

---

# **Mission Pinball Framework API Reference**

***Release 0.20.0-dev69***

**Brian Madden, Gabe Knuth**

September 14, 2015



## CONTENTS

<b>1</b>	<b>Subpackages</b>	<b>1</b>
1.1	mpf.devices package . . . . .	1
1.2	mpf.game package . . . . .	20
1.3	mpf.media_controller package . . . . .	20
1.4	mpf.platform package . . . . .	32
1.5	mpf.plugins package . . . . .	48
1.6	mpf.system package . . . . .	52
<b>2</b>	<b>Module contents</b>	<b>87</b>
	<b>Python Module Index</b>	<b>89</b>
	<b>Index</b>	<b>91</b>



## SUBPACKAGES

### 1.1 mpf.devices package

#### 1.1.1 Submodules

##### **mpf.devices.autofire module**

Contains the base class for autofire coil devices.

```
class mpf.devices.autofire.AutofireCoil(machine, name, config, collection=None, validate=True)
```

Bases: mpf.system.device.Device

Base class for coils in the pinball machine which should fire automatically based on switch activity 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.

**class\_label** = ‘autofire’

**collection** = ‘autofires’

**config\_section** = ‘autofire\_coils’

**disable** (*\*\*kwargs*)

Disables the autofire coil rule.

**enable** (*\*\*kwargs*)

Enables the autofire coil rule.

**validate** ()

Autofire rules only work if the switch is on the same platform as the coil.

In the future we may expand this to support other rules various platform vendors might have.

##### **mpf.devices.ball\_device module**

Contains the base class for ball devices.

```
class mpf.devices.ball_device.BallDevice(machine, name, config, collection=None, validate=True)
```

Bases: mpf.system.device.Device

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.

**class\_label** = 'ball\_device'

**collection** = 'ball\_devices'

**config\_section** = 'ball\_devices'

**configure\_eject\_targets** (*config=None*)

**count\_balls** (*stealth=False, \*\*kwargs*)

Counts the balls in the device and processes any new balls that came in or balls that have gone out.

#### Parameters

- **stealth** – Boolean value that controls whether any events will be posted based on any ball count change info. If True, results will not be posted. If False, they will. Default is False.
- **\*\*kwargs** – Catches unexpected args since this method is used as an event handler.

**eject** (*balls=1, target=None, timeout=None, get\_ball=False*)

Ejects one or more balls from the device.

#### Parameters

- **balls** – Integer of the number of balls to eject. Default is 1.
- **target** – Optional target that should receive the ejected ball(s), either a string name of the ball device or a ball device object. Default is None which means this device will eject this ball to the first entry in the eject\_targets list.
- **timeout** – How long (in ms) to wait for the ball to make it into the target device after the ball is ejected. A value of None means the default timeout from the config file will be used. A value of 0 means there is no timeout.
- **get\_ball** – Boolean as to whether this device should attempt to get a ball to eject if it doesn't have one. Default is False.

Note that if this device's 'balls\_per\_eject' configuration is more than 1, then it will eject the nearest number of balls it can.

**eject\_all** (*target=None*)

Ejects all the balls from this device

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

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

**eject\_failed** (*retry=True, force\_retry=False*)

Marks the current eject in progress as 'failed.'

Note this is not typically a method that would be called manually. It's called automatically based on ejects timing out or balls falling back into devices while they're in the process of ejecting. But you can call it manually if you want to if you have some other way of knowing that the eject failed that the system can't figure out on it's own.

#### Parameters

- **retry** – Boolean as to whether this eject should be retried. If True, the ball device will retry the eject again as long as the 'max\_eject\_attempts' has not been exceeded. Default is True.

- **force\_retry** – Boolean that forces a retry even if the ‘max\_eject\_attempts’ has been exceeded. Default is False.

**get\_additional\_ball\_capacity()**

Returns an integer value of the number of balls this device can receive. A return value of 0 means that this device is full and/or that it's not able to receive any balls at this time due to a current eject\_in\_progress.

**get\_status(request=None)**

Returns a dictionary of current status of this ball device.

**Parameters request** – A string of what status item you'd like to request. Default will return all status items. Options include: \* balls \* eject\_in\_progress\_target \* eject\_queue

**Returns**

- balls
- eject\_in\_progress\_target
- eject\_queue

**Return type** A dictionary with the following keys

**hold(\*\*kwargs)****is\_full()**

Checks to see if this device is full, meaning it is holding either the max number of balls it can hold, or it's holding all the known balls in the machine.

Returns: True or False

**is\_playfield()**

Returns True if this ball device is a Playfield-type device, False if it's a regular ball device.

**num\_balls\_ejectable**

How many balls are in this device that could be ejected.

**request\_ball(balls=1)**

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.

Note that a device will never request more balls than it can hold. Also, only devices that are fed by other ball devices (or a combination of ball devices and diverters) can make this request. e.g. if this device is fed from the playfield, then this request won't work.

**status\_dump()**

Dumps the full current status of the device to the log.

**stop()**

Stops all activity in this device.

Cancels all pending eject requests. Cancels eject confirmation checks.

## mpf.devices.diverter module

Parent contains the base class for diverter devices.

**class** mpf.devices.diverter.**Diverter**(machine, name, config, collection=None, validate=True)

Bases: mpf.system.device.Device

Represents a diverter in a pinball machine.

Args: Same as the Device parent class.

**activate()**

Physically activates this diverter's coil.

**class\_label = 'diverter'**

**collection = 'diverters'**

**config\_section = 'diverters'**

**deactivate()**

Deactivates 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)**

Disables 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.

**disable\_held\_coil()**

Physically disables the coil holding this diverter open.

**disable\_hw\_switch()**

Removes the hardware rule to disable the hardware activation switch for this diverter.

**enable(auto=False, activations=-1, \*\*kwargs)**

Enables 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.
- **activations** – Integer of how many times you'd like this diverter to activate before it will automatically disable itself. Default is -1 which is unlimited.

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.

**enable\_hw\_switches()**

Enables the hardware switch rule which causes this diverter to activate when the switch is hit.

This is typically used for diverters on loops and ramps where you don't want the diverter to physically activate until the ramp entry switch is activated.

If this diverter is configured with a `activation_time`, this method will also set switch handlers which will set a delay to deactivate the diverter once the activation `activation_time` expires.

If this diverter is configured with a deactivation switch, this method will set up the switch handlers to deactivate the diverter when the deactivation switch is activated.



**schedule\_deactivation** (*time=None*)

Schedules a delay to deactivate this diverter.

**Parameters** **time** – The MPF string time of how long you’d like the delay before deactivating the diverter. Default is None which means it uses the ‘activation\_time’ setting configured for this diverter. If there is no ‘activation\_time’ setting and no delay is passed, it will disable the diverter immediately.

## mpf.devices.driver module

Contains the Driver parent class.

**class** mpf.devices.driver.**Driver** (*machine, name, config, collection=None, validate=True*)

Bases: mpf.system.device.Device

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

**class\_label** = ‘coil’

**collection** = ‘coils’

**config\_section** = ‘coils’

**disable** ()

Disables this driver

**enable** ()

Enables 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

**pulse** (*milliseconds=None, power=1.0*)

Pulses 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 only used if milliseconds is not specified.)

**pwm** (*on\_ms, off\_ms, orig\_on\_ms=0*)

Quickly turns this driver on and off to have the effect of holding this driver ‘on’ without burning up the coil.

### Parameters

- **on\_ms** – Integer of how long this driver is on in the ‘on’ portion.
- **off\_ms** – Integer of how long this driver is off in the ‘off’ portion.
- **orig\_on\_ms** – Integer of how long this driver should be held on initially before the on/off pulsing starts.

For example, to rapidly pulse a driver 1ms and then off 4ms, pass the values on\_ms=1, off\_ms=4.

**timed\_pwm** (*on\_ms, off\_ms, runtime\_ms*)

Quickly pulses a driver on/off for a specified number of milliseconds.

#### Parameters

- **on\_ms** – Integer of how long this driver is on in the ‘on’ portion.
- **off\_ms** – Integer of how long this driver is off in the ‘off’ portion.
- **runtime\_ms** – Integer of the total number of milliseconds this driver should pulse on and off for.

## mpf.devices.drop\_target module

Contains the base classes for drop targets and drop target banks.

**class** mpf.devices.drop\_target.**DropTarget** (*machine, name, config, collection=None, validate=True*)

Bases: mpf.system.device.Device

Represents a single drop target in a pinball machine.

Args: Same as the *Target* parent class

**add\_to\_bank** (*bank*)

Adds this drop target to a drop target bank, which 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.

**class\_label** = ‘drop\_target’

**collection** = ‘drop\_targets’

**config\_section** = ‘drop\_targets’

**knockdown** (*\*\*kwargs*)

Pulses the knockdown coil to knock down this drop target.

**reset** (*\*\*kwargs*)

Resets 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.

**class** mpf.devices.drop\_target.**DropTargetBank** (*machine, name, config, collection=None, validate=True*)

Bases: mpf.system.device.Device

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

```

class_label = 'drop_target_bank'
collection = 'drop_target_banks'
config_section = 'drop_target_banks'

```

```
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)
```

Resets 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.)

## mpf.devices.flasher module

Contains the Flasher parent class.

```
class mpf.devices.flasher.Flasher(machine, name, config, collection=None, validate=True)
```

Bases: `mpf.system.device.Device`

Generic class that holds flasher objects.

```
class_label = 'flasher'
```

```
collection = 'flashers'
```

```
config_section = 'flashers'
```

```
flash(milliseconds=None)
```

Flashes 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 system- wide `default_flash_ms` settings. (Current default is 50ms.)

## mpf.devices.flipper module

Contains the base class for flippers.

```
class mpf.devices.flipper.Flipper(machine, name, config, collection=None, validate=True)
```

Bases: `mpf.system.device.Device`

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

Contains several methods for actions that can be performed on this flipper, like `enable()` (page 8), `disable()` (page 8), etc.

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

More details: <http://missionpinball.com/docs/devices/flippers/>

### Parameters

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

- **config** – A dictionary that holds the configuration values which specify how this flipper should be configured. If this is None, it will use the system config settings that were read in from the config files when the machine was reset.
- **collection** – A reference to the collection list this device will be added
- **to.** –

**class\_label** = 'flipper'

**collection** = 'flippers'

**config\_section** = 'flippers'

**disable** (\*\*kwargs)

Disables 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)

Enables 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 F. Disable Main Button inactive G. Disable Hold Button inactive

One coil, using EOS switch Rule Type Coil Switch Action A. Enable Main Button active H. PWM Main EOS active F. Disable Main Button inactive

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

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

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

**enable\_no\_hold** ()

Enables the flippers in 'no hold' mode.

No Hold is a novelty mode where the flippers to not stay up even when the buttons are held in.

This mode is not yet implemented.

**enable\_partial\_power** (percent)

Enables flippers which operated at less than full power.

This is a novelty mode, like "weak flippers" from the Wizard of Oz.

**Parameters percent** – A floating point value between 0 and 1.0 which represents the percentage of power the flippers will be enabled at.

This mode is not yet implemented.

**classmethod invert ()**

Enables inverted flippers.

Inverted flippers is a novelty mode where the left flipper button controls the right flippers and vice-versa.

This mode is not yet implemented.

**sw\_flip ()**

Activates 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 ()**

Deactivates the flipper via software as if the flipper button was released. See the documentation for `sw_flip()` for details.

## mpf.devices.gi module

Contains the GI (General Illumination) parent classes.

**class** `mpf.devices.gi.GI` (*machine, name, config, collection=None, validate=True*)

Bases: `mpf.system.device.Device`

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.

**add\_handler** (*callback*)

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

**class\_label** = 'gi'

**collection** = 'gi'

**config\_section** = 'gis'

**disable** (*\*\*kwargs*)

Disables this GI string.

**enable** (*brightness=255, fade\_ms=0, start\_brightness=None, \*\*kwargs*)

Enables this GI string.

### Parameters

- **brightness** – Int from 0-255 of how bright you want this to be. 255 is on. 0 is off. 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.
- **start\_brightness** – Starting brightness level for a fade.

**remove\_handler** (*callback=None*)

Removes a handler from the list of registered handlers.

## mpf.devices.led module

Contains the LED parent classes.

**class** `mpf.devices.led.LED` (*machine, name, config, collection=None, validate=True*)

Bases: `mpf.system.device.Device`

Represents an light connected to an new-style interface board. Typically this is an LED.

DirectLEDs can have any number of elements. Typically they're either single element (single color), or three element (RGB), though dual element (red/green) and quad-element (RGB + UV) also exist and can be used.

**class\_label** = 'led'

**collection** = 'leds'

**color** (*color, fade\_ms=None, brightness\_compensation=True, priority=0, cache=True, force=False, blend=False*)

Sets this LED to the color passed.

### Parameters

- **color** – a list of integers which represent the red, green, and blue values the LED will be set to. If this list is fewer than three items, it assumes zeros for the rest.
- **fade\_ms** – Integer value of how long the LED should fade from its current color to the color you're passing it here.
- **brightness\_compensation** – Boolean value which controls whether this LED will be light using the current brightness compensation. Default is True.
- **priority** – Arbitrary integer value of the priority of this request. If the incoming priority is lower than the current priority, this incoming color request will have no effect. Default is 0.
- **cache** – Boolean which controls whether this new color command will update the LED's cache. Default is True.
- **force** – Boolean which will force this new color command to be applied to the LED, regardless of the incoming or current priority. Default is True.
- **blend** – Not yet implemented.

**compensate** (*color*)

Applies the current brightness compensation values to the passed color.

**Parameters** **color** – a 3-item color list of ints

**Returns** The brightness-compensated 3-item color list of ints

**config\_section** = 'leds'

**classmethod** **device\_class\_init** (*machine*)

**disable** (*fade\_ms=0, priority=0, cache=True, force=False*)

Disables an LED, including all elements of a multi-color LED.

**get\_state** ()

Returns the current state of this LED

**off** (*fade\_ms=0, priority=0, cache=True, force=False*)

**on** (*brightness=255, fade\_ms=0, start\_brightness=None, priority=0, cache=True, force=False*)

**restore** ()

Sets this LED to the cached state.

**set\_brightness\_compensation** (*value*)

Sets the brightness compensation for this LED.

**Parameters** **value** – Str or list (of 1-to-3 items) of the new brightness compensation value to set. List items are floats. 1.0 is standard full brightness. 0.0 is off. 2.0 is 200% brightness (which only comes into play if the LED is not at full brightness). If the value is a string, it's converted to a list, broken by commas.

