# MPF Documentation Developer Documentation

*Release 0.33.49*

The Mission Pinball Framework Team

Dec 23, 2017

# DEVELOPER DOCUMENTATION

This is the developer documentation for the Mission Pinball Framework (MPF), version 0.33. **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

# CHAPTER 3

# API Reference

# CHAPTER 4

# Writing Tests

# CHAPTER 5

# Extending, Adding to, and Enhancing MPF

# CHAPTER 6

# BCP Protocol

# CHAPTER 7

# Index

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

## 7.1 Overview & Tour of MPF code

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

### 7.1.1 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.

## 7.1.2 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.

### 7.1.3 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 scriptlets and custom machine code
- Stepping through the initialization and reset phases

### 7.1.4 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 in a few ways:

**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 MPF 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.

## 7.2 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*.

### 7.2.1 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.

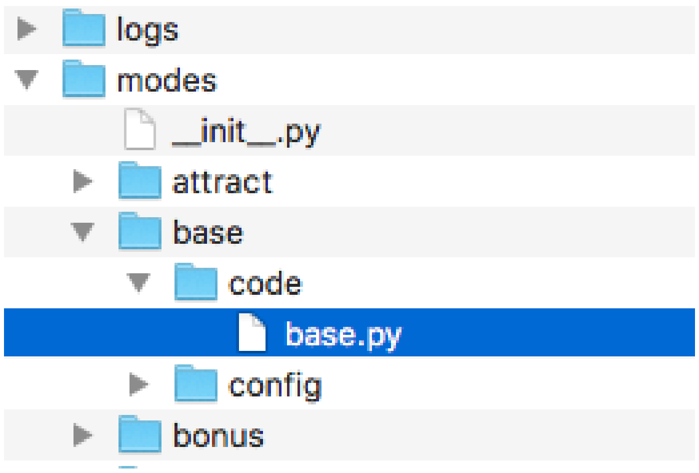
This "mode code" (as we call it) has access to the full MPF API. You can post events, register event handlers which run custom things when events are posted, access device state and control devices, read and set player variables, post slides... really anything MPF can do, you can do.

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.

```python
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?
        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.

### 7.2.2 How to add machine-wide custom code

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

Scriptlets 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.

Scriptlets, on the other hand, are sort of "machine-level" custom code. Scriptlets 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.

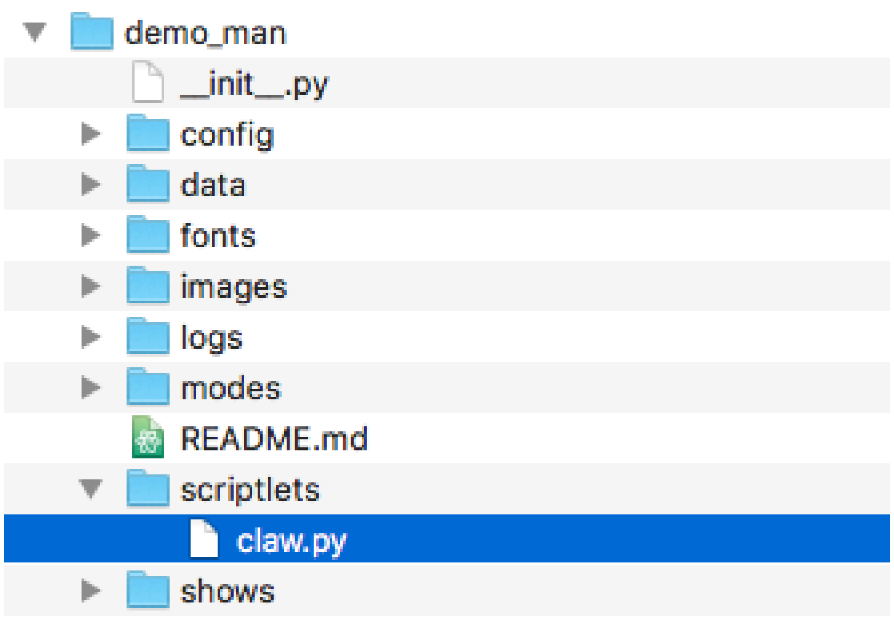
(You can read about how to download and run *Demo Man* in the example games section section of the MPF User Documentation.)

Here's how to create a scriptlet:

#### 1. Create your scriptlet file

First, add a `scriptlets` folder to your machine folder. Then inside there, create the Python file that will hold your scriptlet. 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:



### 2. Open and edit your scriptlet file

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

```
from mpf.core.scriptlet import Scriptlet


class Claw(Scriptlet):
    pass
```

Note that MPF contains a `Scriptlet` 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 scriptlet. There's also an `on_load()` method which is called when the scriptlet is loaded which you can use in your own code.

### 3. Add the scriptlet to your machine config

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

For *Demo Man*, that looks like this:

```
scriptlets:
  - claw.Claw
```

### 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 scriptlet.

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

```python
"""Claw controller Scriptlet for Demo Man"""

from mpf.core.scriptlet import Scriptlet


class Claw(Scriptlet):

    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)

