic_blocks.Accrual attribute*), 54
value (*mpf.devices.logic_blocks.Counter attribute*), 66
value (*mpf.devices.logic_blocks.Sequence attribute*), 92
VariablePlayer (*class in mpf.config_players.variable_player*), 153
verify_switches() (*mpf.core.switch_controller.SwitchController method*), 51
verify_system_info() (*mpf.core.machine.MachineController method*), 40
verify_system_info() (*mpf.tests.MpfTestCase.TestMachineController method*), 204
vers_resp() (*mpf.platforms.opp.opp.OppHardwarePlatform method*), 134
VirtualHardwarePlatform (*class in mpf.platforms.virtual*), 144
VirtualPinballPlatform (*class in mpf.platforms.virtual_pinball.virtual_pinball*), 145
vpx_changed_gi_strings() (*mpf.platforms.virtual_pinball.virtual_pinball.VirtualPinballPlatf method*), 146
vpx_changed_lamps() (*mpf.platforms.virtual_pinball.virtual_pinball.VirtualPinballPlatf method*), 146
vpx_changed_solenoids() (*mpf.platforms.virtual_pinball.virtual_pinball.VirtualPinballPlatf method*), 146
vpx_get_mech() (*mpf.platforms.virtual_pinball.virtual_pinball.VirtualP method*), 146
vpx_get_switch() (*mpf.platforms.virtual_pinball.virtual_pinball.Virtu method*), 146
vpx_mech() (*mpf.platforms.virtual_pinball.virtual_pinball.VirtualPinbal method*), 146
vpx_set_mech() (*mpf.platforms.virtual_pinball.virtual_pinball.VirtualP method*), 146
vpx_set_switch() (*mpf.platforms.virtual_pinball.virtual_pinball.Virtu method*), 147
vpx_start() (*mpf.platforms.virtual_pinball.virtual_pinball.VirtualPinba method*), 147
vpx_switch() (*mpf.platforms.virtual_pinball.virtual_pinball.VirtualPin method*), 147

W

wait_for_any_event() (*mpf.core.events.EventManager method*), 37
wait_for_any_switch() (*mpf.core.switch_controller.SwitchController method*), 51
wait_for_asset_load() (*mpf.core.assets.AsyncioSyncAssetManager method*), 29

wait_for_event() (*mpf.core.events.EventManager method*), 37
wait_for_ready() (*mpf.devices.score_reel.ScoreReel method*), 89
wait_for_ready() (*mpf.devices.score_reel_group.ScoreReelGroup method*), 88
wait_for_ready_to_receive() (*mpf.devices.ball_device.ball_device.BallDevice method*), 60
wait_for_ready_to_receive() (*mpf.devices.playfield.Playfield static method*), 86
wait_for_switch() (*mpf.core.switch_controller.SwitchController method*), 51
warning_log() (*mpf.core.ball_search.BallSearch method*), 206
warning_log() (*mpf.core.data_manager.DataManager method*), 221
warning_log() (*mpf.core.delays.DelayManager method*), 223
warning_log() (*mpf.core.logging.LogMixin method*), 208
warning_log() (*mpf.core.mode.Mode method*), 211
warning_log() (*mpf.modes.attract.code.attract.Attract method*), 104
warning_log() (*mpf.modes.bonus.code.bonus.Bonus method*), 106
warning_log() (*mpf.modes.carousel.code.carousel.Carousel method*), 109
warning_log() (*mpf.modes.credits.code.credits.Credits method*), 111
warning_log() (*mpf.modes.game.code.game.Game method*), 114
warning_log() (*mpf.modes.high_score.code.high_score.HighScore method*), 117
warning_log() (*mpf.modes.match.code.match.Match method*), 119
warning_log() (*mpf.modes.service.code.service.Service method*), 121
warning_log() (*mpf.modes.tilt.code.tilt.Tilt method*), 124
warning_log() (*mpf.tests.MpfBcpTestCase.MockBcpClient method*), 155
warning_log() (*mpf.tests.MpfTestCase.TestMachineController method*), 204
warning_log() (*mpf.tests.TestDataManager.TestDataManager method*), 201

X

x (*mpf.devices.light.Light attribute*), 80
x (*mpf.devices.switch.Switch attribute*), 98

Y

y (*mpf.devices.light.Light attribute*), 80
y (*mpf.devices.switch.Switch attribute*), 98

Z

z (*mpf.devices.light.Light attribute*), 80
z (*mpf.devices.switch.Switch attribute*), 98