The brightness compensation list is three items long, one for each RGB element. If the LED has less than three elements, additional values are ignored.

If the value list is only one item, that value is used for all three elements.

If the value list is two items, a value of 1.0 is used for the third item.

## mpf.devices.matrix\_light module

Contains the MatrixLight parent classes.

**class** mpf.devices.matrix\_light.**MatrixLight** (*machine, name, config, collection=None, validate=True*)

Bases: mpf.system.device.Device

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.

**add\_handler** (*callback*)

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

**class\_label** = 'light'

**collection** = 'lights'

**config\_section** = 'matrix\_lights'

**off** (*fade\_ms=0, priority=0, cache=True, force=False*)

**on** (*brightness=255, fade\_ms=0, start\_brightness=None, priority=0, cache=True, force=False*)

Turns on this matrix light.

### Parameters

- **brightness** – How bright this light should be, as an int between 0 and 255. 0 is off. 255 is full on. Note that intermediary values are not yet implemented, so 0 is off, anything from 1-255 is full on.
- **fade\_ms** – Not yet implemented
- **start\_brightness** – Not yet implemented.
- **priority** – The priority of the incoming request. If this priority is lower than the current cached priority, this on command will have no effect. (Unless force=True)
- **cache** – Boolean as to whether this light should cache these new settings. This cache can be used for the light to “go back” to it's previous state. Default is True.
- **force** – Whether the light should be forced to go to the new state, regardless of the incoming and current priority. Default is False.

**remove\_handler** (*callback=None*)

Removes a handler from the list of registered handlers.

**restore()**

Restores the light state from cache.

### **mpf.devices.new\_device\_template module**

Template file for a new device driver.

**class** mpf.devices.new\_device\_template.**YourNewDevice**(*machine, name, config, collection=None, validate=True*)

Bases: mpf.system.device.Device

**class\_label** = 'your\_new\_device'

**collection** = 'your\_new\_devices'

**config\_section** = 'your\_new\_devices'

**classmethod device\_class\_init**(*machine*)

This @classmethod is optional, but is called automatically before individual devices based on this device class are created. You can use it for any system-wide settings, configurations, or objects that you might need for these types of devices outside of the individual devices themselves.

For example, led.py uses this to make sure the global fade\_ms default fade time is a float. The EM score reels devices use this to set up the score controller that has to exist to manage them.

You can safely delete this method if your device doesn't need it. (Most don't need it.)

### **mpf.devices.playfield module**

Contains the Playfield device class which represents the actual playfield in a pinball machine.

**class** mpf.devices.playfield.**Playfield**(*machine, name, config, collection=None, validate=True*)

Bases: mpf.devices.ball\_device.BallDevice (page 1)

**add\_ball**(*balls=1, source\_name=None, source\_device=None, trigger\_event=None*)

Adds live ball(s) to the playfield.

#### **Parameters**

- **balls** – Integer of the number of balls you'd like to add.
- **source\_name** – Optional string name of the ball device you'd like to add the ball(s) from.
- **source\_device** – Optional ball device object you'd like to add the ball(s) from.
- **trigger\_event** – The optional name of an event that MPF will wait for before adding the ball into play. Typically used with player- controlled eject tag events. If None, the ball will be added immediately.

**Returns** True if it's able to process the add\_ball() request, False if it cannot.

Both source\_name and source\_device args are included to give you two options for specifying the source of the ball(s) to be added. You don't need to supply both. (it's an "either/or" thing.) Both of these args are 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 ball 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, or if the player tilts then there are still balls on the playfield but no balls in play.

**ball\_found** (*num=1*)

Used when a previously missing ball is found. Updates the balls known and balls missing variables.

**Parameters** **num** (*int*) – Specifies how many balls have been found. Default is 1.

**ball\_lost** ()

Mark a ball as lost

**ball\_search\_begin** (*force=False*)

Begin the ball search process

**ball\_search\_disable** ()

Disables ball search.

Note this is used to prevent a future ball search from happening (like when all balls become contained). This method is not used to cancel an existing ball search. (Use *ball\_search\_end* for that.)

**ball\_search\_end** ()

End the ball search, either because we found the ball or are giving up.

**ball\_search\_failed** ()

Ball Search did not find the ball.

**ball\_search\_schedule** (*secs=None, force=False*)

Schedules a ball search to start. By default it will schedule it based on the time configured in the machine configuration files.

If a ball search is already scheduled, this method will reset that schedule to the new time passed.

**Parameters**

- **secs** – Schedules the ball search that many secs from now.
- **force** – Boolean to force a ball search. Set True to force a ball search. Otherwise it will only schedule it if *self.flag\_no\_ball\_search* is False. Default is False

**balls**

**class\_label** = 'playfield'

**collection** = 'playfields'

**config\_section** = 'playfields'

**count\_balls** (*\*\*kwargs*)

Used to count the number of balls that are contained in a ball device. Since this is the playfield device, this method always returns zero.

Returns: 0

**eject** (*\*args, \*\*kwargs*)

**eject\_all** (*\*args, \*\*kwargs*)

**get\_additional\_ball\_capacity** ()

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

**ok\_to\_confirm\_ball\_via\_playfield\_switch()**

Used to check whether it's ok for a ball device which ejects to the playfield to confirm its eject via a playfield switch being hit.

Returns: True or False

Right now this is simple. If there are no playfield balls, then any playfield switch hit is assumed to be from the newly-ejected ball. If there are other balls on the playfield, then we can't use this confirmation method since we don't know whether a playfield switch hit is from the newly-ejected ball(s) or a current previously-live playfield ball.

**player\_eject\_request** (*balls, device*)

A player has hit a switch tagged with the `player_eject_request_tag`.

**Parameters**

- **balls** – Integer of the number of balls that will be ejected.
- **device** – The ball device object that will eject the ball(s).

**playfield\_switch\_hit** ()

A switch tagged with '<this playfield name>\_active' was just hit, indicating that there is at least one ball on the playfield.

**remove\_player\_controlled\_eject** ()

Removed the player-controlled eject so a player hitting a switch no longer calls the device(s) to eject a ball.

**setup\_player\_controlled\_eject** (*balls, device, trigger\_event*)

Used to set up an eject from a ball device which will eject a ball to the playfield.

**Parameters**

- **balls** – Integer of the number of balls this device should eject.
- **device** – The ball device object that will eject the ball(s) when a switch with the player-controlled eject tag is hit.
- **trigger\_event** – The name of the MPF event that will trigger the eject.

When this method is called, MPF will set up an event handler to look for the `trigger_event`.

**mpf.devices.playfield\_transfer module**

Transfer a ball between two playfields. E.g. lower to upper playfield via a ramp

```
class mpf.devices.playfield_transfer.PlayfieldTransfer (machine, name, config, collection=None, validate=True)
```

```
    Bases: mpf.system.device.Device
```

```
    class_label = 'playfield_transfer'
```

```
    collection = 'playfield_transfer'
```

```
    config_section = 'playfield_transfer'
```

**mpf.devices.score\_reel module**

Contains the base classes for mechanical EM-style score reels.

```
class mpf.devices.score_reel.ScoreReel (machine, name, config, collection=None, validate=True)
```

Bases: mpf.system.device.Device

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).

**advance** (*direction=None*)

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

This method also schedules delays to post the following events:

*reel\_<name>\_pulse\_done*: When the coil is done pulsing *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

**Parameters** *direction* (*int, optional*) – If *direction* is 1, advances the reel to the next higher position. If *direction* is -1, advances the reel down one position (if the reel has a decrement coil). If *direction* is not passed, this method will compare the reel's *\_destination\_index* to its *assumed\_value* and will advance it in the direction it needs to go if those values do not match.

**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*)

Checks 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*.

```
class_label = 'score_reel'
```

```
collection = 'score_reels'
```

```
config_section = 'score_reels'
```

**logical\_to\_physical** (*value*)

Converts a logical reel displayed value to what the physical switch value should be.

For example, if a reel has switches for the 0 and 9 values, then an input of 0 will return 0 (since that's what the physical value should be for that logical value). In that case it will return 9 for an input of 9, but it will return -999 for any input value of 1 through 8 since there are no switches for those values.

Note this method does not perform any physical or logical check against the reel's actual position, rather, it's only used to indicate what hardware switch value should be expected for the display value passed.

**Parameters** *value* (*int*) – The value you want to check.

**Returns** The physical switch value, which is same as the input value if there's a switch there, or -999 if not.

**pulse\_ms**

Returns an integer representing the number of milliseconds the coil will pulse for.

This method is used by the jump and step advances so they know when a reel's coil is done firing so they can fire the next reel in the group.

**Parameters**

- **direction** (*int*, *optional*) – Lets you specify which coil you want to
- **the time for. Default is 1** (*get*) –
- **down**). –

**Returns:** Integer of the coil pulse time. If there is no coil for the direction you specify, returns 0.

**set\_destination\_value** (*direction=1*)

Returns the integer value of the destination this reel is moving to.

**Parameters**

- **direction** (*int*, *optional*) – The direction of the reel movement this
- **should get the value for. Default is 1 which means of 'up'. (method)** –
- **can pass -1 the next lower value. (You)** –

**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.

**class** `mpf.devices.score_reel.ScoreReelController` (*machine*)

Bases: `object`

The overall controller that is in charge of and manages the score reels in a pinball machine.

The main thing this controller does is keep track of how many ScoreReelGroups there are in the machine and how many players there are, as well as maps the current player to the proper score reel.

This controller is also responsible for working around broken ScoreReelGroups and “stacking” and switching out players when there are multiple players per ScoreReelGroup.

**game\_starting** (*queue*, *game*)

Resets the score reels when a new game starts.

This is a queue event so it doesn't allow the game start to continue until it's done.

**Parameters**

- **queue** – A reference to the queue object for the game starting event.
- **game** – A reference to the main game object. This is ignored and only included because the `game_starting` event passes it.

#### **map\_new\_score\_reel\_group()**

Creates a mapping of a player to a score reel group.

#### **rotate\_player(\*\*kwargs)**

Called when a new player's turn starts.

The main purpose of this method is to map the current player to their `ScoreReelGroup` in the backbox. It will do this by comparing length of the list which holds those mappings (`player_to_score_reel_map`) to the length of the list of players. If the player list is longer that means we don't have a `ScoreReelGroup` for that player.

In that case it will check the tags of the `ScoreReelGroups` to see if one of them is tagged with `playerX` which corresponds to this player. If not then it will pick the next free one. If there are none free, then it will "double up" that player on an existing one which means the same Score Reels will be used for both players, and they will reset themselves automatically between players.

#### **score\_change(score, change)**

Called whenever the score changes and adds the score increase to the current active `ScoreReelGroup`.

This method is the handler for the score change event, so it's called automatically.

#### **Parameters**

- **score** – Integer value of the new score. This parameter is ignored, and included only because the score change event passes it.
- **change** – Integer value of the change to the score.

**class** `mpf.devices.score_reel.ScoreReelGroup(machine, name, config, collection=None, validate=True)`

Bases: `mpf.system.device.Device`

Represents a logical grouping of score reels in a pinball machine, where 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.system.device.Device`.

#### **add\_value(value, jump=False, target=None)**

Adds 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**

**assumed\_value\_list**

**chime** (*chime*)

**class\_label** = 'score\_reel\_group'

**collection** = 'score\_reel\_groups'

**config\_section** = 'score\_reel\_groups'

**classmethod device\_class\_init** (*machine*)

**get\_physical\_value\_list** ()

Queries all the reels in the group and builds a list of their actual current physical state, with 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** ()

Initialized 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 in `__init__()` because all the other reels have to be setup first.

**int\_to\_reel\_list** (*value*)

Converts 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 physical 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 integer 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*)

Tests 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*)

Lights up this ScoreReelGroup based on the 'light\_tag' in its config.

**reel\_list\_to\_int** (*reel\_list*)

Converts 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** **value** – The list containing the values for each score reel position.

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

**set\_rollover\_reels** ()

Calls 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*)

Resets 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 are 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** ()

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

**unlight** (*relight\_on\_valid=False, \*\*kwargs*)

Turns off the lights for this ScoreReelGroup based on the 'light\_tag' in its config.

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

Called to 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.

## mpf.devices.switch module

Contains the Switch parent class.

**class** `mpf.devices.switch.Switch` (*machine, name, config, collection=None, validate=True*)

Bases: `mpf.system.device.Device`

A switch in a pinball machine.

**class\_label** = 'switch'

**collection** = 'switches'

```
config_section = 'switches'
```

`mpf.devices.target` module

### 1.1.2 Module contents

## 1.2 `mpf.game` package

### 1.2.1 Submodules

`mpf.game.attract` module

`mpf.game.game` module

`mpf.game.player` module

### 1.2.2 Module contents

## 1.3 `mpf.media_controller` package

### 1.3.1 Subpackages

`mpf.media_controller.core` package

#### Submodules

`mpf.media_controller.core.display` module

`mpf.media_controller.core.font_manager` module

`mpf.media_controller.core.keyboard` module   MC keyboard processor

**class** `mpf.media_controller.core.keyboard.Keyboard` (*mc*)

Bases: `object`

Base class which allows a computer keyboard to be used to simulate switch activity in a pinball machine. This is good for testing purposes when you aren't sitting at the actual machine.

The Keyboard class gets its settings from the Machine Configuration Files in the [keymap] section.

This module uses a Pygame window to capture the key events.

**Parameters** *mc* – The main media controller object.

**add\_key\_map** (*key*, *switch\_name=None*, *toggle\_key=False*, *invert=False*, *event\_dict=None*)

Adds an entry to the `key_map` which is used to see what to do when key events are received.

#### Parameters

- **key** – The built-up string of the key combination which optionally includes modifier keys.
- **switch\_name** – String name of the switch this key combination is tied to.



- **toggle\_key** – Boolean as to whether this key should be a toggle key. (i.e. push on / push off).
- **invert** – Boolean as to whether this key combination should be inverted. (Key down = switch inactive, key up = switch active.) Default is False.
- **event\_dict** – Dictionary of events with parameters that will be posted when this key combination is pressed. Default is None.

**get\_key\_press\_string** (*symbol, modifiers*)

Converts a Pygame key symbol with modifiers into the string format that MPF uses in its internal key map.

**Parameters**

- **symbol** – The Pygame symbol of the key.
- **modifiers** – The Pygame modifier value for any modifier keys that were active along with this key event.

Returns: A string in the proper format MPF uses.