        # 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

        # 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)
        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()
```

## 7.3 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.

### 7.3.1 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*)

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*)

Bases: `object`

Base class for the auditor.

**Parameters** **`machine`** – A refence to the machine controller object.

##### 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*)

Called to log an auditable event.

**Parameters**

- **`audit_class`** – A string of the section we want this event to be

- **`to.`** (*`logged`*) –
- **`event`** – A string name of the event we're auditing.
- **`**kawargs`** – Not used, but included since some of the audit events might include random kwargs.

**`audit_event`**(*eventname*, ***kwargs*)
Registered as an event handlers to log an event to the audit log.

**Parameters**

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

**`audit_player`**(***kwargs*)
Called to write player data to the audit log.

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

**Parameters** **`not used, but included since some types of events include`** (*`**kwargs,`*) – 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 = None`**
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*)
Bases: `mpf.core.mpf_controller.MpfController`

Base class for the Ball Controller which is used to keep track of all the balls in a pinball machine.

**Parameters** **`machine`** (`MachineController`) – A reference to the instance of the MachineController object.

### 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*)
Inform ball controller about a capured ball (which might be new).

**`are_balls_collected`**(*target*)
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'*)
Used to 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`**()
Dump ball count of all devices.

**`request_to_start_game`**(***kwargs*)
Method registered for 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*)
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.config_processor

**class** `mpf.core.config_processor.`**`ConfigProcessor`**(*machine*)

Bases: `object`

Config processor which loads the config.

#### Accessing the config_processor in code

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

#### Methods & Attributes

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

**static load_config_file**(*filename*, *config_type*, *verify_version=True*, *halt_on_error=True*, *ignore_unknown_sections=False*)

Load a config file.

### self.machine.device_manager

**class** `mpf.core.device_manager.`**`DeviceManager`**(*machine*)

Bases: `mpf.core.mpf_controller.MpfController`

Manages devices in a MPF machine.

#### 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_collection_control_events**(***kwargs*)

Create control events for collection.

**create_devices**(*collection_name*, *config*)

Create devices for 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.

**register_monitorable_device**(*device*)
Register a monitorable device.

## self.machine.events

**class** `mpf.core.events.`**EventManager**(*machine*)
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_handler**(*event*, *handler*, *priority=1*, ***kwargs*)
Register an event handler to respond to an event.

If you add a handlers for an event for which it has already been registered, the new one will overwrite the old one. This is useful for changing priorities of existing handlers. Also it's good to know that you can safely add a handler over and over.

**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. Note that all event strings will be converted to lowercase.
- **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*)
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. This string will be converted to lowercase.

**Returns** True or False

**`get_event_and_condition_from_string`**(*event_string*)
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, cleaned up a by converting it to lowercase.

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

**Return type** 2-item tuple

**`post`**(*event*, *callback=None*, ***kwargs*)
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. Note that this event name will be converted to lowercase.
- **`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. (Just make sure your handlers are expecting them. You can add **kwargs to your handler methods if certain ones don't need them.)

**post_async**(*event*, ***kwargs*)
Post event and wait until all handlers are done.

**post_boolean**(*event*, *callback=None*, ***kwargs*)
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. Note that this event name will be converted to lowercase.
- **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. Note that this event name will be converted to lowercase.
- **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.)

**post_queue_async**(*event*, ***kwargs*)
Post queue event, wait until all handlers are done and locks are released.

**post_relay**(*event*, *callback=None*, ***kwargs*)
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. Note that this event name will be converted to lowercase.
- **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*, ***kwargs*)
Post relay event, wait until all handlers are done and return result.

**process_event_queue**()
Check if there are any other events that need to be processed, and then process them.

**remove_handler**(*method*)
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*, *handler*)
Remove the handler you pass from the event you pass.

**Parameters**

- **event** – The name of the event you want to remove the handler from. This string will be converted to lowercase.
- **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*)
Remove a registered event handler by key.

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

**remove_handlers_by_keys**(*key_list*)
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*, *handler*, *priority=1*, ***kwargs*)
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. This string will be converted to lowercase.
- **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 kwatgs 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: [<class 'str'>]*)
Wait for any event from event_names.

**wait_for_event**(*event_name: str*)
Wait for event.

## self.machine.extra_ball_controller

**class** `mpf.core.extra_balls.`**ExtraBallController**(*machine*)
Bases: `mpf.core.mpf_controller.MpfController`

Tracks and manages extra balls at the global level.

### Accessing the extra_ball_controller in code

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

### Methods & Attributes

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

**award**(***kwargs*)
Immediately awards an extra ball.

This event first checks to make sure the limits of the max extra balls have not been exceeded.

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

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

**light**(***kwargs*)
Lights the extra ball.

This method also increments the player's extra_balls_lit count.

**relight**(***kwargs*)
Relights the extra ball when a player's turn starts.

This event does not post the "extra_ball_lit_awarded" event so you can use it to not show the extra ball awards when a player starts their turn with an extra ball lit.

## self.machine.info_lights

**class** mpf.plugins.info_lights.**InfoLights**(*machine*)

Bases: `object`

Plugin which uses lights to represent game state.

### 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.logic_blocks

**class** mpf.core.logic_blocks.**LogicBlocks**(*machine: mpf.core.machine.MachineController*)

Bases: `mpf.core.mpf_controller.MpfController`

LogicBlock Manager.

### Accessing the logic_blocks in code

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

### Methods & Attributes

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

**player_turn_start**(*player*, ***kwargs*)

Create blocks for current player.

**Parameters** **player** – Player object

**player_turn_stop**(*player: mpf.core.player.Player*, ***kwargs*)

Remove blocks from current player.

**Parameters** **player** – Player pnkect

## 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`** – Dictionary of options the machine controller uses to configure itself.

**`options`**
*dict* – A dictionary of options built from the command line options used to launch mpf.py.

**`config`**
*dict* – A dictionary of machine's configuration settings, merged from various sources.

**`game`**
*mpf.modes.game.code.game.Game* – the current game

**`machine_path`**
The root path of this machine_files folder

**`plugins`**

**`scriptlets`**

**`hardware_platforms`**

**`events`**
*mpf.core.events.EventManager*

### 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*)
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*)
Clear a boot hold.

**`create_data_manager`**(*config_name*)
Return a new DataManager for a certain config.

**Parameters** **`config_name`** – Name of the config

**`create_machine_var`**(*name*, *value=0*, *persist=False*, *expire_secs=None*, *silent=False*)
Create a new machine variable.

**Parameters**

- **`name`** – String name of the variable.
- **`value`** – The value of the variable. This can be any Type.
- **`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 be loaded with a value of 0. 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.

**get_machine_var**(*name*)
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*, *overwrite*)
Return platform section.

**init_done**()
Finish init.

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

**is_machine_var**(*name*)
Return true if machine variable exists.

**power_off**(***kwargs*)
Attempt to perform a power down of the pinball machine and ends MPF.

This method is not yet implemented.

**register_boot_hold**(*hold*)
Register a boot hold.

**register_monitor**(*monitor_class*, *monitor*)
Register a monitor.

**Parameters**

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

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*)
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=''*, *endswith=''*)
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**()
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**()
Start the main machine run loop.

**set_default_platform**(*name*)
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*, *value*, *force_events=False*)
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.
- **force_events** – Boolean which will force the event posting, the machine monitor callback, and writing the variable to disk (if it's set to persist). By default these things only happen if the new value is different from the old value.

**stop**(***kwargs*)
Perform a graceful exit of MPF.

**validate_machine_config_section**(*section*)
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*)
Bases: `mpf.core.mpf_controller.MpfController`

Parent class for the Mode Controller.

There is one instance of this in MPF and it's responsible for loading, unloading, and managing all modes.

**Parameters** **machine** – The main MachineController instance.

### 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.

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

**`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.

**`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 when the 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.

Note that these methods will be called every single time this mode is started.

**`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, active*)
Called when a mode goes active or inactive.

## 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.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**() → [<class 'mpf.core.service_controller.CoilMap'>]
Return a map of all coils 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.

**_settings**
*dict[str, SettingEntry]* – Available 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_value**(*setting_name*)
Return the current value of a setting.

**get_setting_value_label**(*setting_name*)
Return label for value.

**get_settings**() → {<class 'mpf.core.settings_controller.SettingEntry'>, <class 'str'>}
Return all available settings.

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

### self.machine.shot_profile_manager

**class** `mpf.core.shot_profile_manager.`**ShotProfileManager**(*machine*)
Bases: `object`

Controller for show profiles.

#### Accessing the shot_profile_manager in code

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

### Methods & Attributes

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

**`mode_start_for_shot_groups`** (*config*, *priority*, *mode*, ***kwargs*)
Apply profiles to member shots of a dict of shot groups.

**Parameters**

- **`config`** – Dict containing shot groups. Keys are shot group names. Values are settings for each shot group.
- **`priority`** – Int of the priority these profiles will be applied at. unused.
- **`mode`** – A Mode class object for the mode which is applying these profiles. Used as the key to remove the profiles a specific mode applied later.
- **`kwargs`** – unused

**`mode_start_for_shots`** (*config*, *mode*, ***kwargs*)
Set the shots' enable_tables.

Called on mode start.

**`mode_stop_for_shot_groups`** (*mode*)
Remove all the profiles that were applied to shots based on shot group settings in a mode.

**Parameters** **`mode`** – A Mode class which represents the mode that applied the profiles originally which will be used to determine which shot profiles should be removed.

**`mode_stop_for_shots`** (*mode*)
Remove shot profile from mode.

**`process_profile_config`** (*profile_name*, *config*)
Process a shot profile config to convert everything to the format the shot controller needs.

**Parameters** **`config`** – Dict of the profile settings to process.

**`register_profile`** (*name*, *profile*)
Register a new shot profile with the shot controller which will allow it to be applied to shots.

**Parameters**

- **`name`** – String name of the profile you're registering.
- **`profile`** – Dict of the profile settings.

**`register_profiles`** (*config*, ***kwargs*)
Register multiple shot profiles.

**Parameters**

- **`config`** – Dict containing the profiles you're registering. Keys are profile names, values are dictionaries of profile settings.
- **`kwargs`** – unused

## 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.

'hardware shows' are coordinated light, flasher, coil, and event effects. The ShowController handles priorities, restores, running and stopping Shows, etc. There should be only one per machine.

**Parameters** **`machine`** – Parent machine object.

#### 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.

**`get_next_show_id`**()
Return the next show id.

**`get_running_shows`**(*name*)
Return a list of running shows by show name or instance name.

**Parameters** **`name`** – String name of the running shows you want to get. This can be a show name (which will return all running instances of that show) or a key (which will also return all running show instances that have that instance name).

**Returns** A list of RunningShow() objects.

**`notify_show_starting`**(*show*)
Register a running show.

**`notify_show_stopping`**(*show*)
Remove a running show.

**`register_show`**(*name*, *settings*)
Register a named show.

### self.machine.switch_controller

**class** `mpf.core.switch_controller.`**`SwitchController`**(*machine*)
Bases: `mpf.core.mpf_controller.MpfController`

Handles all switches in the machine.

Base class for the switch controller, which is responsible for receiving all switch activity in the machine and converting them into events.

More info: http://docs.missionpinball.org/en/latest/core/switch_controller.html

#### 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*)
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 (rounded up to the nearst machine timer tick).
- **`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.

**`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.

Returns True if the current switch is active. If optional arg ms is passed, will only return true if switch has been active for that many ms.

Note this method does consider whether a switch is NO or NC. So an NC switch will show as active if it is open, rather than closed.

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

Returns True if the current switch is inactive. If optional arg *ms* is passed, will only return true if switch has been inactive for that many ms.

Note this method does consider whether a switch is NO or NC. So an NC switch will show as active if it is closed, rather than open.

**`is_state`**(*switch_name*, *state*, *ms=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.

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 log file of all switches that are currently active.

This is used to set the "initial" switch states of standalone testing tools, like our log file playback utility, but it might be useful in other scenarios when weird things are happening.

This method dumps these events with logging level "INFO."

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

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

**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 paramenter 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'.)

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

**`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 paramenter 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`**(*name*)
Populate self.registered_switches.

**Parameters** **`name`** – Name of switch

**`remove_monitor`**(*monitor*)
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 hander by key returned from add_switch_handler.

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

**`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.

Loop through all the switches and queries their hardware states via their platform interfaces and them 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: [<class 'str'>], state: int = 1, only_on_change=True, ms=0*)
Wait for the first switch in the list to change into state.

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

### self.machine.switch_player

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

Plays switches from config.

#### 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.

## 7.3.2 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`**(*self_inner*, **args*, ***kwargs*)

Bases: `mpf.core.system_wide_device.SystemWideDevice`

Implement an accelerometer.

Args: Same as the Device parent class

#### 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`**()
Return current 3D level.

**`get_level_xz`**()
Return current 2D x/z level.

**`get_level_yz`**()
Return current 2D y/z level.

**update_acceleration** (*x, y, z*)
Calculate acceleration based on readings from hardware.

### self.machine.achievement_groups.*

**class** mpf.devices.achievement_group.**AchievementGroup** (*self_inner, *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.

**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** (*self_inner, *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.

**`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`**(*self_inner*, **args*, ***kwargs*)
Bases: `mpf.core.system_wide_device.SystemWideDevice`

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

autofire_coils are used when you want the coils to respond "instantly" without waiting for the lag of the python game code running on the host computer.

Examples of autofire_coils are pop bumpers, slingshots, and flippers.

Args: Same as Device.

#### 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 coil rule.

**`enable`**(***kwargs*)
Enable the autofire coil rule.

### self.machine.ball_devices.*

**class** `mpf.devices.ball_device.ball_device.`**`BallDevice`**(*self_inner*, **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 = None**
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*)
Eject ball to target.

**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.

**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.

**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

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

**`setup_eject_chain`**(*path*, *player_controlled=False*)
Setup an eject chain.

**`setup_eject_chain_next_hop`**(*path*, *player_controlled*)
Setup one hop of the eject chain.

**`setup_player_controlled_eject`**(*target=None*)
Setup a player controlled eject.

**`state`**
Return the device state.

**`stop`**(***kwargs*)
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`**(*self_inner*, **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.

**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**(*self_inner*, **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.

**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**(*self_inner*, **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.

**`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.

**`clear_hw_rule`**(*switch: mpf.devices.switch.Switch*)
Clear all rules for switch and this driver.

**Parameters** **`switch`** – Switch to clear on this driver.

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

**enable**(***kwargs*)
Enable a driver by holding it 'on'.

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_configured_driver**()
Return a configured hw driver.

**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.)

**set_pulse_on_hit_and_enable_and_release_and_disable_rule**(*enable_switch: mpf.devices.switch.Switch*, *disable_switch: mpf.devices.switch.Switch*)
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** – Switch which triggers the rule.
- **disable_switch** – Switch which disables the rule.

**set_pulse_on_hit_and_enable_and_release_rule**(*enable_switch: mpf.devices.switch.Switch*)
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.

**set_pulse_on_hit_and_release_rule**(*enable_switch: mpf.devices.switch.Switch*)
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.

**set_pulse_on_hit_rule**(*enable_switch: mpf.devices.switch.Switch*)
Add pulse on hit rule to driver.

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

**Parameters** **enable_switch** – Switch which triggers the rule.

### self.machine.combo_switches.*

**class** `mpf.devices.combo_switch.`**`ComboSwitch`**(*self_inner*, **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.

**`state`**

Return current state.

### self.machine.diverters.*

**class** `mpf.devices.diverter.`**`Diverter`**(*self_inner*, **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.

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

**schedule_deactivation**()
Schedule a delay to deactivate this diverter.

### self.machine.drop_target_banks.*

**class** `mpf.devices.drop_target.`**DropTargetBank**(*self_inner*, **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.

**`member_target_change`**`()`

A member drop target has changed state.

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

**`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`**`(`*self_inner*, **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.

**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.)

### self.machine.extra_balls.*

**class** `mpf.devices.extra_ball.`**`ExtraBall`**`(`*machine*, *name*`)`

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.

**`reset`**`(`***kwargs*`)`

Reset extra ball.

Does not reset the additional ball the player received. Only resets the device and allows to award another extra ball to the player.

### self.machine.flashers.*

**class** `mpf.devices.flasher.`**`Flasher`**`(`*machine*, *name*`)`

Bases: `mpf.core.system_wide_device.SystemWideDevice`

Generic class that holds flasher objects.

#### Accessing flashers in code

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

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

#### Methods & Attributes

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

**`flash`**`(`*milliseconds=None*, ***kwargs*`)`

Flashe the flasher.

**Parameters** **`milliseconds`** – Int of how long you want the flash to be, in ms. Default is None which causes the flasher to flash for whatever its default config is, either its own flash_ms or the core- wide default_flash_ms settings. (Current default is 50ms.)

**`get_configured_driver`**`()`
Reconfigure driver.

### self.machine.flippers.*

**class** `mpf.devices.flipper.`**`Flipper`**`(`*self_inner*, **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

**sw_flip**(*include_switch=False*)
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**(*include_switch=False*)
Deactive the flipper via software as if the flipper button was released.

See the documentation for sw_flip() for details.

## self.machine.gis.*

**class** mpf.devices.gi.**Gi**(*self_inner*, **args*, ***kwargs*)
Bases: `mpf.core.system_wide_device.SystemWideDevice`

Represents a light connected to a traditional lamp matrix in a pinball machine.

This light could be an incandescent lamp or a replacement single-color LED. The key is that they're connected up to a lamp matrix.

### Accessing gis in code

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

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

### Methods & Attributes

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

**add_handler**(*callback*)
Register a handler to be called when this GI changes state.

**disable**(***kwargs*)
Disable this GI string.

**enable**(*brightness=255*, ***kwargs*)
Enable this GI string.

**Parameters**

- **brightness** – Int from 0-255 of how bright you want this to be. 255 is on. 0 os iff. Note that not all GI strings on all machines support this.
- **fade_ms** – How quickly you'd like this GI string to fade to this brightness level. This is not implemented.

**remove_handler**(*callback=None*)
Remove a handler from the list of registered handlers.

## self.machine.kickbacks.*

**class** mpf.devices.kickback.**Kickback**(*self_inner*, **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*)
Remove switch handler and call parent.

**enable**(***kwargs*)
Add switch handler and call parent.

## self.machine.led_rings.*

**class** mpf.devices.led_group.**LedRing**(*machine: mpf.core.machine.MachineController*, *name*)
Bases: `mpf.devices.led_group.LedGroup`

A LED ring.

### Accessing led_rings in code

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

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

#### Methods & Attributes

Led_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*, *mode=None*)
Call color on all leds in this group.

**`get_token`**()
Return all LEDs in group as token.

### self.machine.led_stripes.*

**`class`** `mpf.devices.led_group.`**`LedStrip`**(*machine: mpf.core.machine.MachineController*, *name*)
Bases: `mpf.devices.led_group.LedGroup`

A LED stripe.

#### Accessing led_stripes in code

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

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

#### Methods & Attributes

Led_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*, *mode=None*)
Call color on all leds in this group.

**`get_token`**()
Return all LEDs in group as token.

### self.machine.leds.*

**`class`** `mpf.devices.led.`**`Led`**(*self_inner*, **args*, ***kwargs*)
Bases: `mpf.core.system_wide_device.SystemWideDevice`

An RGB LED in a pinball machine.

#### Accessing leds in code

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

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

### Methods & Attributes

Leds 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 LED to 'off'.

**`color`**(*color*, *fade_ms=None*, *priority=0*, *key=None*, *mode=None*)
Add or update a color entry in this LED's stack, which is how you tell this LED 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 LED 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 LED's and/or the machine's default fade_ms setting.
- **`priority`** – Int value of the priority of these incoming settings. If this LED 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.
- **`mode`** – Optional mode instance of the mode that is setting this color. When a mode ends, entries from the stack with that mode will automatically be removed.

**`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_task`**(*dt*)
Perform a fade depending on the current time.

**Parameters** **`dt`** – time since last call

**`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 LED, but not always (since physical LEDs 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.

**classmethod mode_stop**(*mode: mpf.core.mode.Mode*)
Remove all entries from mode.

**Parameters** **mode** – Mode which was removed

**off**(*fade_ms=None*, *priority=0*, *key=None*, ***kwargs*)
Turn LED off.

**Parameters**

- **key** – key for removal later on
- **priority** – priority on stack
- **fade_ms** – duration of fade

**on**(*fade_ms=None*, *priority=0*, *key=None*, ***kwargs*)
Turn LED on.

**Parameters**

- **key** – key for removal later on
- **priority** – priority on stack
- **fade_ms** – duration of fade

**remove_from_stack_by_key**(*key*)
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 LED update, so if the highest priority settings were removed, the LED will be updated with whatever's below it. If no settings remain after these are removed, the LED will turn off.

**remove_from_stack_by_mode**(*mode: mpf.core.mode.Mode*)
Remove a group of color settings from the stack.

**Parameters** **mode** – Mode which was removed

This method triggers a LED update, so if the highest priority settings were removed, the LED will be updated with whatever's below it. If no settings remain after these are removed, the LED will turn off.

**set_color_correction_profile**(*profile*)
Apply a color correction profile to this LED.