**process\_key\_press** (*symbol, modifiers*)

Processes a key press (key down) event by setting the switch and/or posting the event to MPF.

**Parameters**

- **symbol** – The Pygame symbol of the key that was just pressed.
- **modifiers** – The Pygame modifier value for any modifier keys that were active along with this key event.

**process\_key\_release** (*symbol, modifiers*)

Processes a key release (key up) event by setting the switch and/or posting the event to MPF.

**Parameters**

- **symbol** – The Pygame symbol of the key that was just released.
- **modifiers** – The Pygame modifier value for any modifier keys that were active along with this key event.

**send\_switch** (*name, state*)

`mpf.media_controller.core.keyboard.preload_check (mc)`

**mpf.media\_controller.core.language module** Contains the parent class for MPF's Language module.

**class** `mpf.media_controller.core.language.Language` (*machine*)

Bases: `object`

MPF module which handles text, audio, and video replacement of objects for multi-language environments.

**Parameters** **machine** – The main machine object

**get\_language** ()

Returns the string name of the current language.

**get\_text** (*text, language*)

Returns a translated text string for a specific language string.

**Parameters**

- **text** – The text string you'd like to get the replacement for.
- **language** – The language you'd like to lookup for the replacement.

If the specific text string and language combination doesn't exist in the translation file, the original string is returned.

The string lookup is case-sensitive.

This method is similar to `text()`, except this method doesn't strip out the parentheses. (i.e. it's just used to look up what's "inside" the parentheses.)

**set\_language** (*language\_string*)

Sets the current language based on the string passed.

**Parameters language\_string** – The string name of the language you want to set the machine to.

Language strings can be whatever you want, based on how you define them in your config file. It can be an actual language, like English or French, or it can simply be alternate assets, like "Kid-Friendly" versus "Mature."

This language change is instant, and you can safely call it often. Change languages for each player in the same game, or even in the middle of a ball!

**text** (*text*)

Translates a text string (or part of a text string) based on the current language setting.

**Parameters text** – The string of text you want to translate.

Returns: A translated string.

The incoming text string is searched for text within parentheses, and each of those segments is looked up for replacement. You can wrap the entire string in parentheses, or just part of it, or multiple parts.

A new, translated string is returned with the parentheses removed. If a translation is not found in the current language's translation strings, the original text is returned.

The string lookup is case-sensitive since different languages have different rules around casing.

It is not possible to display text with parentheses in it since this method will remove them. If this is something you need, contact us and we can add that feature.

## **mpf.media\_controller.core.media\_controller module**

## **mpf.media\_controller.core.modes module**

**mpf.media\_controller.core.show\_controller module** Manages the show effects in a pinball machine.

**class** `mpf.media_controller.core.show_controller.Show` (*machine, config, file\_name, asset\_manager, actions=None*)

Bases: `mpf.system.assets.Asset` (page 52)

**add\_loaded\_callback** (*loaded\_callback, \*\*kwargs*)

**advance** ()

**change\_speed** (*tocks\_per\_sec=1*)

Changes the playback speed of a running Show.

**Parameters tocks\_per\_sec** – The new tocks\_per\_second play rate.

If you want to change the playback speed by a percentage, you can access the current `tocks_per_second` rate via `Show's tocks_per_second` variable. So if you want to double the playback speed of your show, you could do something like:

```
self.your_show.change_speed(self.your_show.tocks_per_second*2)
```

Note that you can't just update the show's `tocks_per_second` directly because we also need to update `self.secs_per_tock`.

**do\_load** (*callback*, *show\_actions=None*)

**load\_show\_from\_disk** ()

**play** (*repeat=False*, *priority=0*, *blend=False*, *hold=False*, *tocks\_per\_sec=30*, *start\_location=None*, *callback=None*, *num\_repeats=0*, *mode=None*)

Plays a Show. There are many parameters you can use here which affect how the show is played. This includes things like the playback speed, priority, whether this show blends with others, etc. These are all set when the show plays. (For example, you could have a Show file which lights a bunch of lights sequentially in a circle pattern, but you can have that circle “spin” as fast as you want depending on how you play the show.)

#### Parameters

- **repeat** – Boolean of whether the show repeats when it's done.
- **priority** – Integer value of the relative priority of this show. If there's ever a situation where multiple shows want to control the same item, the one with the higher priority will win. (“Higher” means a bigger number, so a show with priority 2 will override a priority 1.)
- **blend** – Boolean which controls whether this show “blends” with lower priority shows and scripts. For example, if this show turns a light off, but a lower priority show has that light set to blue, then the light will “show through” as blue while it's off here. If you don't want that behavior, set `blend` to be `False`. Then off here will be off for sure (unless there's a higher priority show or command that turns the light on). Note that not all item types blend. (You can't blend a coil or event, for example.)
- **hold** – Boolean which controls whether the lights or LEDs remain in their final show state when the show ends.
- **tocks\_per\_sec** – Integer of how fast your show runs (“Playback speed,” in other words.) Your Show files specify action times in terms of ‘tocks’, like “make this light red for 3 tocks, then off for 4 tocks, then a different light on for 6 tocks. When you play a show, you specify how many tocks per second you want it to play. Default is 30, but you might even want `tocks_per_sec` of only 1 or 2 if your show doesn't need to move than fast. Note this does not affect fade rates. So you can have `tocks_per_sec` of 1 but still have lights fade on and off at whatever rate you want. Also the term “tocks” was chosen so as not to confuse it with “ticks” which is used by the machine run loop.
- **start\_location** – Integer of which position in the show file the show should start in. Usually this is 0 but it's nice to start part way through. Also used for restarting shows that you paused.
- **callback** – A callback function that is invoked when the show is stopped.
- **num\_repeats** – Integer of how many times you want this show to repeat before stopping. A value of 0 means that it repeats indefinitely. Note this only works if you also have `repeat=True`.

**stop** (*reset=True*, *hold=None*)

Stops a Show.

Note you can also use this method to clear a stopped show's held lights and LEDs by passing `hold=False`.

**Parameters** **reset** – Boolean which controls whether the show will reset its current position back to zero. Default is `True`.

**class** `mpf.media_controller.core.show_controller.ShowController` (*machine*)

Bases: `object`

Manages all the shows in a pinball machine.

‘shows’ are coordinated display & event sequences. The ShowController handles priorities, restores, running and stopping Shows, etc. There should be only one per machine.

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

**add\_show\_player\_show** (*event, settings, mode=None*)

**play\_show** (*show, mode=None, repeat=False, priority=0, blend=False, hold=False, ticks\_per\_sec=30, start\_location=None, num\_repeats=0, \*\*kwargs*)

**process\_shows\_from\_config** (*config, mode=None, priority=0*)

**stop\_show** (*show, reset=True, hold=False, \*\*kwargs*)

**stop\_shows\_by\_key** (*key*)

**unload\_show\_player\_shows** (*removal\_tuple*)

**mpf.media\_controller.core.slide module** Contains the Slide parent class.

**class** `mpf.media_controller.core.slide.Slide` (*mpfdisplay, machine, name=None, priority=0, persist=False, removal\_key=None, expire\_ms=0, mode=None*)

Bases: `object`

Parent class for a Slide object.

MPF uses the concept of “slides” (think PowerPoint) as a collection of various display elements that should be shown on the display. There can be more than one slides defined at any given time, though only one is active at any given time. (Unless a transitioning is happening which transitions between an old slide and a new slide.

**Parameters**

- **mpfdisplay** – The display object this slide is for.
- **name** – String name of this slide.
- **persist** – Boolean as for whether this slide should be automatically destroyed once it’s not shown on the display anymore.
- **removal\_key** – A unique key that can identify this slide for its removal later.
- **expire\_ms** – How many ms this slide should live for. Default is 0 which means it will not automatically be removed.
- **mode** – A reference to the Mode which created this slide.

**name**

The string name of this slide.

**elements**

List of display elements that are active on this slide.

**surface**

Reference to the Pygame surface this slide uses.

**priority**

Integer of the relative priority of this slide. Lower priority slides won’t be shown if a higher priority slide is currently active. (A slide of the same priority as the active slide can displace it.) Note that ‘higher’ numbers equal ‘higher’ priority, i.e. Priority 2 is higher than Priority 1.

**active**

Boolean as to whether this slide is currently active. Active slides will constantly ensure that their display elements are refreshed, so this saves CPU cycles by not keeping non-active slides up-to-date.

**persist**

Boolean as to whether this slide should persist when it becomes non-active. A value of False means that this slide will be destroyed as soon as it's no longer active.

**width**

Width of this slide, in pixels.

**height**

Height of this slide, in pixels.

**depth**

Integer value of the color depth, either 8 or 24.

**palette**

The Pygame palette this slide uses. (8-bit only)

**removal\_key**

Unique identifier that can be used later to remove this slide.

**expire\_ms**

Integer of ms that will cause this slide to automatically remove itself. The timer doesn't start until the slide is shown.

**add\_element** (*element\_type*, *name=None*, *x=None*, *y=None*, *h\_pos=None*, *v\_pos=None*, *text\_variables=None*, *\*\*kwargs*)

Adds a display element to the slide.

**Parameters**

- **element\_type** – String name of the type of element you're adding. (i.e. 'Text', 'Image', 'Shape', etc.).
- **name** – Friendly name of the new element.
- **x** – 'x' position of the upper left corner of this element. This is either the 'x' position, or an offset in pixels, or an offset percentage. See the documentation for the `calc_position` method for details.
- **y** – 'y' position or offset, like x above.
- **h\_pos** – Relative horizontal position: left, center, or right.
- **v\_pos** – Relative vertical position: top, center, or bottom.
- **\*\*kwargs** – A list of key/value settings for the element you're adding.

**Returns**

**name:** String name of the element. **element:** The newly-created display element object.  
**layer:** Integer of the relative layer of this element on the slide.

**x:** x position of the upper left corner of this element on the slide.

**y:** y position of the upper left corner of this element on the slide.

**Return type** An element dictionary, which includes

**add\_ready\_callback** (*callback*, *\*\*kwargs*)

**blit\_8bit\_alpha** (*source\_surface, dest\_surface, x, y*)

Blits an 8-bit surface onto another using the DMD-style alpha values.

**Parameters**

- **source\_surface** – Source 8-bit pygame surface
- **dest\_surface** – Destination 8-bit Pygame surface the source surface will be blitted *to*.
- **x** – x position of the upper left corner of where the source surface will be blitted to on the destination surface.
- **y** – y position (goes with *x* above)

Note this blit is expensive, so it's only used when it's specifically called for.

**clear** ()

Removes all elements from the slide and resets the slide to all black.

**get\_subsurface** (*rect, layer=0*)

Returns a surface of the slide based on the rect passed, but only for the elements of the passed layer and lower

**Parameters**

- **rect** – A pygame Rect object which defines the rectangle that will be returned.
- **layer** – Optional layer which defines the highest layer element that should be included in the surface.

Returns: A Pygame surface.

**ready** ()

**refresh** (*force\_dirty=False*)

Refreshes the slide by clearing it, and updating all the display elements.

**Parameters** **force\_dirty** – Boolean which controls whether you want to force all the elements to be marked as dirty so they're regenerated.

**remove** ()

Removes the slide. If this slide is active, the next-highest priority slide will automatically be shown.

**remove\_element** (*name*)

Removes a display element from the slide.

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

**schedule\_removal** (*removal\_time=None*)

Schedules this slide to automatically be removed.

**Parameters** **removal\_time** – MPF time string of when this slide should be removed. If no time is specified, the slide's existing removal time is used. If the slide has no existing time, the slide will not be removed.

**show** ()

Shows this slide by making it active.

This is immediate. If you want a transition, use the `MPFDisplay.transition()` method.

This method will only show the slide if its priority is the same or higher than the existing slide.

**update** ()

Updates this slide by calling each display element's `update()` method, and blits the results if there's an update.

**mpf.media\_controller.core.slide\_builder module** Contains the parent classes for MPF's display SlideBuilder class.

**class** `mpf.media_controller.core.slide_builder.SlideBuilder` (*machine*)

Bases: `object`

Parent class for SlideBuilder objects which are things you configure via the machine config files that let you display text messages based on game events. You can use this to show game status, players, scores, etc. Any setting that is available via the text method of the display controller is available here, including positioning, fonts, size, delays, etc.

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

**build\_slide** (*settings, display=None, slide\_name=None, priority=None, mode=None, \*\*kwargs*)

Builds a slide from a SlideBuilder set of keyword arguments.

#### Parameters

- **settings** – Python dictionary of settings for this slide. This includes settings for the various Display Elements as well as any transition.
- **display** – String name of the display this slide is being built for.
- **slide\_name** – String name of the slide that's being built. If this slide exists, the elements here will be added to that slide. If it doesn't exist, a new slide will be created. If no slide name is passed, a new slide will be created and given a UUID4 name.
- **priority** – Integer of the priority of this slide.
- **mode** – A reference to the Mode instance that built this slide. Used to make sure that each mode keeps at least one active slide.
- **\*\*kwargs** – Catch all since this method is often registered as a callback for events which means there could be random event keyword argument pairs attached.

**Returns:** Slide object from the slide it built (whether or not it's showing now).

**preprocess\_settings** (*settings, base\_priority=0*)

Takes an unstructured list of slide\_player settings and processed them so they can be displayed.

#### Parameters

- **settings** – A list of dictionary of slide\_player settings for a slide.
- **base\_priority** – An integer that will be added to slide's priority from the config settings.

Returns: A python list with all the settings in the right places.

This method does a bunch of things, like making sure all the needed values are there, and moving certain things to the first and last elements when there are multiple elements used on one slide. (For example, if one of the elements wants to clear the slide, it has to happen first. If there's a transition, it has to happen last after the slide is built, etc.

The returned settings list can be safely called with the by `display()` with the `preprocessed=True` flag.

**process\_config** (*config, mode=None, priority=0*)

**unload\_slide\_events** (*removal\_tuple*)

**mpf.media\_controller.core.sound module** MPF plugin for sounds. Includes SoundController, Channel, Sound, Track, and StreamTrack parent classes.

**class** `mpf.media_controller.core.sound.Channel` (*machine, parent\_track, channel\_number*)  
Bases: `object`

Parent class that holds a Pygame sound channel. One or more of these are tied to an MPF Track.

#### Parameters

- **machine** – The main machine object.
- **parent\_track** – The MPF track object this channel belongs to.
- **channel\_number** – Integer number that is used to identify this channel.

**play** (*sound, \*\*settings*)

Plays a sound on this channel.