**Parameters** **profile** – An RGBColorCorrectionProfile() instance

**stack = None**
A list of dicts which represents different commands that have come in to set this LED 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 LED 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 LED is fading to. If** a command comes in with no fade, then this will be the same as the 'color' below.

**color: The current color of the LED based on this command. This value** is updated automatically as fades progress, and it's the value that's actually written to the hardware (prior to color correction).

**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 LED).

**mode: Optional mode where the brightness was set. Used to remove** entries when a mode ends.

**classmethod update_leds**(*dt*)

Write leds to hardware platform.

Called periodically (default at the end of every frame) to write the new led colors to the hardware for the LEDs that changed during that frame.

**Parameters** **dt** – time since last call

**write_color_to_hw_driver**()

Set color to hardware platform.

Physically update the LED hardware object based on the 'color' setting of the highest priority setting from the stack.

This method is automatically called whenever a color change has been made (including when fades are active).

## self.machine.lights.*

**class** `mpf.devices.matrix_light.`**MatrixLight**(*self_inner*, **args*, ***kwargs*)

Bases: `mpf.core.system_wide_device.SystemWideDevice`

Represents a light connected to a traditional lamp matrix in a pinball machine.

This light could be an incandescent lamp or a replacement single-color LED. The key is that they're connected up to a lamp matrix.

### 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.

**add_handler**(*callback*)

Register a handler to be called when this light changes state.

**Parameters** **callback** – Monitor callback to add

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

**`fade_task`**(*dt*)
Update the brightness depending on the time for a fade.

**Parameters** **`dt`** – time since last call

**`get_brightness`**()
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 LED, but not always (since physical LEDs 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.

**`classmethod mode_stop`**(*mode: mpf.core.mode.Mode*)
Remove all mode entries from stack.

**Parameters** **`mode`** – Mode which was removed

**`off`**(*fade_ms=0*, *priority=0*, *key=None*, *mode=None*, ***kwargs*)
Turn this light off.

**Parameters**

- **`fade_ms`** – Int of the number of ms you want this light to fade to the brightness in. A value of 0 means it's instant. A value of None (the default) means that it will use this lights'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.
- **`mode`** – Optional mode instance of the mode that is setting this brightness. When a mode ends, entries from the stack with that mode will automatically be removed.
- **`**kwargs`** – Not used. Only included so this method can be used as an event callback since events could pass random kwargs.

**`on`**(*brightness=255*, *fade_ms=None*, *priority=0*, *key=None*, *mode=None*, ***kwargs*)
Turn light on.

Add or updates a brightness entry in this lights's stack, which is how you tell this light how bright you want it to be.

**Parameters**

- **`brightness`** – How bright this light should be, as an int between 0 and 255. 0 is off. 255 is full on. Note that matrix lights in older (even WPC) machines had slow matrix update speeds, and effective brightness levels will be far less than 255.
- **`fade_ms`** – Int of the number of ms you want this light to fade to the brightness in. A value of 0 means it's instant. A value of None (the default) means that it will use this lights'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.
- **`mode`** – Optional mode instance of the mode that is setting this brightness. When a mode ends, entries from the stack with that mode will automatically be removed.
- **`**kwargs`** – Not used. Only included so this method can be used as an event callback since events could pass random kwargs.

**`remove_from_stack_by_key`**(*key*)

Remove a group of brightness settings from the stack.

**Parameters** **`key`** – The key of the settings to remove (based on the 'key' parameter that was originally passed to the brightness() 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.

**`remove_from_stack_by_mode`**(*mode: mpf.core.mode.Mode*)

Remove a group of brightness settings from the stack.

**Parameters** **`mode`** – Mode which was removed

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.

**`remove_handler`**(*callback=None*)

Remove a handler from the list of registered handlers.

**Parameters** **`callback`** – Monitor callback to remove

**`stack = None`**

A list of dicts which represents different commands that have come in to set this light to a certain brightness (and/or fade). Each entry in the list contains the following key/value pairs:

**priority: The relative priority of this brightness 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_brightness: Brightness this light when this command came in. dest_time: Clock time that represents when a fade (from

start_brightness to dest_brightness ) 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_brightness: Brightness of the destination this light is fading** to. If a command comes in with no fade, then this will be the same as the 'brightness' below.

**brightness: The current brightness of the light based on this command.** (0-255) This value is updated automatically as fades progress, and it's the value that's actually written to the hardware.

**key: An arbitrary unique identifier to keep multiple entries in the** stack separate. If a new brightness 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).

**mode: Optional mode where the brightness was set. Used to remove** entries when a mode ends.

**`update_hw_light`**`()`
Set brightness to hardware platform.

Physically updates the light hardware object based on the 'brightness' setting of the highest priority setting from the stack.

This method is automatically called whenever a brightness change has been made (including when fades are active).

**`classmethod update_matrix_lights`**`(`*dt*`)`
Write changed lights to hardware.

**Parameters** **`dt`** – time since last call

### self.machine.magnets.*

**`class`** `mpf.devices.magnet.`**`Magnet`**`(`*self_inner*, **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.

**`release_ball`**`(`***kwargs*`)`
Release the grabbed ball.

**`reset`**`(`***kwargs*`)`
Release ball and disable magnet.

### self.machine.motors.*

**class** `mpf.devices.motor.`**`Motor`**(*machine: mpf.core.machine.MachineController*, *name: str*)
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.

**`reset`**(***kwargs*)
Go to reset position.

### self.machine.multiball_locks.*

**class** `mpf.devices.multiball_lock.`**`MultiballLock`**(*self_inner*, **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.

**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** (*self_inner*, **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

**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.physical_dmds.*

**class** `mpf.devices.physical_dmd.`**PhysicalDmd**(*machine*, *name*)
Bases: `mpf.core.system_wide_device.SystemWideDevice`

A physical DMD.

### Accessing physical_dmds in code

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

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

### Methods & Attributes

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

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

**Parameters** **data** – bytes to send

## self.machine.physical_rgb_dmds.*

**class** `mpf.devices.physical_rgb_dmd.`**PhysicalRgbDmd**(*machine*, *name*)
Bases: `mpf.core.system_wide_device.SystemWideDevice`

A physical DMD.

### Accessing physical_rgb_dmds in code

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

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

#### Methods & Attributes

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

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

**Parameters** **data** – bytes to send

### 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.

**transfer**(***kwargs*)
Transfer a ball to the target playfield.

### self.machine.playfields.*

**class** `mpf.devices.playfield.`**Playfield**(*self_inner*, **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 look for a device tagged with 'ball_add_live'. If you don't provide a source and you don't have a device with the 'ball_add_live' tag, MPF will quit.

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_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**
The number of balls on the playfield.

**expected_ball_received**()
Handle an expected ball.

**classmethod get_additional_ball_capacity**()
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**()
True since it is a playfield.

**mark_playfield_active_from_device_action**()
Mark playfield active because a device on the playfield detected activity.

**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.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.

**add_value** (*value*, *jump=False*, *target=None*)

Add value to a ScoreReelGroup.

You can also pass a negative value to subtract points.

You can control the logistics of how these pulses are applied via the *jump* parameter. If jump is False (default), then this method will respect the proper "sequencing" of reel advances. For example, if the current value is 1700 and the new value is 2200, this method will fire the hundreds reel twice (to go to 1800 then 1900), then on the third pulse it will fire the thousands and hundreds (to go to 2000), then do the final two pulses to land at 2200.

**Parameters**

- **value** – The integer value you'd like to add to (or subtract from) the current value
- **jump** – Optional boolean value which controls whether the reels should "count up" to the new value in the classic EM way (jump=False) or whether they should just jump there as fast as they can (jump=True). Default is False.
- **target** – Optional integer that's the target for where this reel group should end up after it's done advancing. If this is not specified then the target value will be calculated based on the current reel positions, though sometimes this get's wonky if the reel is jumping or moving, so it's best to specify the target if you can.

**assumed_value_int**

Return integer representation of the value we assume is shown on this ScoreReelGroup.

A value of -999 means the value is unknown.

**assumed_value_list**

Return list that holds the values of the reels in the group.

**classmethod chime** (*chime*, ***kwargs*)

Pulse chime.

**get_physical_value_list** ()

Query all the reels in the group and builds a list of their actual current physical state.

This is either the value of the current switch or -999 if no switch is active. This method also updates each reel's physical value.

Returns: List of physical reel values.

**initialize** (***kwargs*)

Initialize the score reels by reading their current physical values and setting each reel's rollover reel.

This is a separate method since it can't run int __iniit__() because all the other reels have to be setup first.

**`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.

**`is_desired_valid`**(*notify_event=False*)
Test to see whether the machine thinks the ScoreReelGroup is currently showing the desired value.

In other words, is the ScoreReelGroup "done" moving? Note this ignores placeholder non-controllable digits.

Returns: True or False

**`light`**(*relight_on_valid=False*, ***kwargs*)
Light up this ScoreReelGroup based on the 'light_tag' in its config.

**`classmethod reel_list_to_int`**(*reel_list*)
Convert an list of integers to a single integer.

This method is like *int_to_reel_list* except that it works in the opposite direction.

The list inputted is expected to be in "reverse" order, with the ones digit in the [0] index position. Values of *None* are converted to zeros. For example, if you pass *[None, 0, 3, 2, 1]*, this method will return an integer value of *12300*.

Note this method does not take into consideration how many reel positions are in this ScoreReelGroup. It just converts whatever you pass it.

**Parameters** **`reel_list`** – The list containing the values for each score reel position.

**Returns** The resultant integer based on the list passed.

**`set_rollover_reels`**()
Call each reel's *set_rollover_reel* method and passes it a pointer to the next higher up reel.

This is how we know whether we're able to advance the next higher up reel when a particular reel rolls over during a step advance.

**`set_value`**(*value=None*, *value_list=None*)
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)*.
- **`value_list`** – A list of the value you'd like the reel group to display.

**`tick`**(*dt*)

Automatically called once per machine tick and checks to see if there are any jumps or advances in progress.

If so, calls those methods.

**`unlight`**(*relight_on_valid=False*, ***kwargs*)

Turn off the lights for this ScoreReelGroup based on the 'light_tag' in its config.

**`validate`**(*value=None*)

Validate that this score reel group is in the position the machine wants it to be in.

If lazy or strict confirm is enabled, this method will also make sure the reels are in their proper physical positions.

**Parameters** **`value`** (*`ignored`*) – This method takes an argument of *value*, but it's not used. It's only there because when reels post their events after they're done moving, they include a parameter of *value* which is the position they're in. So we just need to have this argument listed so we can use this method as an event handler for those events.

## 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.

**`advance`**()

Perform the coil firing to advance this reel one position (up or down).

This method also schedules delays to post the following events:

*reel_<name>_ready*: When the config['repeat_pulse_time'] time is up *reel_<name>_hw_value*: When the config['hw_confirm_time'] time is up

Args:

**Returns: If this method is unable to advance the reel (either because** it's not ready, because it's at its maximum value and does not have rollover capabilities, or because you're trying to advance it in a direction but it doesn't have a coil for that direction), it will return *False*. If it's able to pulse the advance coil, it returns *True*.

**`check_hw_switches`**(*no_event=False*)

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?

**Parameters** **`no_event`** – A boolean switch that allows you to suppress the event posting from this call if you just want to update the values.

**Returns: The hardware value of the switch, either the position or -999.** If the reel is not ready, it returns *False*.

**`set_destination_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.

**`set_rollover_reel`**(*reel*)

Set this reels' rollover_reel to the object of the next higher reel.

## self.machine.servos.*

**class** `mpf.devices.servo.`**`Servo`**(*self_inner*, **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.

**`reset`**(***kwargs*)
Go to reset position.

### self.machine.shot_groups.*

**class** `mpf.devices.shot_group.`**`ShotGroup`**(*machine*, *name*)
Bases: `mpf.core.mode_device.ModeDevice`, `mpf.core.system_wide_device.SystemWideDevice`

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.

**`advance`**(*steps=1*, *mode=None*, *force=False*, ***kwargs*)
Advance the current active profile from every shot in the group one step forward.

**`check_for_complete`**(*mode*)
Check all the shots in this shot group.

If they are all in the same state, then a complete event is posted.

**`disable`**(*mode=None*, ***kwargs*)
Disable this shot group.

Also disables all the shots in this group.

**`disable_rotation`**(***kwargs*)
Disable shot rotation.

If disabled, rotation events do not actually rotate the shots.

**enable** (*mode=None*, *profile=None*, ***kwargs*)
Enable this shot group.

Also enables all the shots in this group.

**enable_rotation** (***kwargs*)
Enable shot rotation.

If disabled, rotation events do not actually rotate the shots.

**enabled**
Return true if enabled.

**hit** (*mode*, *profile*, *state*, ***kwargs*)
One of the member shots in this shot group was hit.

**Parameters**

- **profile** – String name of the active profile of the shot that was hit.
- **mode** – unused
- **kwargs** – unused

**remove_active_profile** (*mode*, ***kwargs*)
Remove the current active profile from every shot in the group.

**reset** (*mode=None*, ***kwargs*)
Reset each of the shots in this group back to the initial state in whatever shot profile they have applied.

This is the same as calling each shot's reset() method one-by-one.

**rotate** (*direction=None*, *states=None*, *exclude_states=None*, *mode=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.shots.*

**class** mpf.devices.shot.**Shot** (*machine*, *name*)

Bases: `mpf.core.mode_device.ModeDevice`, `mpf.core.system_wide_device.SystemWideDevice`

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**

*List of tuples* – (id, current_position_index, next_switch)

**add_profile** (*profile_dict*)

Add a profile to shot.

**advance** (*steps=1*, *mode=None*, *force=False*, ***kwargs*)

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.

**deregister_group** (*group*)

Deregister a group.

Notify this shot that it is no longer part of this group. Note this is called by :class:`ShotGroup`. If you want to manually remove a shot from a group, do it from there.

**disable** (*mode=None*, ***kwargs*)

Disable this shot.

If the shot is not enabled, hits to it will not be processed.

**enable** (*mode=None*, *profile=None*, ***kwargs*)

Enable shot.

**enabled**

Return true if enabled.

**get_profile_by_key** (*key*, *value*)

Return profile for a key value pair.

**hit** (*mode='default#$%'*, *_wf=None*, ***kwargs*)
Advance the currently-active shot profile.

**Parameters**

- **mode** – (Optional) The mode instance that was hit. If this is not specified, this hit is registered via the highest-priority mode that this shot is active it. A value of None represents the base machine config (e.g. no Mode). The crazy default string it so this method can differentiate between no mode specified (where it uses the highest one) and a value of "None" which is the base machine-wide config.
- **_wf** – (Internal use only) A list of remaining modes from the enable table of the original hit. Used to waterfall hits (which is where hits are cascaded down to this shot in lower priority modes if blocking is not set.

Note that the shot must be enabled in order for this hit to be processed.

**jump** (*mode*, *state*, *show_step=1*, *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.

**player_turn_start** (*player*, ***kwargs*)
Update the player reference to the current player and to apply the default machine-wide shot profile.

Called by the shot profile manager when a player's turn starts.

**player_turn_stop** ()
Remove the profiles from the shot and remove the player reference.

Called by the shot profile manager when the player's turn ends.

**register_group** (*group*)
Register a group.

Notify this shot that it has been added to a group, meaning it will update this group of its state changes. Note this is called by :class:`ShotGroup`. If you want to manually add a shot to a group, do it from there.

**remove_active_profile** (*mode='default#$%'*, ***kwargs*)
Remove the active profile.

**remove_profile_by_mode** (*mode*)
Remove profile for mode.

**reset** (*mode=None*, ***kwargs*)
Reset the shot profile for the passed mode back to the first state (State 0) and reset all sequences.

**update_current_state_name** (*mode*)
Update current state name.

**update_profile** (*profile=None*, *enable=None*, *mode=None*)
Update profile.

### self.machine.switches.*

**class** `mpf.devices.switch.`**`Switch`**(*self_inner*, **args*, ***kwargs*)

Bases: `mpf.core.system_wide_device.SystemWideDevice`

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 for this switch.

**`get_configured_switch`**()

Reconfigure switch.

**`hw_state = None`**

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.

**`remove_handler`**(*callback*, *state=1*, *ms=0*)

Remove switch handler for this switch.

**`state = None`**

The logical state of a switch. 1 = active, 0 = inactive. This takes into consideration the NC or NO settings for the switch.

### self.machine.timed_switches.*

**class** `mpf.devices.timed_switch.`**`TimedSwitch`**(*self_inner*, **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.

## 7.3.3 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.

Its 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 mode is active.

**`add_mode_event_handler`**(*event*, *handler*, *priority=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*, *console_level='basic'*, *file_level='basic'*)

Configure the logging for the module this class is mixed into.

**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*, **args*, ***kwargs*)

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*, **args*, ***kwargs*)

Log a message at the error level.

These messages will always be shown in the console and the log file.

**`info_log`**(*msg*, **args*, ***kwargs*)

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().

**`result_of_start_request`**(*ev_result=True*)

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*)

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`**()

Called when the a switch tagged with *start* is activated.

**start_button_released**()
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=None*, ***kwargs*)
Stop this mode.

**Parameters** ****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.

**warning_log**(*msg*, **args*, ***kwargs*)
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 mode is active.

**add_mode_event_handler**(*event*, *handler*, *priority=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*, *console_level='basic'*, *file_level='basic'*)
Configure the logging for the module this class is mixed into.

**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*, **args*, ***kwargs*)
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*, **args*, ***kwargs*)
Log a message at the error level.

These messages will always be shown in the console and the log file.

**`hurry_up`**(***kwargs*)
Changes 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.

**`info_log`**(*msg*, **args*, ***kwargs*)
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().

**`start`**(*mode_priority=None*, *callback=None*, ***kwargs*)
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=None*, ***kwargs*)
Stop this mode.

**Parameters** `**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.

**warning_log**(*msg*, **args*, ***kwargs*)
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 mode is active.

**add_mode_event_handler**(*event*, *handler*, *priority=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*, *console_level='basic'*, *file_level='basic'*)
Configure the logging for the module this class is mixed into.

**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*, **args*, ***kwargs*)
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*, **args*, ***kwargs*)
Log a message at the error level.

These messages will always be shown in the console and the log file.

**`info_log`**(*msg*, **args*, ***kwargs*)
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().

**`start`**(*mode_priority=None*, *callback=None*, ***kwargs*)
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=None*, ***kwargs*)
Stop this mode.

**Parameters** **`**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.

**`warning_log`**(*msg*, **args*, ***kwargs*)
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 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*, *handler*, *priority=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*, *console_level='basic'*, *file_level='basic'*)

Configure the logging for the module this class is mixed into.

**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*, **args*, ***kwargs*)
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*, **args*, ***kwargs*)
Log a message at the error level.

These messages will always be shown in the console and the log file.

**`info_log`**(*msg*, **args*, ***kwargs*)
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().

**`start`**(*mode_priority=None*, *callback=None*, ***kwargs*)
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=None*, ***kwargs*)
Stop this mode.

**Parameters** **`**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.

**`toggle_credit_play`**(***kwargs*)
Toggle between free and credits play.

**`warning_log`**(*msg*, **args*, ***kwargs*)
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.mode.Mode*

Base mode that runs an active game on a pinball machine.

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 mode is active.

**add_mode_event_handler**(*event*, *handler*, *priority=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.

**award_extra_ball**()

Called when the same player should shoot again.

**ball_drained**(*balls=0*, ***kwargs*)

Ball drained.

**ball_ended**(*ev_result=True*, ***kwargs*)

Called when the ball has successfully ended.

This method is called after all the registered handlers of the queue event *ball_ended* finish. (So typically this means that animations have finished, etc.)

This method also decides if the same player should shoot again (if there's an extra ball) or whether the machine controller should rotate to the next player. It will also end the game if all players and balls are done.

**ball_ending**()
Start the ball ending process.

This method posts the queue event *ball_ending*, giving other modules an opportunity to finish up whatever they need to do before the ball ends. Once all the registered handlers for that event have finished, this method calls `ball_ended()`.

Currently this method also disables the autofire_coils and flippers, though that's temporary as we'll move those into config file options.

**ball_started**(*ev_result=True*, ***kwargs*)
Ball started.

**ball_starting**(*is_extra_ball=False*)
Called when a new ball is starting.

Note this method is called for each ball that starts, even if it's after a Shoot Again scenario for the same player.

Posts a queue event called *ball_starting*, giving other modules the opportunity to do things before the ball actually starts. Once that event is clear, this method calls `ball_started()`.

**balls_in_play**
Return balls in play.

**configure_logging**(*logger*, *console_level='basic'*, *file_level='basic'*)
Configure the logging for the module this class is mixed into.

**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*, **args*, ***kwargs*)
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*, **args*, ***kwargs*)
Log a message at the error level.

These messages will always be shown in the console and the log file.

**game_ended**(***kwargs*)
Actually ends the game once the *game_ending* event is clear.

Eventually this method will do lots of things. For now it just advances the machine flow which ends the `Game` mode and starts the `Attract` mode.

**game_ending**()
Called when the game decides it should end.

This method posts the queue event *game_ending*, giving other modules an opportunity to finish up whatever they need to do before the game ends. Once all the registered handlers for that event have finished, this method calls `game_end()`.

**game_started**(*ev_result=True*, ***kwargs*)
All the modules that needed to do something on game start are done, so our game is officially 'started'.

**info_log**(*msg*, **args*, ***kwargs*)
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().

**player_add_success**(*player*, ***kwargs*)
Called when a new player is successfully added to the current game.

This includes when the first player is added.

**player_rotate**()
Rotate the game to the next player.

This method is called after a player's turn is over, so it's even used in single-player games between balls.

All it does really is set `player` to the next player's number.

Args:

**player_turn_start**()
Called at the beginning of a player's turn.

Note this method is only called when a new player is first up. So if the same player shoots again due to an extra ball, this method is not called again.

**player_turn_stop**()
Called when player turn stopped.

**request_player_add**(***kwargs*)
Called by any module that wants 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_add_success* is posted with a reference to the new player object as a *player* kwarg.

**start**(*mode_priority=None*, *callback=None*, ***kwargs*)
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=None*, ***kwargs*)
Stop this mode.

**Parameters** ****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.

**warning_log**(*msg*, **args*, ***kwargs*)
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`

Mode which tracks high scores and lets the player enter its 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 mode is active.

**add_mode_event_handler**(*event*, *handler*, *priority=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*, *console_level='basic'*, *file_level='basic'*)
Configure the logging for the module this class is mixed into.

**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*, **args*, ***kwargs*)
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*, **args*, ***kwargs*)
Log a message at the error level.

These messages will always be shown in the console and the log file.

**info_log**(*msg*, **args*, ***kwargs*)
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().

**start**(*mode_priority=None*, *callback=None*, ***kwargs*)
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=None*, ***kwargs*)