#### Parameters

- **sound** – The sound object to play.
- **\*\*settings** – Additional settings for this sound's playback.

**sound\_is\_done** ()

Indicates that the sound that was playing on this channel is now done.

This is the callback method that's automatically called by Pygame. It will check the queue and automatically play any queued sounds.

**class** `mpf.media_controller.core.sound.Sound` (*machine, config, file\_name, asset\_manager*)  
Bases: `mpf.system.assets.Asset` (page 52)

**do\_load** (*callback*)

**play** (*loops=0, priority=0, fade\_in=0, volume=1, \*\*kwargs*)

Plays this sound.

#### Parameters

- **loops** – Integer of how many times you'd like this sound to repeat. A value of -1 means it will loop forever.
- **priority** – The relative priority of this sound which controls what happens if the track this sound is playing on is playing the max simultaneous sounds.
- **fade\_in** – MPF time string for how long this sound should fade in when it starts.
- **volume** – Volume for this sound as a float between 0.0 and 1.0. Zero is mute, 1 is full volume, anything in between is in between.
- **\*\*kwargs** – Catch all since this method might be used as an event callback which could include random kwargs.

**stop** (*fade\_out=0, reset=True, \*\*kwargs*)

Stops this sound playing.

#### Parameters

- **fade\_out** – MPF time string for how long this sound will fade out as it stops.
- **reset** – Boolean for whether this sound should reset its playback position to the beginning. Default is True.
- **\*\*kwargs** – Catch all since this method might be used as an event callback which could include random kwargs.



**class** `mpf.media_controller.core.sound.SoundController` (*machine*)

Bases: `object`

Parent class for the sound controller which is responsible for all audio, sounds, and music in the machine. There is only one of these per machine.

**Parameters** *machine* – The main machine controller object.

**create\_track** (*name, config=None*)

Creates a new MPF track add registers in the central track list.

**Parameters**

- **name** – String name of this track used for identifying where sounds are played.
- **config** – Config dictionary for this track.

Note: “Tracks” in MPF are like channels.. you might have a “music” track, a “voice” track, a “sound effects” track, etc.

**get\_volume** ()

**register\_sound\_event** (*config, priority=0, block=False*)

Sets up game sounds from the config file.

**Parameters** *config* – Python dictionary which contains the game sounds settings.

**register\_sound\_events** (*config, mode=None, priority=0*)

**set\_volume** (*volume=None, change=None, \*\*kwargs*)

Sets the overall volume of the sound system.

**Parameters**

- **volume** – The new volume level, a floating point value between 0.0 and 1.0. 1.0 is full volume. 0.0 is mute.
- **change** – A positive or negative value between 0.0 and 1.0 of a change in volume that will be made.
- **kwargs** – Not used here. Included because this method is often called from events which might contain additional kwargs.

Note that the volume can never be increased above 1.0. This sound volume level only affects MPF. You might have to set the overall system sound to in the OS.

**unregister\_sound\_event** (*key*)

**unregister\_sound\_events** (*key\_list*)

**class** `mpf.media_controller.core.sound.StreamTrack` (*machine, config*)

Bases: `object`

Parent class for MPF’s “Stream” track which corresponds to Pygame’s music channel.

**Parameters**

- **machine** – The main machine object.
- **config** – Python dictionary containing the configuration settings for this track.

Sounds played on this track are streamed from disk rather than loaded into memory. This is good for background music since those files can be large and there’s only one playing at a time.

**fadeout** (*ms*)

Fades the sound out.

**Parameters** **ms** – The number of milliseconds to fade out the sound.

**pause()**

Pauses the current sound and remembers the current position so playback can be resumed from the same point via the `unpause()` method.

**play** (*sound*, *\*\*settings*)

Plays a sound on this track.

#### Parameters

- **sound** – The MPF sound object to play.
- **\*\*settings** – Additional settings for this sound’s playback.

This stream track only supports playing one sound at a time, so if you call this when a sound is currently playing, the new sound will stop the current sound.

**stop** (*sound=None*)

Stops the playing sound and resets the current position to the beginning.

**unpause()**

Resumes playing of a previously-paused sound. If the sound was not paused, it starts playing it from the beginning.

**class** `mpf.media_controller.core.sound.Track` (*machine*, *name*, *global\_channel\_list*, *config*)

Bases: `object`

Parent class for an MPF track. Each sound track in MPF can be made up of one or more Pygame sound channels to support multiple simultaneous sounds.

#### Parameters

- **machine** – The main machine controller object.
- **name** – A string of the name this channel will be referred to, such as “voice” or “sfx.”
- **global\_channel\_list** – A python list which keeps track of the global Pygame channels in use.
- **config** – A python dictionary containing the configuration settings for this track.

**create\_channel** (*machine*, *global\_channel\_list*)

Factory method which creates a Pygame sound channel to be used with this track.

#### Parameters

- **machine** – The main machine object.
- **global\_channel\_list** – A list which contains the global list of Pygame channels in use by MPF.

**get\_sound()**

Returns the next sound from the queue to be played.

**Returns:** A tuple of the sound object, the priority, and dictionary of additional settings for that sound. If the queue is empty, returns `None`.

This method will ensure that the sound returned has not expired. If the next sound in the queue is expired, it removes it and returns the next one.

**play** (*sound*, *priority*, *\*\*settings*)

Plays a sound on this track.

**Args:** **sound:** The MPF sound object you want to play. **priority:** The relative priority of this sound. **\*\*settings:** One or more additional settings for this playback.

This method will automatically find an available Pygame channel to use.

If this new sound has a higher priority than the lowest playing sound, it will interrupt that sound to play. Otherwise it will be added to the queue to be played when a channel becomes available.

**queue\_sound** (*sound*, *priority*, *exp\_time=None*, *\*\*settings*)

Adds a sound to the queue to be played when a Pygame channel becomes free.

#### Parameters

- **sound** – The MPF sound object.
- **priority** – The priority of this sound.
- **exp\_time** – Real world time of when this sound will expire. (It will not play if the queue is freed up after it expires.)
- **\*\*settings** – Additional settings for this sound’s playback.

Note that this method will insert this sound into a position in the queue based on its priority, so highest-priority sounds are played first.

**stop** (*sound*)

`mpf.media_controller.core.sound.asset_class`  
alias of `Sound` (page 28)

`mpf.media_controller.core.sound.preload_check` (*machine*)

**mpf.media\_controller.core.window module**

#### Module contents

**mpf.media\_controller.decorators package**

#### Submodules

**mpf.media\_controller.decorators.blink module**

#### Module contents

**mpf.media\_controller.display\_modules package**

#### Submodules

**mpf.media\_controller.display\_modules.dmd module**

**mpf.media\_controller.display\_modules.playfield\_lights module**

#### Module contents

**mpf.media\_controller.elements package**

#### Submodules

**mpf.media\_controller.elements.animation module**

**mpf.media\_controller.elements.image module**

**mpf.media\_controller.elements.movie module**

**mpf.media\_controller.elements.shape module**

**mpf.media\_controller.elements.text module**

**mpf.media\_controller.elements.virtualdmd module**

#### Module contents

**mpf.media\_controller.transitions package**

#### Submodules

**mpf.media\_controller.transitions.move\_in module**

**mpf.media\_controller.transitions.move\_out module**

#### Module contents

### 1.3.2 Submodules

**mpf.media\_controller.version module**

### 1.3.3 Module contents

## 1.4 mpf.platform package

### 1.4.1 Submodules

**mpf.platform.fadecandy module**

**mpf.platform.fast module**

Contains the hardware interface and drivers for the FAST Pinball platform hardware, including the FAST Core and WPC controllers as well as FAST I/O boards.

```
class mpf.platform.fast.FASTDMD(machine, sender)
```

```
    Bases: object
```

```
    tick()
```

```
    update(data)
```

```
class mpf.platform.fast.FASTDirectLED(number)
    Bases: object

    color(color)
        Instantly sets this LED to the color passed.

        Parameters
            • color – a 3-item list of integers representing R, G, and B values,
            • each. (0-255) –

    disable()
        Disables (turns off) this LED instantly. For multi-color LEDs it turns all elements off.

    enable()

    fade(color, fade_ms)

    hex_to_rgb(value)

    rgb_to_hex(rgb)

class mpf.platform.fast.FASTDriver(config, sender)
    Bases: object

    Base class for drivers connected to a FAST Controller.

    disable()
        Disables (turns off) this driver.

    enable()
        Enables (turns on) this driver.

    pulse(milliseconds=None)
        Pulses this driver.

    pwm(on_ms=10, off_ms=10, original_on_ms=0, now=True)
        Enables this driver in a pwm pattern.

class mpf.platform.fast.FASTGString(number, sender)
    Bases: object

    off()

    on(brightness=255, fade_ms=0, start=0)

class mpf.platform.fast.FASTMatrixLight(number, sender)
    Bases: object

    off()
        Disables (turns off) this matrix light.

    on(brightness=255, fade_ms=0, start=0)
        Enables (turns on) this driver.

class mpf.platform.fast.FASTSwitch(number, debounce_open, debounce_close, sender)
    Bases: object

class mpf.platform.fast.HardwarePlatform(machine)
    Bases: mpf.system.platform.Platform (page 77)

    Platform class for the FAST hardware controller.

    Parameters machine – The main MachineController instance.
```

**clear\_hw\_rule** (*sw\_name*)

Clears 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** *sw\_name* – The string name of the switch whose rule you want to clear.

**configure\_dmd** ()

Configures a hardware DMD connected to a FAST controller.

**configure\_driver** (*config*, *device\_type*='coil')

**configure\_gi** (*config*)

**configure\_led** (*config*)

**configure\_matrixlight** (*config*)

**configure\_switch** (*config*)

Configures 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 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.

**get\_hw\_switch\_states** ()

**null\_dmd\_sender** (\*args, \*\*kwargs)

**process\_received\_message** (*msg*)

Sends an incoming message from the FAST controller to the proper method for servicing.

**receive\_dx** (*msg*)

**receive\_id** (*msg*)

**receive\_local\_closed** (*msg*)

**receive\_local\_open** (*msg*)

**receive\_lx** (*msg*)

**receive\_ni** (*msg*)

**receive\_nw\_closed** (*msg*)

**receive\_nw\_open** (*msg*)

**receive\_px** (*msg*)

**receive\_rx** (*msg*)

**receive\_sa** (*msg*)

**receive\_sx** (*msg*)

**receive\_wd** (*msg*)

**receive\_wx** (*msg*)

**register\_processor\_connection** (*name, communicator*)

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.

**tick** ()

**update\_leds** ()

Updates 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.

**write\_hw\_rule** (*sw, sw\_activity, coil\_action\_ms, coil=None, pulse\_ms=0, pwm\_on=8, pwm\_off=8, delay=0, recycle\_time=0, debounced=True, drive\_now=False*)

Used to write (or update) a hardware rule to the FAST controller.

*Hardware Rules* are used to configure the hardware controller to automatically change driver states based on switch changes. These rules are completely handled by the hardware (i.e. with no interaction from the Python game code). They're used for things that you want to happen fast, like firing coils when flipper buttons are pushed, slingshots, pop bumpers, etc.

You can overwrite existing hardware rules at any time to change or remove them.

#### Parameters

- **sw** – Which switch you're creating this rule for. The parameter is a reference to the switch object itself.
- **sw\_activity** – Int which specifies whether this coil should fire when the switch becomes active (1) or inactive (0)
- **coil\_action\_ms** – Int of the total time (in ms) that this coil action should take place. A value of -1 means it's forever. A value of 0 means the coil disables itself when this switch goes into the state specified.
- **coil** – The coil object this rule is for.
- **pulse\_ms** – How long should the coil be pulsed (ms)
- **pwm\_on** – Integer 0 (off) through 8 (100% on) for the initial pwm power of this coil
- **pwm\_off** – pwm level 0-8 of the power of this coil during the hold phase (after the initial kick).
- **delay** – Not currently implemented
- **recycle\_time** – How long (in ms) should this switch rule wait before firing again. Put another way, what's the "fastest" this rule can fire? This is used to prevent "machine gunning" of slingshots and pop bumpers. Do not use it with flippers.
- **debounced** – Should the hardware fire this coil after the switch has been debounced?
- **drive\_now** – Should the hardware check the state of the switches when this rule is first applied, and fire the coils if they should be? Typically this is True, especially with flippers because you want them to fire if the player is holding in the buttons when the machine enables the flippers (which is done via several calls to this method.)

```
class mpf.platform.fast.SerialCommunicator(machine, platform, port, baud, send_queue, receive_queue)
    Bases: object

    identify_connection()
        Identifies which processor this serial connection is talking to.

    query_fast_io_boards()
        Queries the NET processor to see if any FAST IO boards are connected, and if so, queries the IO boards to log them and make sure they're the proper firmware version.

    send(msg)
        Sends a message to the remote processor over the serial connection.

        Parameters msg – String of the message you want to send. The <CR> character will be added automatically.

    stop()
        Stops and shuts down this serial connection.
```

### mpf.platform.openpixel module

### mpf.platform.p3\_roc module

Contains the drivers and interface code for pinball machines which use the Multimorphic P3-ROC hardware controllers.

This code can be used with P-ROC driver boards, or with Stern SAM, Stern Whitestar, Williams WPC, or Williams WPC95 driver boards.

Much of this code is from the P-ROC drivers section of the pyprocgame project, written by Adam Preble and Gerry Stellenberg. It was originally released under the MIT license and is released here under the MIT License.

More info on the P3-ROC hardware platform: <http://pinballcontrollers.com/>

Original code source on which this module was based: <https://github.com/preble/pyprocgame>

If you want to use the Mission Pinball Framework with P3-ROC hardware, you also need libpinproc and pypinproc. More info: <http://www.pinballcontrollers.com/forum/index.php?board=10.0>

```
class mpf.platform.p3_roc.DriverAlias(key, value)
    Bases: object

    decode(addr)

    matches(addr)

class mpf.platform.p3_roc.HardwarePlatform(machine)
    Bases: mpf.system.platform.Platform (page 77)

    Platform class for the P3-ROC hardware controller.

    Parameters machine – The MachineController instance.

    machine
        The MachineController instance.

    proc
        The P3-ROC pinproc.PinPROC device.

    machine_type
        Constant of the pinproc.MachineType
```



**clear\_hw\_rule** (*sw\_name*)

Clears 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** *sw\_num* (*int*) – The number of the switch whose rule you want to clear.

**configure\_dmd** ()

The P3-ROC does not support a physical DMD, so this method does nothing. It's included here in case it's called by mistake.