Stop this mode.

**Parameters** ****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.

**warning_log**(*msg*, **args*, ***kwargs*)
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 mode is active.

**`add_mode_event_handler`**(*event*, *handler*, *priority=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*, *console_level='basic'*, *file_level='basic'*)

Configure the logging for the module this class is mixed into.

**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*, **args*, ***kwargs*)
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*, **args*, ***kwargs*)
Log a message at the error level.

These messages will always be shown in the console and the log file.

**info_log** (*msg*, **args*, ***kwargs*)
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().

**start** (*mode_priority=None*, *callback=None*, ***kwargs*)
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=None*, ***kwargs*)
Stop this mode.

**Parameters** ****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.

**warning_log** (*msg*, **args*, ***kwargs*)
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*, *config*, *name*, *path*)
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 mode is active.

**`add_mode_event_handler`**(*event*, *handler*, *priority=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*, *console_level='basic'*, *file_level='basic'*)
Configure the logging for the module this class is mixed into.

**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*, **args*, ***kwargs*)
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*, **args*, ***kwargs*)
Log a message at the error level.

These messages will always be shown in the console and the log file.

**`info_log`**(*msg*, **args*, ***kwargs*)
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().

**start** (*mode_priority=None*, *callback=None*, ***kwargs*)

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=None*, ***kwargs*)

Stop this mode.

**Parameters** ****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.

**warning_log** (*msg*, **args*, ***kwargs*)

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.

#### 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 mode is active.

**add_mode_event_handler** (*event*, *handler*, *priority=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*, *console_level='basic'*, *file_level='basic'*)
Configure the logging for the module this class is mixed into.

**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*, **args*, ***kwargs*)
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*, **args*, ***kwargs*)
Log a message at the error level.

These messages will always be shown in the console and the log file.

**`info_log`**(*msg*, **args*, ***kwargs*)
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().

**`reset_warnings`**(***kwargs*)
Reset the tilt warnings for the current player.

**`slam_tilt`**(***kwargs*)
Process a slam tilt.

**`start`**(*mode_priority=None*, *callback=None*, ***kwargs*)
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=None*, ***kwargs*)
Stop this mode.

**Parameters** ****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.

**tilt**(***kwargs*)
Cause the ball to tilt.

**tilt_settle_ms_remaining**()
Return the amount of milliseconds remaining until the tilt settle time has 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*, **args*, ***kwargs*)
Log a message at the warning level.

These messages will always be shown in the console and the log file.

## 7.3.4 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.**HardwarePlatform**(*machine*)
Bases: *mpf.platforms.openpixel.HardwarePlatform*

Base class for the open pixel hardware platform.

**Parameters** **machine** – The main `MachineController` object.

#### 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.`**`HardwarePlatform`**(*machine*)

Bases: `mpf.core.platform.ServoPlatform`, `mpf.core.platform.MatrixLightsPlatform`, `mpf.core.platform.GiPlatform`, `mpf.core.platform.DmdPlatform`, `mpf.core.platform.LedPlatform`, `mpf.core.platform.SwitchPlatform`, `mpf.core.platform.DriverPlatform`

Platform class for the FAST hardware controller.

**Parameters** **`machine`** – The main `MachineController` instance.

#### 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: dict*) → mpf.platforms.fast.fast_driver.FASTDriver

Configure a driver.

**Parameters** **`config`** – Driver config.

Returns: Driver object

**`configure_gi`**(*config: dict*) → mpf.platforms.fast.fast_gi.FASTGIString

Configure a GI.

**Parameters** **`config`** – GI config.

Returns: GI object.

**configure_led**(*config: dict*, *channels: int*)
Configure a WS2812 LED.

**Parameters**

- **config** – LED config.
- **channels** – Number of channels (3 for RGB)

Returns: LED object.

**configure_matrixlight**(*config: dict*) → mpf.platforms.fast.fast_light.FASTMatrixLight
Configure a matrix light.

**Parameters** **config** – Matrix light config.

Returns: Matrix light object.

**configure_servo**(*config: dict*)
Configure a servo.

**Parameters** **config** – Servo config.

Returns: Servo object.

**configure_switch**(*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.

**convert_number_from_config**(*number*)
Convert a number from config format to int.

**classmethod get_coil_config_section**()
Return coil config section.

**classmethod get_coil_overwrite_section**()
Return coil overwrite section.

**get_hw_switch_states**()
Return hardware states.

**classmethod get_switch_config_section**()
Return switch config section.

**initialize**()
Initialise platform.

**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*, *coil*)
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*, *coil*)
Set pulse on hit rule on driver.

**`stop`**()
Stop platform and close connections.

**`update_leds`**(*dt*)
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.

**Parameters** **`dt`** – time since last call

**`validate_switch_overwrite_section`**(*switch: mpf.devices.switch.Switch*, *config_overwrite: dict*) → dict

Validate switch overwrite section for platform.

**Parameters**

- **`switch`** – switch to validate
- **`config_overwrite`** – overwrite config to validate

Returns: Validated config.

## self.machine.hardware_platforms['i2c_servo_controller']

**class** `mpf.platforms.i2c_servo_controller.`**`HardwarePlatform`**(*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`**(*config*)

Configure servo.

**`initialize`**()

Method is called after all hardware platforms were instantiated.

**`stop`**()

Stop platform.

## self.machine.hardware_platforms['openpixel']

**class** `mpf.platforms.openpixel.`**`HardwarePlatform`**(*machine*)

Bases: `mpf.core.platform.LedPlatform`

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_led`**(*config*, *channels*)
Configure an LED.

**Parameters**

- **`config`** – config dict of led
- **`channels`** – number of channels (up to three are supported)

**`initialize`**()
Initialise openpixel platform.

**`stop`**()
Stop platform.

## self.machine.hardware_platforms['opp']

**class** `mpf.platforms.opp.opp.`**`HardwarePlatform`**(*machine*)
Bases: `mpf.core.platform.MatrixLightsPlatform`, `mpf.core.platform.LedPlatform`, `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*, *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*.

**`configure_driver`**(*config: dict*)
Configure a driver.

**Parameters** **`config`** – Config dict.

**configure_led**(*config: dict*, *channels: int*)
Configure LED.

**Parameters**

- **config** – Config dict.
- **channels** – Number of channels. OPP supports up to three.

**configure_matrixlight**(*config*)
Configure a direct incandescent bulb.

**Parameters** **config** – Config dict.

**configure_switch**(*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.

**classmethod get_hold_value**(*coil*)
Get OPP hold value (0-15).

**get_hw_switch_states**()
Get initial hardware switch states.

This changes switches from active low to active high

**classmethod get_minimum_off_time**(*coil*)
Return minimum off factor.

The hardware applies this factor to pulse_ms to prevent the coil from burning.

**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.

**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.

**`reconfigure_driver`** (*driver*, *use_hold: bool*)
Reconfigure a driver.

**Parameters**

- **`driver`** – Driver object.
- **`use_hold`** – Whether this driver stays enabled after a trigger or not.

**`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*, *disable_switch*, *coil*)
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*, *coil*)
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*, *coil*)
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*, *coil*)
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.

**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.**HardwarePlatform**(*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`**(*config*, *callback*)
Configure the accelerometer on the P3-ROC.

**`configure_driver`**(*config*)
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_gi`**(*config*)
Configure a GI driver on the P3-Roc.

GIs are coils in P3-Roc

**`configure_matrixlight`**(*config*)
Configure a matrix light in P3-Roc.

**`configure_switch`**(*config*)
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 reference to the switch object that was just created. proc_num : Integer of the actual hardware switch number the P3-ROC

uses to refer to this switch. Typically your machine configuration files would specify a switch number like *SD12* or *7/5*. This *proc_num* is an int between 0 and 255.

**Return type** switch

**`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)

**`i2c_read16`**(*address*, *register*)
Read an 16-bit value from the I2C bus of the P3-Roc.

**`i2c_read8`**(*address*, *register*)
Read an 8-bit value from the I2C bus of the P3-Roc.

**`i2c_write8`**(*address*, *register*, *value*)
Write an 8-bit value to the I2C bus of the P3-Roc.

**classmethod scale_accelerometer_to_g**(*raw_value*)
Convert internal representation to g.

**tick**(*dt*)
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.**HardwarePlatform**(*machine*)
Bases: `mpf.platforms.p_roc_common.PROCBasePlatform`, `mpf.core.platform.DmdPlatform`

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*)
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_gi**(*config*)
Configure a GI.

**configure_matrixlight**(*config*)
Configure a matrix light.

**configure_switch**(*config*)
Configure a P-ROC switch.

**Parameters config** – Dictionary of settings for the switch. In the case of the P-ROC, it uses the following:

**Returns**

A reference to the switch object that was just created. proc_num : Integer of the actual hardware switch number the P-ROC

uses to refer to this switch. Typically your machine configuration files would specify a switch number like *SD12* or *7/5*. This *proc_num* is an int between 0 and 255.

**Return type** switch

**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)

**tick**(*dt*)

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.`**HardwarePlatform**(*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**(*config*)

Configure a servo device in paltform.

**Parameters** **config** (*dict*) – Configuration of device

**initialize**()

Method is called after all hardware platforms were instantiated.

**stop**()

Close serial.

### self.machine.hardware_platforms['smart_virtual']

**class** `mpf.platforms.smart_virtual.`**HardwarePlatform**(*machine*)

Bases: *`mpf.platforms.virtual.HardwarePlatform`*

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*)
Configure driver.

**`initialize`**()
Initialise platform.

### self.machine.hardware_platforms['smartmatrix']

**class** `mpf.platforms.smartmatrix.`**`SmartMatrix`**(*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`**()
Configure rgb dmd.

**`initialize`**()
Initialise platform.

**`stop`**()
Stop platform.

**`update`**(*data*)
Update DMD data.

### self.machine.hardware_platforms['snux']

**class** `mpf.platforms.snux.`**`HardwarePlatform`**(*machine*)
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`**
True when A side cannot be switches off right away.

**`c_side_active`**
True when 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*)
Configure a driver on the snux board.

**Parameters** **`config`** – Driver config dict

**`driver_action`**(*driver*, *coil*, *milliseconds*)
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.
- **`milliseconds`** – Integer of the number of milliseconds this action is for. 0 = pulse, -1 = enable (hold), any other value is a timed action (either pulse or long_pulse)

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**(*dt*)
Snux main loop.

Called based on the timer_tick event

**Parameters** **dt** – time since last call

**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.MatrixLightsPlatform, 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*)
Configure a driver on Stern Spike.

**configure_matrixlight**(*config*)
Configure a light on Stern Spike.

**configure_switch**(*config*)
Configure switch on Stern Spike.

**get_hw_switch_states**()
Return current switch states.

**initialize**()
Initialise platform.

**send_cmd_and_wait_for_response**(*node*, *cmd*, *data*, *response_len*) → typing.Generator[[int, NoneType], str]
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*, *disable_switch*, *coil*)
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*, *coil*)
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 0x80 0x00 0x4a 0x40 0x00 0x00 0x06 0x05 Len: 25

**set_pulse_on_hit_and_release_rule** (*enable_switch*, *coil*)
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*, *coil*)
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 0x14 0x00 0x00 0x00 0x38 0x00 0x40 0x00 0x00 0x00 0x00 0x00 Len: 25

**stop** ()
Stop hardware and close connections.

### self.machine.hardware_platforms['virtual']

**class** `mpf.platforms.virtual.`**HardwarePlatform** (*machine*)
Bases: `mpf.core.platform.AccelerometerPlatform`, `mpf.core.platform.I2cPlatform`, `mpf.core.platform.ServoPlatform`, `mpf.core.platform.MatrixLightsPlatform`, `mpf.core.platform.GiPlatform`, `mpf.core.platform.LedPlatform`, `mpf.core.platform.SwitchPlatform`, `mpf.core.platform.DriverPlatform`, `mpf.core.platform.DmdPlatform`, `mpf.core.platform.RgbDmdPlatform`

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**(*config*, *callback*)
Configure accelerometer.

**configure_dmd**()
Configure DMD.

**configure_driver**(*config*)
Configure driver.

**configure_gi**(*config*)
Configure GI.

**configure_led**(*config*, *channels*)
Configure led.

**configure_matrixlight**(*config*)
Configure matrix light.

**configure_rgb_dmd**()
Configure DMD.

**configure_servo**(*config*)
Configure a servo device in paltform.

**configure_switch**(*config*)
Configure switch.

**get_hw_switch_states**()
Return hw switch states.

**i2c_read16**(*address*, *register*)
Read I2C.

**i2c_read8**(*address*, *register*)
Read I2C.