**configure\_driver** (*config*, *device\_type*='coil')

Creates 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.
- **device\_type** – String with value of either 'coil' or 'switch'.

**Returns** A reference to the PROCDriver object which is the actual object you can use to pulse(), patter(), enable(), etc.

**configure\_gi** (*config*)

Configures a P3-ROC GI string light.

**configure\_led** (*config*)

Configures a P3-ROC RGB LED controlled via a PD-LED.

**configure\_matrixlight** (*config*)

Configures a P3-ROC matrix light.

**configure\_switch** (*config*)

Configures a P3-ROC switch.

**Parameters**

- **config** – Dictionary of settings for the switch. In the case of the P3-ROC, it uses the following:
- **number** – The number (or number string) for the switch as specified in the machine configuration file.
- **debounce** – Boolean which specifies whether the P3-ROC should debounce this switch first before sending open and close notifications to the host computer.

**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.

**state** [An integer of the current hardware state of the switch, used] to set the initial state state in the machine. A value of 0 means the switch is open, and 1 means it's closed. Note this state is the physical state of the switch, so if you configure the switch to be normally-closed (i.e. "inverted" then your code will have to invert it too.) MPF handles this automatically if the switch type is 'NC'.

**Return type** switch

**get\_hw\_switch\_states** ()

**tick** ()

Checks the P3-ROC for any events (switch state changes).

Also tickles the watchdog and flushes any queued commands to the P3-ROC.

**write\_hw\_rule** (sw, sw\_activity, coil\_action\_ms, coil=None, pulse\_ms=0, pwm\_on=0, pwm\_off=0, delay=0, recycle\_time=0, debounced=True, drive\_now=False)

Used to write (or update) a hardware rule to the P3-ROC.

*Hardware Rules* are used to configure the P3-ROC to automatically change driver states based on switch changes. These rules are completely handled by the P3-ROC hardware (i.e. with no interaction from the Python game code). They're used for things that you want to happen fast, like firing coils when flipper buttons are pushed, slingshots, pop bumpers, etc.

You can overwrite existing hardware rules at any time to change or remove them.

#### Parameters

- **sw** – switch object Which switch you're creating this rule for. The parameter is a reference to the switch object itself.
- **sw\_activity** – int Do you want this coil to fire when the switch becomes active (1) or inactive (0)
- **coil\_action\_ms** – int The total time (in ms) that this coil action should take place. A value of -1 means it's forever.
- **coil** – coil object Which coil is this rule controlling
- **pulse\_ms** – int How long should the coil be pulsed (ms)
- **pwm\_on** – int If the coil should be held on at less than 100% duty cycle, this is the “on” time (in ms).
- **pwm\_off** – int If the coil should be held on at less than 100% duty cycle, this is the “off” time (in ms).
- **delay** – int Not currently implemented for the P3-ROC hardware
- **recycle\_time** – int How long (in ms) should this switch rule wait before firing again. Put another way, what's the “fastest” this rule can fire? This is used to prevent “machine gunning” of slingshots and pop bumpers. Do not use it with flippers. Note the P3-ROC has a non-configurable delay time of 125ms. (So it's either 125ms or 0.) So if you set this delay to anything other than 0, it will be 125ms.
- **debounced** – bool Should the P3-ROC fire this coil after the switch has been debounced? Typically no.
- **drive\_now** – bool Should the P3-ROC check the state of the switches when this rule is first applied, and fire the coils if they should be? Typically this is True, especially with flippers because you want them to fire if the player is holding in the buttons when the machine enables the flippers (which is done via several calls to this method.)

**class** mpf.platform.p3\_roc.PDBCoil (pdb, number\_str)

Bases: object

Base class for coils connected to a P3-ROC that are controlled via P3-ROC driver boards (i.e. the PD-16 board).

**bank** ()

```

    is_direct_coil (string)
    is_pdb_coil (string)
    output ()
class mpf.platform.p3_roc.PDBConfig (proc, config)
    Bases: object

    This class is only used when the P3-ROC is configured to use P3-ROC driver boards such as the PD-16 or
    PD-8x8. i.e. not when it's operating in WPC or Stern mode.

    aliases = None
    configure_globals (proc, lamp_source_bank_list, enable=True)
    get_globals (config)
    get_proc_number (device_type, number_str)
        Returns the P3-ROC number for the requested driver string.

        This method uses the driver string to look in the indexes list that was set up when the PDBs were config-
        ured. The resulting P3-ROC index * 3 is the first driver number in the group, and the driver offset is to
        that.

    indexes = []
    initialize_drivers (proc)
    proc = None
class mpf.platform.p3_roc.PDBLED (board, address, proc_driver, invert=False)
    Bases: object

    Represents an RGB LED connected to a PD-LED board.

    color (color)
        Instantly sets this LED to the color passed.

        Parameters

- color – a 3-item list of integers representing R, G, and B values,
- each. (0-255) –

disable ()
        Disables (turns off) this LED instantly. For multi-color LEDs it turns all elements off.

    enable ()
        Enables (turns on) this LED instantly. For multi-color LEDs it turns all elements on.

    fade (color, fade_ms)
    normalize_color (color)
class mpf.platform.p3_roc.PDBLight (pdb, number_str)
    Bases: object

    Base class for lights connected to a PD-8x8 driver board.

    dedicated_bank ()
    dedicated_output ()
    is_direct_lamp (string)
    is_pdb_lamp (string)

```

**sink\_bank()**

**sink\_board()**

**sink\_output()**

**source\_bank()**

**source\_board()**

**source\_output()**

**split\_matrix\_addr\_parts** (*string*)

Input is of form C-Ax-By-z:R-Ax-By-z or C-x/y/z:R-x/y/z or aliasX:aliasY. We want to return only the address part: Ax-By-z, x/y/z, or aliasX. That is, remove the two character prefix if present.

**class** mpf.platform.p3\_roc.**PDBSwitch** (*pdb, number\_str*)

Bases: object

Base class for switches connected to a P3-ROC.

**parse\_matrix\_num** (*num\_str*)

**proc\_num** ()

**class** mpf.platform.p3\_roc.**PROCDriver** (*number, proc\_driver*)

Bases: object

Base class for drivers connected to a P3-ROC. This class is used for all drivers, regardless of whether they're connected to a P-ROC driver board (such as the PD-16 or PD-8x8) or an OEM driver board.

**disable** ()

Disables (turns off) this driver.

**enable** ()

Enables (turns on) this driver.

**future\_pulse** (*milliseconds=None, timestamp=0*)

Enables this driver for *milliseconds* at P3-ROC timestamp: *timestamp*. If no parameter is provided for *milliseconds*, *pulse\_ms* is used. If no parameter is provided or *timestamp*, 0 is used. *ValueError* will be raised if *milliseconds* is outside of the range 0-255.

**pulse** (*milliseconds=None*)

Enables this driver for *milliseconds*.

*ValueError* will be raised if *milliseconds* is outside of the range 0-255.

**pwm** (*on\_ms=10, off\_ms=10, original\_on\_ms=0, now=True*)

Enables a pitter-patter sequence.

It starts by activating the driver for *original\_on\_ms* milliseconds. Then it repeatedly turns the driver on for *on\_ms* milliseconds and off for *off\_ms* milliseconds.

**schedule** (*schedule, cycle\_seconds=0, now=True*)

Schedules this driver to be enabled according to the given *schedule* bitmask.

**state** ()

Returns a dictionary representing this driver's current configuration state.

**tick** ()

**timed\_pwm** (*on\_ms=10, off\_ms=10, run\_time=0, now=True*)

Enables a pitter-patter sequence that runs for *run\_time* milliseconds.

Until it ends, the sequence repeatedly turns the driver on for *on\_ms* milliseconds and off for *off\_ms* milliseconds.

```
class mpf.platform.p3_roc.PROCMatrixLight (number, proc_driver)
```

Bases: object

```
off ()
```

Disables (turns off) this driver.

```
on (brightness=255, fade_ms=0, start=0)
```

Enables (turns on) this driver.

```
class mpf.platform.p3_roc.PROCSwitch (number)
```

Bases: object

```
mpf.platform.p3_roc.decode_pdb_address (addr, aliases=[])
```

Decodes Ax-By-z or x/y/z into PDB address, bank number, and output number.

Raises a ValueError exception if it is not a PDB address, otherwise returns a tuple of (addr, bank, number).

```
mpf.platform.p3_roc.is_pdb_address (addr, aliases=[])
```

Return True if the given address is a valid PDB address.

## mpf.platform.p\_roc module

Contains the drivers and interface code for pinball machines which use the Multimorphic R-ROC hardware controllers.

This code can be used with P-ROC driver boards, or with Stern SAM, Stern Whitestar, Williams WPC, or Williams WPC95 driver boards.

Much of this code is from the P-ROC drivers section of the pyprocgame project, written by Adam Preble and Gerry Stellenberg. It was originally released under the MIT license and is released here under the MIT License.

More info on the P-ROC hardware platform: <http://pinballcontrollers.com/>

Original code source on which this module was based: <https://github.com/preble/pyprocgame>

If you want to use the Mission Pinball Framework with P-ROC hardware, you also need libpinproc and pypinproc. More info: <http://www.pinballcontrollers.com/forum/index.php?board=10.0>

```
class mpf.platform.p_roc.DriverAlias (key, value)
```

Bases: object

```
decode (addr)
```

```
matches (addr)
```

```
class mpf.platform.p_roc.HardwarePlatform (machine)
```

Bases: `mpf.system.platform.Platform` (page 77)

Platform class for the P-ROC hardware controller.

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

**machine**

The MachineController instance.

**proc**

The P-ROC pinproc.PinPROC device.

**machine\_type**

Constant of the pinproc.MachineType

```
clear_hw_rule (sw_name)
```

Clears 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** *sw\_num* – Int of the number of the switch whose rule you want to clear.

**configure\_dmd** ()

Configures a hardware DMD connected to a classic P-ROC.

**configure\_driver** (*config*, *device\_type*='coil')

Creates 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.
- **device\_type** – String with value of either 'coil' or 'switch'.

**Returns** A reference to the PROCDriver object which is the actual object you can use to pulse(), patter(), enable(), etc.

**configure\_gi** (*config*)

Configures a P-ROC GI string light.

**configure\_led** (*config*)

Configures a P-ROC RGB LED controlled via a PD-LED.

**configure\_matrixlight** (*config*)

Configures a P-ROC matrix light.

**configure\_switch** (*config*)

Configures a P-ROC switch.

**Parameters**

- **config** – Dictionary of settings for the switch. In the case of the P-ROC, it uses the following:
- **number** – The number (or number string) for the switch as specified in the machine configuration file.
- **debounce** – Boolean which specifies whether the P-ROC should debounce this switch first before sending open and close notifications to the host computer.

**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.

**state** [An integer of the current hardware state of the switch, used] to set the initial state in the machine. A value of 0 means the switch is open, and 1 means it's closed. Note this state is the physical state of the switch, so if you configure the switch to be normally-closed (i.e. "inverted" then your code will have to invert it too.) MPF handles this automatically if the switch type is 'NC'.

**Return type** switch

**get\_hw\_switch\_states** ()

**tick()**

Checks 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.

**write\_hw\_rule** (*sw*, *sw\_activity*, *coil\_action\_ms*, *coil=None*, *pulse\_ms=0*, *pwm\_on=0*, *pwm\_off=0*, *delay=0*, *recycle\_time=0*, *debounced=True*, *drive\_now=False*)

Used to write (or update) a hardware rule to the P-ROC.

*Hardware Rules* are used to configure the P-ROC to automatically change driver states based on switch changes. These rules are completely handled by the P-ROC hardware (i.e. with no interaction from the Python game code). They're used for things that you want to happen fast, like firing coils when flipper buttons are pushed, slingshots, pop bumpers, etc.

You can overwrite existing hardware rules at any time to change or remove them.

#### Parameters

- **sw** (*switch object*) – Which switch you're creating this rule for. The parameter is a reference to the switch object itself.
- **sw\_activity** (*int*) – Do you want this coil to fire when the switch becomes active (1) or inactive (0)
- **coil\_action\_ms** (*int*) – The total time (in ms) that this coil action should take place. A value of -1 means it's forever.
- **coil** (*coil object*) – Which coil is this rule controlling
- **pulse\_ms** (*int*) – How long should the coil be pulsed (ms)
- **pwm\_on** (*int*) – If the coil should be held on at less than 100% duty cycle, this is the “on” time (in ms).
- **pwm\_off** (*int*) – If the coil should be held on at less than 100% duty cycle, this is the “off” time (in ms).
- **delay** (*int*) – Not currently implemented for the P-ROC hardware
- **recycle\_time** (*int*) – How long (in ms) should this switch rule wait before firing again. Put another way, what's the “fastest” this rule can fire? This is used to prevent “machine gunning” of slingshots and pop bumpers. Do not use it with flippers. Note the P-ROC has a non-configurable delay time of 125ms. (So it's either 125ms or 0.) So if you set this delay to anything other than 0, it will be 125ms.
- **debounced** (*bool*) – Should the P-ROC fire this coil after the switch has been debounced? Typically no.
- **drive\_now** (*bool*) – Should the P-ROC check the state of the switches when this rule is first applied, and fire the coils if they should be? Typically this is True, especially with flippers because you want them to fire if the player is holding in the buttons when the machine enables the flippers (which is done via several calls to this method.)

**class** mpf.platform.p\_roc.PDBCoil (*pdb*, *number\_str*)

Bases: object

Base class for coils connected to a P-ROC that are controlled via P-ROC driver boards (i.e. the PD-16 board).

**bank()**

**is\_direct\_coil** (*string*)

**is\_pdb\_coil** (*string*)

**output()**

**class** `mpf.platform.p_roc.PDBConfig (proc, config)`

Bases: `object`

This class is only used when the P-ROC is configured to use P-ROC driver boards such as the PD-16 or PD-8x8. i.e. not when it's operating in WPC or Stern mode.

**aliases** = `None`

**configure\_globals** (*proc, lamp\_source\_bank\_list, enable=True*)

**get\_proc\_number** (*device\_type, number\_str*)

Returns the P-ROC number for the requested driver string.

This method uses the driver string to look in the indexes list that was set up when the PDBs were configured. The resulting P-ROC index \* 3 is the first driver number in the group, and the driver offset is to that.

**indexes** = `[]`

**initialize\_drivers** (*proc*)

**proc** = `None`

**class** `mpf.platform.p_roc.PDBLED (board, address, proc_driver, invert=False)`

Bases: `object`

Represents an RGB LED connected to a PD-LED board.