**i2c_write8**(*address*, *register*, *value*)
Write to I2C.

**initialize**()
Initialise platform.

**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_overwrite_section**(*driver*, *config_overwrite*)
Validate coil overwrite sections.

**validate_coil_section**(*driver*, *config*)
Validate coil sections.

**validate_switch_overwrite_section**(*switch*, *config_overwrite*)
Validate switch overwrite sections.

**validate_switch_section**(*switch*, *config*)
Validate switch sections.

## 7.3.5 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.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*)
Parse short config version.

**play**(*settings*, *context*, *priority=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.gi_player

**class** `mpf.config_players.gi_player.`**GiPlayer**(*machine*)
    Bases: `mpf.config_players.device_config_player.DeviceConfigPlayer`

    Enables GIs based on config.

#### Accessing the gi_player in code

The gi_player is available via `self.machine.gi_player`.

#### Methods & Attributes

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

**`clear_context`**(*context*)
Disable all used GIs at the end.

**`get_express_config`**(*value*)
Parse express config.

**`play`**(*settings*, *context*, *calling_context*, *priority=0*, ***kwargs*)
Enable GIs.

### self.machine.led_player

**class** `mpf.config_players.led_player.`**`LedPlayer`**(*machine*)
Bases: `mpf.config_players.device_config_player.DeviceConfigPlayer`

Sets LED color based on config.

#### Accessing the led_player in code

The led_player is available via `self.machine.led_player`.

#### Methods & Attributes

The led_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.

**`get_full_config`**(*value*)
Return full config.

**`play`**(*settings*, *context*, *calling_context*, *priority=0*, ***kwargs*)
Set LED color based on config.

### 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 brightness which was set in context.

**get_express_config**(*value*)
Parse express config.

**get_full_config**(*value*)
Return full config.

**play**(*settings, context, calling_context, priority=0, **kwargs*)
Set brightness 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, 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.

**get_express_config** (*value*)
    No express config.

**play** (*settings*, *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.

**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.score_player

**class** `mpf.config_players.score_player.`**ScorePlayer** (*machine*)
    Bases: `mpf.core.config_player.ConfigPlayer`

Posts events based on config.

#### Accessing the score_player in code

The score_player is available via `self.machine.score_player`.

#### Methods & Attributes

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

**clear_context** (*context*)
Clear context.

**get_express_config** (*value*)
Parse express config.

**get_list_config** (*value*)
Parse list.

**play** (*settings*, *context*, *calling_context*, *priority=0*, ***kwargs*)
Score variable.

**validate_config_entry** (*settings*, *name*)
Validate one entry of this player.

### 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.

**play** (*settings*, *context*, *calling_context*, *priority=0*, *queue=None*, ***kwargs*)
Play, start, stop, pause, resume or advance show based on config.

### self.machine.trigger_player

**class** `mpf.config_players.trigger_player.`**TriggerPlayer** (*machine*)
Bases: `mpf.config_players.device_config_player.DeviceConfigPlayer`

Executes BCP triggers based on config.

#### Accessing the trigger_player in code

The trigger_player is available via `self.machine.trigger_player`.

#### Methods & Attributes

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

**`get_express_config`**(*value*)
Not supported.

**`play`**(*settings*, *context*, *calling_context*, *priority=0*, ***kwargs*)
Execute BCP triggers.

### 7.3.6 Testing Class API

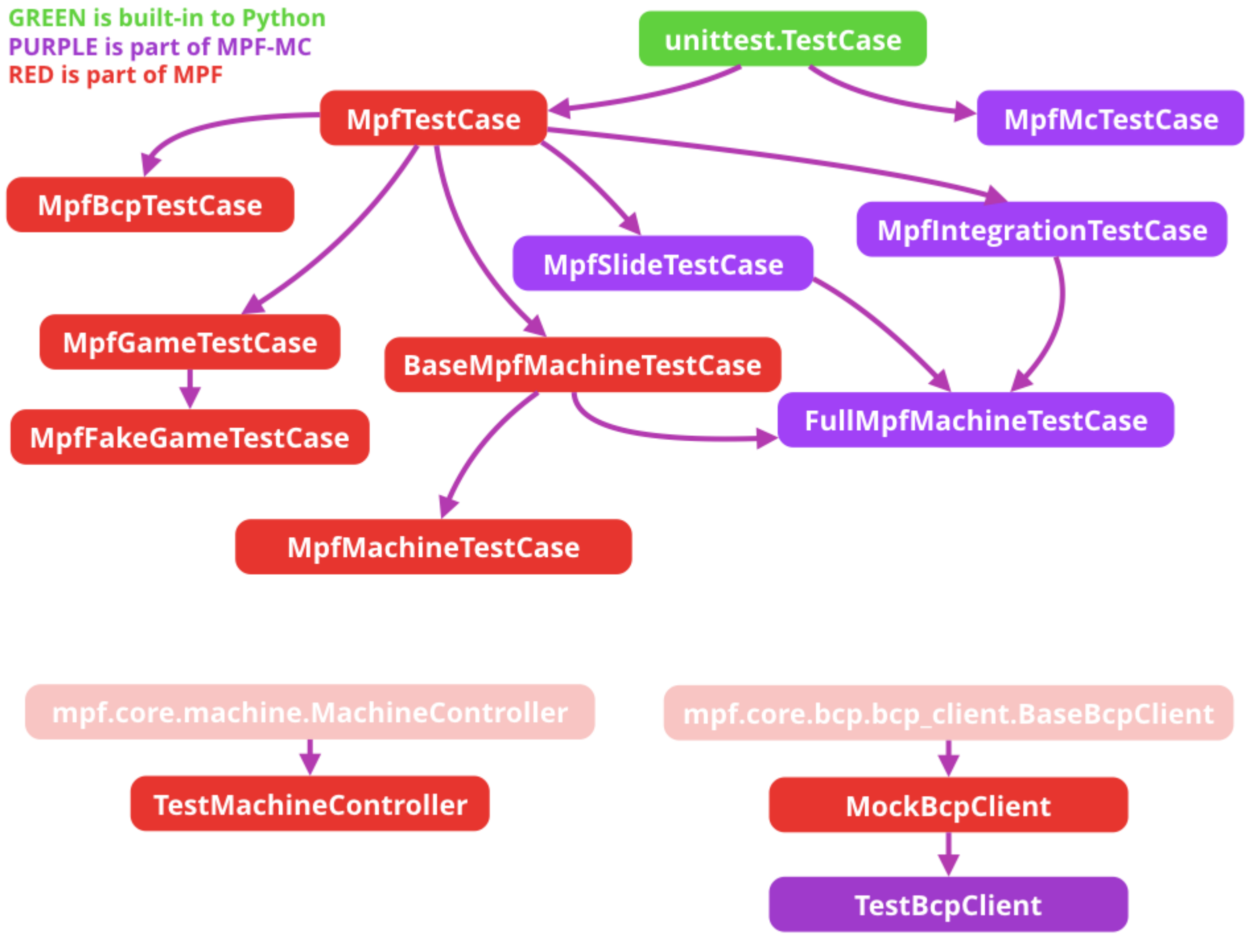
MPF includes several unit test classes which you can use to *write tests which test MPF* or to *write tests for your own game*.

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`

### Methods & Attributes

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

**warning_log**(*msg*, **args*, ***kwargs*)
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*

MpfTestCase with mocked BCP.

### Methods & Attributes

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

**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.

**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.

**assertEventCalledWith** (*event_name*, ***kwargs*)
Assert that event was called with kwargs.

**assertEventNotCalled** (*event_name*)
Assert that event was not called.

**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.

**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).

**assertShotProfile**(*shot_name*, *profile_name*)
Assert that the highest priority profile for a shot is a certain profile name.

**assertShotProfileState**(*shot_name*, *state_name*)
Assert that the highest priority profile for a shot is in a certain state.

**assertShotShow**(*shot_name*, *show_name*)
Assert that the highest priority running show for a shot is a certain show name.

**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** ()
Override this method in your own test class to point to the config file you need for your tests.

**getMachinePath** ()
Override this method in your own test class to point to the machine folder you need for your tests.

Path is related to the MPF package root

**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.

**unittest_verbosity** ()
Return the verbosity setting of the currently running unittest program, or 0 if none is running.

## MpfFakeGameTestCase

**class** `mpf.tests.MpfFakeGameTestCase.`**MpfFakeGameTestCase** (*methodName*)
Bases: *mpf.tests.MpfGameTestCase.MpfGameTestCase*

Testcase for fake game.

### Methods & Attributes

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

**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.

**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.

**assertEventCalledWith** (*event_name*, ***kwargs*)
Assert that event was called with kwargs.

**assertEventNotCalled** (*event_name*)
Assert that event was not called.

**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.

**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).

**`assertShotProfile`**(*shot_name*, *profile_name*)
Assert that the highest priority profile for a shot is a certain profile name.

**`assertShotProfileState`**(*shot_name*, *state_name*)
Assert that the highest priority profile for a shot is in a certain state.

**`assertShotShow`**(*shot_name*, *show_name*)
Assert that the highest priority running show for a shot is a certain show name.

**`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.

**fill_troughs**()

Fill all troughs.

**getConfigFile**()

Override this method in your own test class to point to the config file you need for your tests.

**getMachinePath**()

Override this method in your own test class to point to the machine folder you need for your tests.

Path is related to the MPF package root

**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_two_player_game**()
Start two player game.

**unittest_verbosity**()
Return the verbosity setting of the currently running unittest program, or 0 if none is running.

### MpfGameTestCase

**class** mpf.tests.MpfGameTestCase.**MpfGameTestCase**(*methodName*)
Bases: *mpf.tests.MpfTestCase.MpfTestCase*

Testcase for games.

#### Methods & Attributes

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

**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.

**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.

**assertEventCalledWith**(*event_name*, ***kwargs*)
Assert that event was called with kwargs.

**assertEventNotCalled**(*event_name*)
Assert that event was not called.

**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.

**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).

**assertShotProfile**(*shot_name*, *profile_name*)
Assert that the highest priority profile for a shot is a certain profile name.

**assertShotProfileState**(*shot_name*, *state_name*)
Assert that the highest priority profile for a shot is in a certain state.

**assertShotShow**(*shot_name*, *show_name*)
Assert that the highest priority running show for a shot is a certain show name.

**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.

**fill_troughs**()
Fill all troughs.

**getConfigFile**()
Override this method in your own test class to point to the config file you need for your tests.

**getMachinePath**()
Override this method in your own test class to point to the machine folder you need for your tests.

Path is related to the MPF package root

**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_two_player_game**()
Start two player game.

**unittest_verbosity**()
Return the verbosity setting of the currently running unittest program, or 0 if none is running.

## 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.

**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.

**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.

**assertEventCalledWith** (*event_name*, ***kwargs*)
Assert that event was called with kwargs.

**assertEventNotCalled** (*event_name*)
Assert that event was not called.

**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.

**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).

**assertShotProfile**(*shot_name*, *profile_name*)
Assert that the highest priority profile for a shot is a certain profile name.

**assertShotProfileState**(*shot_name*, *state_name*)
Assert that the highest priority profile for a shot is in a certain state.

**assertShotShow**(*shot_name*, *show_name*)
Assert that the highest priority running show for a shot is a certain show name.

**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.

**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.

**unittest_verbosity**()
Return the verbosity setting of the currently running unittest program, or 0 if none is running.

## MpfTestCase

**class** `mpf.tests.MpfTestCase.`**MpfTestCase**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

### Methods & Attributes

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

**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.

**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.

**assertEventCalledWith**(*event_name*, ***kwargs*)
Assert that event was called with kwargs.

**assertEventNotCalled**(*event_name*)
Assert that event was not called.

**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.

**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).

**assertShotProfile**(*shot_name*, *profile_name*)
Assert that the highest priority profile for a shot is a certain profile name.

**assertShotProfileState**(*shot_name*, *state_name*)
Assert that the highest priority profile for a shot is in a certain state.

**assertShotShow**(*shot_name*, *show_name*)
Assert that the highest priority running show for a shot is a certain show name.

**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**()
Override this method in your own test class to point to the config file you need for your tests.

**getMachinePath**()
Override this method in your own test class to point to the machine folder you need for your tests.

Path is related to the MPF package root

**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.

**unittest_verbosity**()
Return the verbosity setting of the currently running unittest program, or 0 if none is running.

### TestDataManager

**class** mpf.tests.TestDataManager.**TestDataManager**(*data*)
Bases: *mpf.core.data_manager.DataManager*

### 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.

**`remove_key`**(*key*)
Remove key by name.

**`save_key`**(*key*, *value*, *delay_secs=0*)
Update an individual key and then write the entire dictionary to disk.

**Parameters**

- **`key`** – String name of the key to add/update.
- **`value`** – Value of the key
- **`delay_secs`** – Optional number of seconds to wait before writing the data to disk. Default is 0.

**`warning_log`**(*msg*, **args*, ***kwargs*)
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*, *clock*, *mock_data*, *enable_plugins=False*)

Bases: *`mpf.core.machine.MachineController`*

MachineController used in tests.

### 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*)
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*)
Clear a boot hold.

**`create_machine_var`**(*name*, *value=0*, *persist=False*, *expire_secs=None*, *silent=False*)
Create a new machine variable.

**Parameters**

- **`name`** – String name of the variable.

- **value** – The value of the variable. This can be any Type.
- **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 be loaded with a value of 0. 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.

**get_machine_var**(*name*)
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*, *overwrite*)
Return platform section.

**init_done**()
Finish init.

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

**is_machine_var**(*name*)
Return true if machine variable exists.

**power_off**(***kwargs*)
Attempt to perform a power down of the pinball machine and ends MPF.

This method is not yet implemented.

**register_boot_hold**(*hold*)
Register a boot hold.

**register_monitor**(*monitor_class*, *monitor*)
Register a monitor.

**Parameters**

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

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*)
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=''*, *endswith=''*)
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`**()

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`**()

Start the main machine run loop.

**`set_default_platform`**(*name*)

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*, *value*, *force_events=False*)

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.
- **`force_events`** – Boolean which will force the event posting, the machine monitor callback, and writing the variable to disk (if it's set to persist). By default these things only happen if the new value is different from the old value.

**`stop`**(***kwargs*)

Perform a graceful exit of MPF.

**`validate_machine_config_section`**(*section*)

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*, **args*, ***kwargs*)

Log a message at the warning level.

These messages will always be shown in the console and the log file.

### 7.3.7 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*, *playfield*)
Bases: mpf.core.mpf_controller.MpfController

Ball search controller.

#### 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.

**cancel_ball_search**(***kwargs*)
Cancel the current ball search and mark the ball as missing.

**configure_logging**(*logger*, *console_level='basic'*, *file_level='basic'*)
Configure the logging for the module this class is mixed into.

**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*, **args*, ***kwargs*)
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.

Will stop the ball search if it is running.

**enable**(***kwargs*)
Enable but do not start ball search.

Ball search is started by a timeout. Enable also resets that timer.

**error_log**(*msg*, **args*, ***kwargs*)
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.

Did not find the missing ball. Execute the failed action which either adds a replacement ball or ends the game.

**info_log**(*msg*, **args*, ***kwargs*)
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().

**register**(*priority*, *callback*, *name*)
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

**request_to_start_game**(***kwargs*)
Method registered for the *request_to_start_game* event.

Prevents the game from starting while ball search is running.

**reset_timer**()
Reset the timer to start ball search.

This also cancels an active running ball search.

This is called by the playfield anytime a playfield switch is hit.

**start**()
Actually start ball search.

**stop**()
Stop an active 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*, **args*, ***kwargs*)
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 manager.

**static load**(*filename*, *verify_version=False*, *halt_on_error=True*, *round_trip=False*)
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*, ***kwargs*)
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*, *console_level='basic'*, *file_level='basic'*)
Configure the logging for the module this class is mixed into.

**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*, **args*, ***kwargs*)
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*, **args*, ***kwargs*)
Log a message at the error level.

These messages will always be shown in the console and the log file.

**info_log**(*msg*, **args*, ***kwargs*)
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().

**warning_log**(*msg*, **args*, ***kwargs*)
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*, *config: dict*, *name: str*, *path*)
Bases: *`mpf.core.logging.LogMixin`*

Parent class for in-game mode code.

#### 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 mode is active.

**`add_mode_event_handler`**(*event*, *handler*, *priority=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 = None`**
Controls whether this mode is stopped when the ball ends, regardless of its stop_events settings.

**`configure_logging`**(*logger*, *console_level='basic'*, *file_level='basic'*)
Configure the logging for the module this class is mixed into.

**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*)
Process this mode's configuration settings from a config dictionary.

**debug_log**(*msg*, **args*, ***kwargs*)
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*, **args*, ***kwargs*)
Log a message at the error level.

These messages will always be shown in the console and the log file.

**static get_config_spec**()
Return config spec for mode_settings.

**info_log**(*msg*, **args*, ***kwargs*)
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().

**mode_init**()
User-overrideable method which will be called when this mode initializes as part of the MPF boot process.

**mode_start**(***kwargs*)
User-overrideable method which will be called whenever this mode starts (i.e. whenever it becomes active).

**mode_stop**(***kwargs*)
User-overrideable method which will be called whenever this mode stops.

**player = None**
Reference to the current player object.

**restart_on_next_ball = None**
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*)
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=None*, ***kwargs*)
Stop this mode.

**Parameters** ****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.

**warning_log**(*msg*, **args*, ***kwargs*)
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. One instance of this class is created for each player.

The Game class maintains a "player" attribute which always points to the current player. You can access this via game.player. (Or self.machine.game.player).

This class is responsible for tracking per-player variables. There are several ways they can be used:

player.ball = 0 (sets the player's 'ball' value to 0) print player.ball (prints the value of the player's 'ball' value)

If the value of a variable is requested but that variable doesn't exist, that variable will automatically be created (and returned) with a value of 0.

Every time a player variable is changed, an MPF is posted with the name "player_<name>". That event will have three parameters posted along with it:

- value (the new value)
- prev_value (the old value before it was updated)
- change (the change in the value)

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 paramter will be True if the new value is different and False if the value didn't change.

Some examples:

player.score = 0

Event posted: 'player_score' with Args: value=0, change=0, prev_value=0

player.score += 500

Event posted: 'player_score' with Args: value=500, change=500, prev_value=0

player.score = 1200

Event posted: '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.

**`is_player_var`**(*var_name*)

Check if player var exists.

**`monitor_enabled = False`**

Class attribute which specifies whether any monitors have been registered to track player variable changes.

### RGBColor

**class** `mpf.core.rgb_color.`**`RGBColor`**(*color=None*, ***kwargs*)

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*, *color*)

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*, *default=None*)

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*, *default=(0, 0, 0)*)
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**()
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*)
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*, *default=(0, 0, 0)*)
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

**class** mpf.core.timer.**Timer**(*machine*, *mode*, *name*, *config*)
Bases: *mpf.core.logging.LogMixin*

Parent class for a mode timer.

**Parameters**

- **machine** – The main MPF MachineController object.
- **mode** – The parent mode object that this timer belongs to.
- **name** – The string name of this timer.
- **config** – A Python dictionary which contains the configuration settings for this timer.

### Methods & Attributes

The Timers has the following methods & attributes available. Note that methods & attributes inherited from the base class 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.

**configure_logging**(*logger*, *console_level='basic'*, *file_level='basic'*)
Configure the logging for the module this class is mixed into.

**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*, **args*, ***kwargs*)
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*, **args*, ***kwargs*)
Log a message at the error level.

These messages will always be shown in the console and the log file.

**`info_log`**(*msg*, **args*, ***kwargs*)
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().

**`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.

**`kill`**()
Stop this timer and also removes all the control events.

**`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.

**`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.

**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.

**warning_log**(*msg*, **args*, ***kwargs*)
Log a message at the warning level.

These messages will always be shown in the console and the log file.

## 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: [<class 'asyncio.futures.Future'>], loop, timeout=None*)
Return first future.

**static bin_str_to_hex_str**(*source_int_str*, *num_chars*)
Convert binary string to hex string.

**static cancel_futures**(*futures: [<class '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*)
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: [<class '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: []*) → [<class 'str'>]
Return a list of names from a list of switch objects.

**static hex_string_to_int**(*inputstring, maxvalue=255*)
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*)
Convert an int from 0-255 to a one-byte (2 chars) hex string, with uppercase characters.

**static is_hex_string**(*string*)
Return true if string is hex.

**static is_power2**(*num*)
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*, *num_chars=2*)
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 pwm32_to_hex_string**(*source_int*)
Convert a PWM32 value to hex.

**static pwm32_to_int**(*source_int*)
Convert a PWM32 value to int.

**static pwm8_to_hex_string**(*source_int*)
Convert an int to a PWM8 string.

**static pwm8_to_int**(*source_int*)
Convert a PWM8 value to int.

**static pwm8_to_on_off**(*source_int*)
Convert a PWM8 value to on/off times.

**static race**(*futures: {<class 'asyncio.futures.Future'>: <class '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*)
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*)
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*)

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*)

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*)

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*)

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*)

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*, *console_level='basic'*, *file_level='basic'*)

Configure the logging for the module this class is mixed into.

**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*, **args*, ***kwargs*)
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*, **args*, ***kwargs*)
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.

**info_log**(*msg*, **args*, ***kwargs*)
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().

**remove_key**(*key*)
Remove key by name.

**save_all**(*data=None*, *delay_secs=0*)
Write this DataManager's data to the disk.

**Parameters**

- **data** – An optional dict() of the data you want to write. If None then it will write the data as it exists in its own data attribute.
- **delay_secs** – Optional integer value of the amount of time you want to wait before the disk write occurs. Useful for writes that occur when MPF is busy, so you can delay them by a few seconds so they don't slow down MPF. Default is 0.

**save_key**(*key*, *value*, *delay_secs=0*)
Update an individual key and then write the entire dictionary to disk.

**Parameters**

- **key** – String name of the key to add/update.
- **value** – Value of the key
- **delay_secs** – Optional number of seconds to wait before writing the data to disk. Default is 0.

**warning_log**(*msg*, **args*, ***kwargs*)
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*)
Bases: `mpf.core.mpf_controller.MpfController`

Handles delays for one object.

### 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*, *callback*, *name=None*, ***kwargs*)
Add a delay.

**Parameters**

- **ms** – Int of the number of milliseconds you want this delay to be for. Note that the resolution of this time is based on your machine's tick rate. The callback will be called on the first machine tick *after* the delay time has expired. For example, if you have a machine tick rate of 30Hz, that's 33.33ms per tick. So if you set a delay for 40ms, the actual delay will be 66.66ms since that's the next tick time after the delay ends.
- **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 of the delay which you can use to remove it later.

**add_if_doesnt_exist**(*ms*, *callback*, *name*, ***kwargs*)
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. Note that the resolution of this time is based on your machine's tick rate. The callback will be called on the first machine tick *after* the delay time has expired. For example, if you have a machine tick rate of 30Hz, that's 33.33ms per tick. So if you set a delay for 40ms, the actual delay will be 66.66ms since that's the next tick time after the delay ends.
- **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*)
Check to see if a delay exists.

**Parameters delay** – A string of the delay you're checking for.

Returns: The delay object if it exists, or None if not.

**clear**()
Remove (clear) all the delays associated with this DelayManager.

**configure_logging**(*logger*, *console_level='basic'*, *file_level='basic'*)
Configure the logging for the module this class is mixed into.

**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*, **args*, ***kwargs*)
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*, **args*, ***kwargs*)
Log a message at the error level.

These messages will always be shown in the console and the log file.

**info_log**(*msg*, **args*, ***kwargs*)
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().

**remove**(*name*)
Remove a delay by name.

I.e. prevents the callback from being fired 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*, *callback*, *name*, ***kwargs*)
Reset a delay, first deleting the old one (if it exists) and then adding new delay with the new settings.

**Parameters** **as add()** (*same*) –

**run_now**(*name*)
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*, **args*, ***kwargs*)
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*)
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*)
Add a delay manager to the list.

## 7.4 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 700 unit tests for MPF and MPF-MC, each which test multiple aspects of the codebase.)

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 game.

### 7.4.1 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
```

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 501. 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
```

(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! :)

### 7.4.2 Writing Unit Tests for MPF

todo

### 7.4.3 Writing Unit Tests for Your Game

It's possible to create unit tests which test the actual functionality of your MPF game. 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 and hitting a sequence of switches, then you can check to make sure the 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.

Here's how you can create a basic unit test for your machine.

If you want to see a real example, check out the tests from Gabe Knuth's *Brooks 'n Dunn* machine:

https://github.com/GabeKnuth/BnD/blob/master/tests/test_bnd.py

#### 1. Add a tests folder to your machine folder

todo

### 7.4.4 Fuzz Testing

todo

## 7.5 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.

### 7.5.1 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".

### 7.5.2 Writing Plugins for MPF

todo

### 7.5.3 Developing your own hardware interface for MPF

todo

## 7.6 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.

### 7.6.1 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.

### 7.6.2 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.

## 7.6.3 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.

## 7.6.4 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

## 7.7 Method & Class Index

# Index