**color** (*color*)

Instantly sets this LED to the color passed.

**Parameters**

- **color** – a 3-item list of integers representing R, G, and B values,
- **each.** (0-255) –

**disable** ()

Disables (turns off) this LED instantly. For multi-color LEDs it turns all elements off.

**enable** ()

Enables (turns on) this LED instantly. For multi-color LEDs it turns all elements on.

**fade** (*color, fade\_ms*)

**normalize\_color** (*color*)

**class** `mpf.platform.p_roc.PDBLight (pdb, number_str)`

Bases: `object`

Base class for lights connected to a PD-8x8 driver board.

**dedicated\_bank** ()

**dedicated\_output** ()

**is\_direct\_lamp** (*string*)

**is\_pdb\_lamp** (*string*)

**sink\_bank** ()

**sink\_board** ()

**sink\_output** ()

**source\_bank** ()



**source\_board()**

**source\_output()**

**split\_matrix\_addr\_parts** (*string*)

Input is of form C-Ax-By-z:R-Ax-By-z or C-x/y/z:R-x/y/z or aliasX:aliasY. We want to return only the address part: Ax-By-z, x/y/z, or aliasX. That is, remove the two character prefix if present.

**class** `mpf.platform.p_roc.PDBSwitch` (*pdb, number\_str*)

Bases: `object`

Base class for switches connected to a P-ROC.

**parse\_matrix\_num** (*num\_str*)

**proc\_num()**

**class** `mpf.platform.p_roc.PROCDMD` (*proc, machine*)

Bases: `object`

Parent class for a physical DMD attached to a P-ROC.

#### Parameters

- **proc** – Reference to the MachineController’s `proc` attribute.
- **machine** – Reference to the MachineController

**dmd**

Reference to the P-ROC’s DMD buffer.

**tick()**

Updates the physical DMD with the latest frame data. Meant to be called once per machine tick.

**update** (*data*)

Updates the DMD with a new frame.

**Parameters** *data* – A 4096-byte raw string.

**class** `mpf.platform.p_roc.PROCDriver` (*number, proc\_driver*)

Bases: `object`

Base class for drivers connected to a P-ROC. This class is used for all drivers, regardless of whether they’re connected to a P-ROC driver board (such as the PD-16 or PD-8x8) or an OEM driver board.

**disable()**

Disables (turns off) this driver.

**enable()**

Enables (turns on) this driver.

**future\_pulse** (*milliseconds=None, timestamp=0*)

Enables this driver for *milliseconds* at P-ROC timestamp: *timestamp*. If no parameter is provided for *milliseconds*, *pulse\_ms* is used. If no parameter is provided or *timestamp*, 0 is used. `ValueError` will be raised if *milliseconds* is outside of the range 0-255.

**pulse** (*milliseconds=None*)

Enables this driver for *milliseconds*.

`ValueError` will be raised if *milliseconds* is outside of the range 0-255.

**pwm** (*on\_ms=10, off\_ms=10, original\_on\_ms=0, now=True*)

Enables a pitter-patter sequence.

It starts by activating the driver for *original\_on\_ms* milliseconds. Then it repeatedly turns the driver on for *on\_ms* milliseconds and off for *off\_ms* milliseconds.

**schedule** (*schedule*, *cycle\_seconds=0*, *now=True*)

Schedules this driver to be enabled according to the given *schedule* bitmask.

**state** ()

Returns a dictionary representing this driver's current configuration state.

**tick** ()

**timed\_pwm** (*on\_ms=10*, *off\_ms=10*, *run\_time=0*, *now=True*)

Enables a pitter-patter sequence that runs for *run\_time* milliseconds.

Until it ends, the sequence repeatedly turns the driver on for *on\_ms* milliseconds and off for *off\_ms* milliseconds.

**class** `mpf.platform.p_roc.PROCMatrixLight` (*number*, *proc\_driver*)

Bases: `object`

**off** ()

Disables (turns off) this driver.

**on** (*brightness=255*, *fade\_ms=0*, *start=0*)

Enables (turns on) this driver.

**class** `mpf.platform.p_roc.PROCSwitch` (*number*)

Bases: `object`

`mpf.platform.p_roc.decode_pdb_address` (*addr*, *aliases=[]*)

Decodes Ax-By-z or x/y/z into PDB address, bank number, and output number.

Raises a `ValueError` exception if it is not a PDB address, otherwise returns a tuple of (*addr*, *bank*, *number*).

`mpf.platform.p_roc.is_pdb_address` (*addr*, *aliases=[]*)

Return `True` if the given address is a valid PDB address.

## **mpf.platform.virtual module**

Contains code for a virtual hardware platform. At this point this is more for testing before you have a P-ROC or FAST board installed. Eventually this can be used to allow the MPF to drive PinMAME and Virtual Pinball machines.

This is similar to the P-ROC's 'FakePinPROC' mode of operation, though unlike that it doesn't require any P-ROC drivers or modules to be installed.

**class** `mpf.platform.virtual.HardwarePlatform` (*machine*)

Bases: `mpf.system.platform.Platform` (page 77)

Base class for the virtual hardware platform.

**clear\_hw\_rule** (*sw\_name*)

**configure\_dmd** ()

**configure\_driver** (*config*, *device\_type='coil'*)

**configure\_gi** (*config*)

**configure\_led** (*config*)

**configure\_matrixlight** (*config*)

**configure\_switch** (*config*)

**get\_hw\_switch\_states** ()

**write\_hw\_rule** (*\*args*, *\*\*kwargs*)

```
class mpf.platform.virtual.VirtualDMD (machine)
    Bases: object

    update (data)

class mpf.platform.virtual.VirtualDriver (number)
    Bases: object

    disable ()

    enable ()

    future_pulse (milliseconds=None, timestamp=0)

    pulse (milliseconds=None)

    pulsed_patter (on_ms=10, off_ms=10, run_time=0, now=True)

    pwm (on_ms=10, off_ms=10, original_on_ms=0, now=True)

    reconfigure (polarity)

    schedule (schedule, cycle_seconds=0, now=True)

    state ()

    tick ()

class mpf.platform.virtual.VirtualGI (number)
    Bases: object

    off ()

    on (brightness, fade_ms, start)

class mpf.platform.virtual.VirtualLED (number)
    Bases: object

    color (color, fade_ms=0, brightness_compensation=True)

    disable ()

    enable (brightness_compensation=True)

class mpf.platform.virtual.VirtualMatrixLight (number)
    Bases: object

    off ()

    on (brightness=255, fade_ms=0, start=0)

class mpf.platform.virtual.VirtualSwitch (number)
    Bases: object

    Represents a switch in a pinball machine used with virtual hardware.
```

## 1.4.2 Module contents

# 1.5 mpf.plugins package

## 1.5.1 Submodules

### mpf.plugins.auditor module

MPF plugin for an auditor which records switch events, high scores, shots, etc.

**class** `mpf.plugins.auditor.Auditor` (*machine*)

Bases: `object`

**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.
- **\*\*kwargs** – 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*)

**audit\_switch** (*switch\_name*, *state*)

**disable** (*\*\*kwargs*)

Disables the auditor.

**enable** (*\*\*kwargs*)

Enables 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.

**load\_from\_disk** (*filename*)

Loads an audit log from disk.

**Parameters filename** – The path and file of the audit file location.

**make\_sure\_path\_exists** (*path*)

Checks to see if the audits folder exists and creates it if not.

**save\_to\_disk** (*filename*)

Dumps the audits from memory to disk.

**Parameters** *filename* – The path and file the audits will be written to.

`mpf.plugins.auditor.plugin_class`  
alias of `Auditor` (page 48)

### mpf.plugins.ball\_save module

MPF plugin for a ball saver code which is used to give the player another ball if their first ball drains too fast.

This plugin is not yet finished and doesn't work yet.

**class** `mpf.plugins.ball_save.BallSave` (*game*)  
Bases: `object`

Base class which implements a ball saver instance. You can use this as-is, enhance it, or replace it altogether.

**ball\_drain** (*balls*)

**enable** (*time=None, balls=None*)

`mpf.plugins.ball_save.plugin_class`  
alias of `BallSave` (page 49)

### mpf.plugins.ball\_search module

MPF plugin for a ball search module which actually controls the coils to search for a missing pinball.

This module is not yet complete and does not work.

**class** `mpf.plugins.ball_search.BallSearch` (*machine*)  
Bases: `object`

Base class which implements the ball search functionality.

This module is responsible for actually firing the coils and moving motors, etc. when the ball search begins. It can respond to multiple “phases” of ball search. (For example, for the few round it might only do easy things like fire pop bumpers. If it doesn't find the ball after that, it will start trying to eject balls from ball devices.)

This ball search module is not responsible for deciding to start or stop a ball search—that is something that Ball Controller does. Also this ball search module doesn't know when a ball is actually found. If a playfield switch is hit then the ball live event will be raised and the ball controller will tell this ball search module that it can stop looking.

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

**end** ()

Ends the active ball search.

**pop\_coil** (*coil*)

Setviates the ‘coil’ based on it's default pulse time. Holds a coil open for the hold time in sec.

This is not yet implemented. (It's copied in from our ball\_controller code from our old python project.)

**start** ()

Begin the ball search process

**tick** ()

Method that runs as a task

`mpf.plugins.ball_search.plugin_class`  
alias of [BallSearch](#) (page 49)

### **mpf.plugins.info\_lights module**

MPF plugin which uses lights to represent Game functions. Typically in an EM machine

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

    ball_started()

    game_ended()

    game_starting(**kwargs)

    match(match)

    player_added(player)

    reset_game_lights()

    tilt()
```

`mpf.plugins.info_lights.plugin_class`  
alias of [InfoLights](#) (page 50)

### **mpf.plugins.osc module**

MPF plugin allows a machine to be controlled by an OSC client.

```
class mpf.plugins.osc.OSC(machine)
    Bases: object

    client_send_osc_message(category, name, data)
        Sends an OSC message to the client to update it Parameters: category - type of update, sw, coil, lamp, led,
        etc. name - the name of the object we're updating data - the data we're sending

    client_update_all()
        Update the OSC client. Good for when it switches to a new tab or connects a new client

    client_update_all_switches()
        Updates all the switch states on the OSC client.

    client_update_light(light_name, brightness)

    client_update_switch(switch_name, ms, state)

    found_new_osc_client(address)

    process_coil(coil, data)
        Processes a coil event received from the OSC client.

    process_config(event, data)

    process_event(event, data)
        Posts an MPF event based on an event received from the OSC client.

    process_flipper(flipper, data)
        Calls the flipper's sw_flip() or sw_release() event.

    process_light(light, data)
        Processes a light event received from the OSC client.
```

**process\_message** (*addr, tags, data, client\_address*)  
 Receives OSC messages and acts on them.

**process\_switch** (*switch, data*)  
 Processes a switch event received from the OSC client.

**register\_data** ()

**register\_lights** ()  
 Adds handlers to all lights so the OSC client can receive updates.

**register\_switches** ()  
 Adds switch handlers to all switches so the OSC client can receive updates.

**setup\_osc\_client** (*address*)  
 Setup a new OSC client

**start** ()  
 Starts the OSC server.

**stop** ()  
 Stops the OSC server.

**update\_audits** (*event, data*)  
 Sends audit data to the OSC client.

**update\_ball** (*\*\*kwargs*)

**update\_config** (*event, data*)  
 Sends config data to the OSC client.

**update\_player** (*\*\*kwargs*)

**update\_score** (*\*\*kwargs*)

`mpf.plugins.osc.plugin_class`  
 alias of [OSC](#) (page 50)

## mpf.plugins.socket\_events module

MPF plugin which sends events to sockets

**class** `mpf.plugins.socket_events.SocketClient` (*machine*)  
 Bases: `object`

**process\_config** (*config*)  
 Processes the SocketEvents from the config.

**Parameters** *config* – Dictionary of the config to process.

**send\_message** (*message*)  
 Sends a message to the remote socket host.

**Parameters** *message* – String of the message to send.

**setup\_client** (*host, port*)  
 Sets up the socket client.

### Parameters

- **host** – String of the host name.
- **port** – Int of the port name.

**stop\_client()**

Stops and shuts down the socket client.

`mpf.plugins.socket_events.plugin_class`

alias of `SocketClient` (page 51)

### **mpf.plugins.switch\_player module**

MPF plugin which automatically plays back switch events from the config file.

**class** `mpf.plugins.switch_player.SwitchPlayer(machine)`

Bases: `object`

`mpf.plugins.switch_player.plugin_class`

alias of `SwitchPlayer` (page 52)

## **1.5.2 Module contents**

# **1.6 mpf.system package**

## **1.6.1 Submodules**

### **mpf.system.assets module**

Contains `AssetManager`, `AssetLoader`, and `Asset` parent classes

**class** `mpf.system.assets.Asset(machine, config, file_name, asset_manager)`

Bases: `object`

**do\_load(callback)**

**load(callback=None)**

**unload()**

**class** `mpf.system.assets.AssetLoader(name, queue, machine)`

Bases: `threading.Thread`

Base class for the Asset Loader with runs as a separate thread and actually loads the assets from disk.

#### **Parameters**

- **name** – String name of what this loader will be called. (Only really used to give a friendly name to it in logs.)
- **queue** – A reference to the asset loader `Queue` which holds assets waiting to be loaded.
- **machine** – The main `MachineController` object.

**run()**

Run loop for the loader thread.

**class** `mpf.system.assets.AssetManager(machine, config_section, path_string, asset_class, asset_attribute, file_extensions)`

Bases: `object`

Base class for an Asset Manager.

#### **Parameters**



- **machine** – The main `MachineController` object.
- **config\_section** – String of the name of the section in the config file for the asset settings that this Asset Manager will machine. e.g. 'image'.
- **path\_string** – The setting in the paths section of the config file that specifies what path these asset files are in. e.g. 'images'.
- **asset\_class** – A class object of the base class for the assets that this Asset Manager will manage. e.g. `Image`.
- **asset\_attribute** – The string name that you want to refer to this asset collection as. e.g. a value of 'images' means that assets will be accessible via `self.machine.images`.
- **file\_extensions** – A tuple of strings of valid file extensions that files for this asset will use. e.g. ('png', 'jpg', 'jpeg', 'bmp')

There will be one Asset Manager for each different type of asset. (e.g. one for images, one for movies, one for sounds, etc.)

#### **create\_loader\_thread()**

Creates a loader thread which will handle the actual reading from disk and loading into memory for assets of this class. Note that one loader thread is created for each class of assets used in your game.

Note that this asset loader as a separate *thread*, not a separate *process*. It will run on the same core as your main MPF Python instance.

Note that it's possible to call this method multiple times to create multiple loader threads, but that will not make things load any faster since this process is limited by CPU and disk I/O. In fact if it's a magnetic disk, think multiple threads would make it slower.

#### **load\_asset(asset, callback, priority=10)**

Loads an asset into memory.

##### **Parameters**

- **asset** – The Asset object to load.
- **callback** – The callback that will be called once the asset has been loaded by the loader thread.
- **priority** – The relative loading priority of the asset. If there's a queue of assets waiting to be loaded, this load request will be inserted into the queue in a position based on its priority.

#### **load\_assets(config, mode=None, load\_key=None, callback=None, \*\*kwargs)**

Loads the assets from a config dictionary.

##### **Parameters**

- **config** – Dictionary that holds the assets to load.
- **mode** – Not used. Included here since this method is registered as a mode start handler.
- **load\_key** – String name of the load key which specifies which assets should be loaded.
- **callback** – Callback method which is called by each asset once it's loaded.
- **\*\*kwargs** – Not used. Included to allow this method to be used as an event handler.

The assets must already be registered in order for this method to work.

#### **locate\_asset\_file(file\_name, path=None)**

Takes a file name and a root path and returns a link to the absolute path of the file

##### **Parameters**

- **file\_name** – String of the file name
- **path** – root of the path to check (without the specific asset path string)

Returns: String of the full path (path + file name) of the asset.

Note this method will add the path string between the path you pass and the file. Also if it can't find the file in the path you pass, it will look for the file in the machine root plus the path string location.

**process\_assets\_from\_disk** (*config*, *path=None*)

Looks at a path and finds all the assets in the folder. Looks in a subfolder based on the asset's path string. Crawls subfolders too. The first subfolder it finds is used for the asset's default config section. If an asset has a related entry in the config file, it will create the asset with that config. Otherwise it uses the default

#### Parameters

- **config** – A dictionary which contains a list of asset names with settings that will be used for the specific asset. (Note this is not needed for all assets, as any asset file found not in the config dictionary will be set up with the folder it was found in's `asset_defaults` settings.)
- **path** – A full system path to the root folder that will be searched for assets. This should *not* include the asset-specific path string. If omitted, only the machine's root folder will be searched.

**register\_and\_load\_machine\_assets** ()

Called on MPF boot to register any assets found in the machine-wide configuration files. (i.e. any assets not specified in mode config files.)

If an asset is set with the load type of 'preload', this method will also load the asset file into memory.

**register\_asset** (*asset*, *config*)

Registers an asset with the Asset Manager.

#### Parameters

- **asset** – String name of the asset to register.
- **config** – Dictionary which contains settings for this asset.

Registering an asset is what makes it available to be used in the game. Note that registering an asset is separate from loading an asset. All assets will be registered on MPF boot, but they can be loaded and unloaded as needed to save on memory.

**register\_assets** (*config*, *mode\_path=None*)

Scans a config dictionary and registers any asset entries it finds.

#### Args:

**config:** A dictionary of asset entries. This dictionary needs to be "localized" to just the section for this particular asset type. e.g. if you're loading "Images" the keys of this dictionary should be `image_1`, `image_2`, etc., not "Images".

**mode\_path:** The full path to the base folder that will be searched for the asset file on disk. This folder should *not* include the asset-specific folder. If omitted, the base machine folder will be searched.

Note that this method merely registers the assets so they can be referenced in MPF. It does not actually load the asset files into memory.

**setup\_defaults** (*config*)

Processed the `asset_defaults` section of the machine config files.

**unload\_assets** (*asset\_set*)

Unloads assets from memory.

**Parameters** `asset_set` – A set (or any iterable) of Asset objects which will be unloaded.

Unloading an asset does not de-register it. It's still available to be used, but it's just unloaded from memory to save on memory.

## **mpf.system.ball\_controller module**

Contrains the BallController class which manages and tracks all the balls in a pinball machine.

**class** `mpf.system.ball_controller.BallController` (*machine*)

Bases: `object`

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.

**are\_balls\_gathered** (*target=['home', 'trough']*)

Checks 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 value of the tag you'd like to check. Default is
- **'home'** –

## **balls**

**gather\_balls** (*target='home', antitarget=None*)

Used to ensure that all balls are in (or not in) ball devices with the tag 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'.

### **Parameters**

- **target** – A string of the tag name of the ball devices you want all the balls to end up in. Default is 'home'.
- **antitarget** – The opposite of target. Will eject all balls from all devices with the string you pass. Default is None.

Note you can't pass both a target and antitarget in the same call. (If you do it will just use the target and ignore the antitarget.)

TODO: Add support to actually move balls into position. e.g. STTNG, the lock at the top of the playfield wants to hold a ball before a game starts, so when a game ends the machine will auto eject one from the plunger with the diverter set so it's held in the rear lock.

## **num\_balls\_known**

**request\_to\_start\_game** ()

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.

## mpf.system.bcp module

MPF plugin which enables the Backbox Control Protocol (BCP) v1.0alpha

**class** `mpf.system.bcp.BCP` (*machine*)

Bases: `object`

The parent class for the BCP client.

This class can support connections with multiple remote hosts at the same time using multiple instances of the `BCPClientSocket` class.

**Parameters** `machine` – A reference to the main MPF machine object.

**The following BCP commands are currently implemented:** `ball_start?player_num=x&ball=x`  
`ball_end`      `config?volume=0.5`      `error`      `get`      `goodbye`      `hello?version=xxx`  
`mode_start?name=xxx&priority=xxx`      `mode_stop?name=xxx`      `player_added?player_num=x`  
`player_score?value=x&prev_value=x&change=x&player_num=x`      `player_turn_start?player_num=x`  
`player_variable?name=x&value=x&prev_value=x&change=x&player_num=x`      `set`      `shot?name=x`  
`switch?name=x&state=x`      `timer`      `trigger?name=xxx`

**bcp\_mode\_start** (*config, priority, mode, \*\*kwargs*)

Sends BCP 'mode\_start' to the connected BCP hosts and schedules automatic sending of 'mode\_stop' when the mode stops.

**bcp\_mode\_stop** (*name, \*\*kwargs*)

Sends BCP 'mode\_stop' to the connected BCP hosts.

**bcp\_player\_added** (*player, num*)

Sends BCP 'player\_added' to the connected BCP hosts.

**bcp\_receive\_dmd\_frame** (*data*)

Called when the BCP client receives a new DMD frame from the remote BCP host. This method forwards the frame to the physical DMD.

**bcp\_receive\_error** (*\*\*kwargs*)

A remote BCP host has sent a BCP error message, indicating that a command from MPF was not recognized.

This method only posts a warning to the log. It doesn't do anything else at this point.

**bcp\_receive\_get** (*\*\*kwargs*)

Processes an incoming BCP 'get' command.

Note that this media controller doesn't implement the 'get' command at this time, but it's included here for completeness since the 'get' command is part of the BCP 1.0 specification so we don't want to return an error if we receive an incoming 'get' command.

**bcp\_receive\_set** (*\*\*kwargs*)

Processes an incoming BCP 'set' command.

Note that this media controller doesn't implement the 'set' command at this time, but it's included here for completeness since the 'set' command is part of the BCP 1.0 specification so we don't want to return an error if we receive an incoming 'set' command.

**bcp\_receive\_switch** (*\*\*kwargs*)

Processes an incoming switch state change request from a remote BCP host.

**bcp\_receive\_trigger** (*name=None, \*\*kwargs*)

Processes an incoming trigger command from a remote BCP host.

**bcp\_reset ()**

Sends the 'reset' command to the remote BCP host.

**bcp\_trigger (name, \*\*kwargs)**

Sends BCP 'trigger' to the connected BCP hosts.

**create\_trigger\_event (event)**

Registers a BCP trigger based on an MPF event.

**Parameters event** – String name of the event you're registering this trigger for.

The BCP trigger will be registered with the same name as the MPF event. For example, if you pass the event "foo\_event", the BCP command that will be sent when that event is posted will be trigger?name=foo\_event.

**decrease\_volume (track='master', \*\*kwargs)**

Sends a command to the remote BCP host to decrease the volume of a track by 1 unit.

**Parameters**

- **track** – The string name of the track you want to decrease the volume on. Default is 'master'.
- **\*\*kwargs** – Ignored. Included in case this method is used as a callback for an event which has other kwargs.

If this decrease causes the volume to go below zero, the decrease is ignored.

**disable\_bcp\_switch (name)**

Disables sending BCP switch commands when this switch changes state.

**Parameters name** – string name of the switch

**disable\_bcp\_switches (tag)**

Disables sending BCP switch commands when a switch with a certain tag changes state.

**Parameters tag** – string name of the tag for the switches you want to stop sending

**disable\_volume\_keys (up\_tag='volume\_up', down\_tag='volume\_down')**

Disables switch handlers so that the switches no longer affect the master system volume.

**Parameters**

- **up\_tag** – String of a switch tag name of the switches that will no longer be used to increase the volume.
- **down\_tag** – String of a switch tag name of the switches that will no longer be used to decrease the volume.

**enable\_bcp\_switch (name)**

Enables sending BCP switch commands when this switch changes state.

**Parameters name** – string name of the switch

**enable\_bcp\_switches (tag)**

Enables sending BCP switch commands when a switch with a certain tag changes state.

**Parameters tag** – string name of the tag for the switches you want to start sending

**enable\_volume\_keys (up\_tag='volume\_up', down\_tag='volume\_down')**

Enables switch handlers to change the master system volume based on switch tags.

**Parameters**

- **up\_tag** – String of a switch tag name that will be used to set which switch(es), when activated, increase the volume.

- **down\_tag** – String of a switch tag name that will be used to set which switch(es), when activated, decrease the volume.

**get\_bcp\_messages ()**

Retrieves and processes new BCP messages from the receiving queue.

**increase\_volume** (*track='master', \*\*kwargs*)

Sends a command to the remote BCP host to increase the volume of a track by 1 unit.

**Parameters**

- **track** – The string name of the track you want to increase the volume on. Default is 'master'.
- **\*\*kwargs** – Ignored. Included in case this method is used as a callback for an event which has other kwargs.

The max value of the volume for a track is set in the Volume: Steps: entry in the config file. If this increase causes the volume to go above the max value, the increase is ignored.

**process\_bcp\_events ()**

Processes the BCP Events from the config.

**register\_mpfmc\_trigger\_events** (*config, \*\*kwargs*)

Scans an MPF config file and creates trigger events for the config settings that need them.

**Parameters**

- **config** – An MPF config dictionary (can be the machine-wide or a mode- specific one).
- **\*\*kwargs** – Not used. Included to catch any additional kwargs that may be associated with this method being registered as an event handler.

**register\_triggers** (*config, priority=0, mode=None*)

Sets up trigger events based on a 'triggers:' section of a config dictionary.

**Parameters**

- **config** – A python config dictionary.
- **priority** – (not used) Included since this method is called as part of a mode start which passed this parameter.
- **mode** – (not used) Included since this method is called as part of a mode start which passed this parameter.

**remove\_bcp\_connection** (*bcp\_client*)

Removes a BCP connection to a remote BCP host.

**Parameters** **bcp\_client** – A reference to the BCPClientSocket instance you want to remove.

**send** (*bcp\_command, callback=None, \*\*kwargs*)

Sends a BCP message.

**Parameters**

- **bcp\_command** – String name of the BCP command that will be sent.
- **callback** – An optional callback method that will be called as soon as the BCP command is sent.
- **\*\*kwargs** – Optional kwarg pairs that will be sent as parameters along with the BCP command.

**Example**

If you call this method like this: `send('trigger', ball=1, string='hello')`

The BCP command that will be sent will be this: `trigger?ball=1&string=hello`

**set\_volume** (*volume*, *track*='master', *\*\*kwargs*)

Sends a command to the remote BCP host to set the volume of a track to the value specified.

**Parameters**

- **volume** – Int of the volume level. Valid range is 0 to the “steps” configuration in your config file. Values outside this range are ignored.
- **track** – The string name of the track you want to set the volume on. Default is ‘master’.
- **\*\*kwargs** – Ignored. Included in case this method is used as a callback for an event which has other kwargs.

**shutdown** ()

Prepares the BCP clients for MPF shutdown.

**class** `mpf.system.bcp.BCPClientSocket` (*machine*, *name*, *config*, *receive\_queue*)

Bases: `object`

Parent class for a BCP client socket. (There can be multiple of these to connect to multiple BCP media controllers simultaneously.)

**Parameters**

- **machine** – The main MachineController object.
- **name** – String name this client.
- **config** – A dictionary containing the configuration for this client.
- **receive\_queue** – The shared Queue() object that holds incoming BCP messages.

**create\_socket\_threads** ()

Creates and starts the sending and receiving threads for the BCP socket.

**Returns** True if the socket exists and the threads were started. False if not.

**get\_from\_socket** (*num\_bytes*=8192)

Reads and returns whatever data is sitting in the receiving socket.

**Parameters** **num\_bytes** – Int of the max number of bytes to read.

**Returns** The data in raw string format.

**receive\_goodbye** ()

Processes incoming BCP ‘goodbye’ command.

**receive\_hello** (*\*\*kwargs*)

Processes incoming BCP ‘hello’ command.

**receive\_loop** ()

Receive loop which reads incoming data, assembles commands, and puts them onto the receive queue.

This method is run as a thread.

**send** (*message*)

Sends a message to the BCP host.

**Parameters** **message** – String of the message to send.

**send\_goodbye()**

Sends BCP 'goodbye' command.

**send\_hello()**

Sends BCP 'hello' command.

**sending\_loop()**

Sending loop which transmits data from the sending queue to the remote socket.

This method is run as a thread.

**setup\_client\_socket()**

Sets up the client socket.

**stop()**

Stops and shuts down the socket client.

`mpf.system.bcp.decode_command_string(bcp_string)`

Decodes a BCP command string into separate command and paramter parts.

**Parameters** `bcp_string` – The incoming UTF-8, URL encoded BCP command string.

**Returns** A tuple of the command string and a dictionary of kwarg pairs.

### Example

Input: `trigger?name=hello&foo=Foo%20Bar` Output: `('trigger', {'name': 'hello', 'foo': 'Foo Bar'})`

Note that BCP commands and parameter names are not case-sensitive and will be converted to lowercase. Parameter values are case sensitive, and case will be preserved.

`mpf.system.bcp.encode_command_string(bcp_command, **kwargs)`

Encodes a BCP command and kwargs into a valid BCP command string.

**Parameters**

- **bcp\_command** – String of the BCP command name.
- **\*\*kwargs** – Optional pair(s) of kwargs which will be appended to the command.

**Returns** A string.

### Example

Input: `encode_command_string('trigger', {'name': 'hello', 'foo': 'Bar'})` Output: `trigger?name=hello&foo=Bar`

Note that BCP commands and parameter names are not case-sensitive and will be converted to lowercase. Parameter values are case sensitive, and case will be preserved.

## mpf.system.config module

Contains the Config class with utility configuration methods

**class** `mpf.system.config.CaseInsensitiveDict`

Bases: `dict`

A class based on Python's 'dict' class that internally stores all keys as lowercase. Set, get, contains, and del methods have been overwritten to automatically convert incoming calls to lowercase.



```
class mpf.system.config.Config(machine)
```

Bases: object

```
static check_config_file_version(file_location)
```

Checks a configuration file to see if it's the proper version for this version of MPF.

**Parameters** *file\_location* – The path to the file to check.

Returns: True if the config version of the file matches. False if not.

This method checks that the a string 'config\_version=x' exists in the first line of the file. If so, it checks that 'x' matches MPF's config\_version specification.

This check is done as integers.

```
check_for_invalid_sections(spec, config, validation_failure_info)
```

```
static dict_merge(a, b, combine_lists=True)
```

Recursively merges 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 event_config_to_dict(config)
```

```
static hexstring_to_int(inputstring, maxvalue=255)
```

Takes a string input of hex numbers and an integer.

#### Parameters

- **input\_string** – 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 hexstring\_to\_list** (*input\_string*, *output\_length*=3)

Takes a string input of hex numbers and returns 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 with zeros if the input string is too short.

**Returns** List of integers, like [255, 255, 0]

**static int\_to\_hex\_string** (*source\_int*)

Converts an int from 0-255 to a one-byte (2 chars) hex string, with uppercase characters.

**static keys\_to\_lower** (*source\_dict*)

Converts 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*)

Converts an incoming string or list into a list of lists.

**static load\_config\_yaml** (*config*=None, *yaml\_file*=None, *new\_config\_dict*=None)

Merges a new config dictionary into an existing one.

This method does what we call a “deep merge” which means it merges together subdictionaries instead of overwriting them. See the documentation for *meth:dict\_merge* for a description of how this works.

If the config dictionary you’re merging in also contains links to additional config files, it will also merge those in.

At this point this method loads YAML files, but it would be simple to load them from JSON, XML, INI, or existing python dictionaries.

**Parameters**

- **config** – The optional current version of the config dictionary that you’re building up. If you don’t pass a dictionary, this method will create one.
- **yaml\_file** – A YAML file containing the settings to deep merge into the config dictionary. This method will try to find a file with that name and open it to read in the settings. It will first try to open it as a file directly (including any path that’s there). If that doesn’t work, it will try to open the file using the last path that worked. (This path is stored in *config['config\_path']*.)
- **new\_config\_dict** – A dictionary of settings to merge into the config dictionary.

Note that you only need to specify a *yaml\_file* or *new\_config\_dictionary*, not both.

**Returns:** Python dictionary which is your source with all the new config options merged in.

**lookup\_invalid\_config\_setting** (*setting*)

**static normalize\_hex\_string** (*source\_hex*, *num\_chars*=2)

Takes an incoming hex value and converts 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 process\_config** (*config\_spec, source, target=None*)

**process\_config2** (*config\_spec, source, section\_name=None, target=None, result\_type='dict'*)

**static string\_to\_list** (*string*)

Converts 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*)

Converts a comma-separated and/or space-separated string into a Python list where 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 validate\_config\_item** (*spec, item='item not in config!@#'*)

**validate\_config\_item2** (*spec, validation\_failure\_info, item='item not in config!@#'*)

**validate\_item** (*item, validator, validation\_failure\_info*)

**validation\_error** (*item, validation\_failure\_info*)

## mpf.system.devices module

## mpf.system.events module

Contains the base classes for the EventManager and QueuedEvents

**class** `mpf.system.events.EventManager` (*machine, setup\_event\_player=True*)

Bases: `object`

**add\_handler** (*event, handler, priority=1, \*\*kwargs*)

Registers 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 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`.

For example: `handler_list.append(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(handler)`

**add\_monitor** (*monitor*)

Adds a new event monitor.

**Parameters monitor** – Reference to the callback function that will be called on every event posting.

Event monitors are similar to event handlers except they're called on every single event. In other words, they're like handlers you register for every event instead of a single event.

The monitor you register will be called on each event posting with the following paramters:

- **event** String name of the evnet
- **ev\_type** String of the type of event
- **callback** Reference to the event callback (if it has one)
- **kwargs** Dict of kwargs that will be passed to the handlers.

**does\_event\_exist** (*event\_name*)

Checks 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

**post** (*event, callback=None, \*\*kwargs*)

Posts 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 system 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\_boolean** (*event, callback=None, \*\*kwargs*)

Posts 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 system 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. (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** (*event, callback, \*\*kwargs*)

Posts 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 system 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** – 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\_relay** (*event, callback=None, \*\*kwargs*)

Posts 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 system 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.)

**process\_event\_player** (*config, mode=None, priority=0*)

**process\_random\_event\_player** (*config, mode=None, priority=0*)

**remove\_handler** (*method*)

Removes 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*)

Removes 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*)

Removes a registered event handler by key.

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

**remove\_handlers\_by\_keys** (*key\_list*)

Removes multiple event handlers based on a passed list of keys

**Parameters key\_list** – A list of keys of the handlers you want to remove

**remove\_monitor** (*monitor*)

Removes / deregisters an event monitor.

**Parameters monitor** – The function you want to deregister.

This method can safely be called even if this monitor is not registered.

**replace\_handler** (*event, handler, priority=1, \*\*kwargs*)

Checks 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 kwargs that will be registered with the new handler.

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

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

**unload\_event\_player\_events** (*event\_keys*)

**class** `mpf.system.events.QueuedEvent` (*callback, \*\*kwargs*)

Bases: `object`

The base class for an event queue which is created each time a queue event is called.

See the documentation at <http://missionpinball.com/docs/system-components/events/> for a description of how queue events work.

**clear** ()

**is\_empty** ()

**kill** ()

**wait** ()

## **mpf.system.light\_controller module**

Manages the light shows in a pinball machine.

**class** `mpf.system.light_controller.LightController` (*machine*)

Bases: `object`

Manages all the light shows in a pinball machine.

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

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

**add\_light\_player\_show** (*event, settings*)

**create\_show\_from\_script** (*script, lights=None, leds=None, light\_tags=None, led\_tags=None, key=None*)

Creates a light show from a script.

### **Parameters**

- **script** – Python dictionary in MPF light script format
- **lights** – String or iterable of multiples strings of the matrix lights that will be included in this show.
- **leds** – String or iterable of multiples strings of the LEDs that will be included in this show.
- **light\_tags** – String or iterable of multiples strings of tags of matrix lights that specify which lights will be in this show.
- **led\_tags** – String or iterable of multiples strings of tags of LEDs that specify which lights will be in this show.
- **key** – Object (typically string) that will be used to stop the show created by this list later.

**play\_show** (*show, priority=0, \*\*kwargs*)

Plays a light show.

### **Parameters**

- **show** – Either the string name of a registered show or a direct reference to the show object you want to play.
- **priority** – The priority this show will play at.
- **\*\*kwargs** – Contains the parameters and settings to control the playing of the show. See `Show.play()` for options and details.

**process\_light\_player** (*config, mode=None, priority=0*)

**process\_light\_scripts** (*config, mode=None, priority=0*)

**restore\_lower\_lights** (*show=None, priority=0*)

Restores the lights and LEDs from lower priority shows under this show.

This is only useful if this show is stopped, because otherwise this show will just immediately override these restored settings.

#### Parameters

- **show** – The show which will set the priority of the lights you want to restore.
- **priority** – An integer value of the lights you want to restore.

In both cases it will only restore lights below the priority you pass, skipping ones that are at the same value.

**run\_registered\_script** (*script\_name, \*\*kwargs*)

**run\_script** (*script, lights=None, leds=None, repeat=True, callback=None, key=None, \*\*kwargs*)

Runs a light script.

#### Parameters

- **script** – A list of dictionaries of script commands. (See below)
- **lights** – A light name or list of lights this script will be applied to.
- **leds** – An LED name or a list of LEDs this script will be applied to.
- **repeat** (*bool*) – Whether the script repeats (loops).
- **callback** – A method that will be called when this script stops.
- **key** – A key that can be used to later stop the light show this script creates. Typically a unique string. If it's not passed, it will either be the first light name or the first LED name.
- **\*\*kwargs** – Since this method just builds a Light Show, you can use any other Light Show attribute here as well, such as `ticks_per_sec`, `blend`, `repeat`, `num_repeats`, etc.

**Returns** `Show` (page 72) object. Since running a script just sets up and runs a regular Show, `run_script` returns the Show object. In most cases you won't need this, but it's nice if you want to know exactly which Show was created by this script so you can stop it later. (See the examples below for usage.)

Scripts are similar to Shows, except they only apply to single lights and you can “attach” any script to any light. Scripts are used anytime you want an light to have more than one action. A simple example would be a flash an light. You would make a script that turned it on (with your color), then off, repeating forever.

Scripts could be more complex, like cycling through multiple colors, blinking out secret messages in Morse code, etc.

Internally we actually just take a script and dynamically convert it into a Show (that just happens to only be for a single light), so we can have all the other Show-like features, including playback speed, repeats, blends, callbacks, etc.



The script is a list of dictionaries, with each list item being a sequential instruction, and the dictionary defining what you want to do at that step. Dictionary items for each step are:

color: The hex color for the light time: How long (in ms) you want the light to be at that color  
fade: True/False. Whether you want that light to fade to the color

(using the *time* above), or whether you want it to switch to that color instantly.

Example usage:

Here's how you would use the script to flash an RGB light between red and off:

```
self.flash_red = [] self.flash_red.append({"color": "ff0000", "tocks": 1})
self.flash_red.append({"color": "000000", "tocks": 1})
self.machine.show_controller.run_script(script=self.flash_red,
lights='light1', priority=4, blend=True)
```

Once the "flash\_red" script is defined as self.flash\_red, you can use it anytime for any light or LED. You can also define lights as a list, like this:

```
self.machine.show_controller.run_script(script=self.flash_red, lights=['light1', 'light2'],
priority=4, blend=True)
```

Most likely you would define your scripts once when the game loads and then call them as needed.

You can also make more complex scripts. For example, here's a script which smoothly cycles an RGB light through all colors of the rainbow:

```
self.rainbow = [] self.rainbow.append({'color': 'ff0000', 'tocks': 1, 'fade': True})
self.rainbow.append({'color': 'ff7700', 'tocks': 1, 'fade': True}) self.rainbow.append({'color': 'ffcc00', 'tocks': 1, 'fade': True})
self.rainbow.append({'color': '00ff00', 'tocks': 1, 'fade': True}) self.rainbow.append({'color': '0000ff', 'tocks': 1, 'fade': True})
self.rainbow.append({'color': 'ff00ff', 'tocks': 1, 'fade': True})
```

If you have single color lights, your *color* entries in your script would only contain a single hex value for the intensity of that light. For example, a script to flash a single-color light on-and-off (which you can apply to any light):

```
self.flash = [] self.flash.append({"color": "ff", "tocks": 1}) self.flash.append({"color": "00", "tocks": 1})
```

If you'd like to save a reference to the [Show](#) (page 72) that's created by this script, call it like this:

```
self.blah = self.machine.show_controller.run_script("light2", self.flash_red, "4",
tocks_per_sec=2)
```

**stop\_script** (*key*, *\*\*kwargs*)

Stops and removes the light show that was created by a light script.

#### Parameters

- **key** – The key that was specified in run\_script().
- **\*\*kwargs** – Not used, included in case this method is called via an event handler that might contain other random parameters.

**stop\_show** (*show=None*, *key=None*, *\*\*kwargs*)

**stop\_shows\_by\_key** (*key*)

**stop\_shows\_by\_keys** (*keys*)

**sync\_ms\_next\_tick** (*sync\_ms*)

Figures out the next tick show should start based on the passed sync\_ms value.

**Parameters** `sync_ms` – Integer of the sync period in ms.

**Returns** Int of a tick number

**unload\_light\_player\_shows** (*removal\_tuple*)

**class** `mpf.system.light_controller.Playlist` (*machine*)

Bases: `object`

A list of [Show](#) (page 72) objects which are then played sequentially.

Playlists are useful for things like attract mode where you play one show for a few seconds, then another, etc.

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

Each step in a playlist can contain more than one [Show](#) (page 72). This is useful if you have a lot of little shows for different areas of the playfield that you want run at the same time. For example, you might have one show that only controls a group of rollover lane lights, and another which blinks the lights in the center of the playfield. You can run them at the by putting them in the same step in your playlist. (Note you don't need to use a playlist if you simply want to run two Shows at the same time. In that case you could just call `Show.play()` (page 72) twice to play both shows.

For each “step” in the playlist, you can specify the number of seconds it runs those shows before moving on, or you can specify that one of the shows in that step plays a certain number of times and then the playlist moves to the next step from there.

You create a show by creating an instance `Playlist` (page 70). Then you add Shows to it via `add_show()` (page 70). Finally, you specify the settings for each step (like how it knows when to move on) via `:meth:step_settings`.

When you start a playlist (via `start()` (page 71), you can specify settings like what priority the show runs at, whether it repeats, etc.)

Example usage from a game mode: (This example assumes we have `self.show1`, `self.show2`, and `self.show3` already loaded.)

Setup the playlist:

```
self.my_playlist = lights.Playlist(self.machine)
self.my_playlist.add_show(step_num=1, show=self.show1, tocks_per_sec=10)
self.my_playlist.add_show(step_num=2, show=self.show2, tocks_per_sec=5)
self.my_playlist.add_show(step_num=3, show=self.show3, tocks_per_sec=32)
self.my_playlist.step_settings(step=1, time=5)
self.my_playlist.step_settings(step=2, time=5)
self.my_playlist.step_settings(step=3, time=5)
```

Run the playlist:

```
self.my_playlist.start(priority=100, repeat=True)
```

Stop the playlist:

```
self.my_playlist.stop()
```

**add\_show** (*step\_num*, *show*, *num\_repeats=0*, *tocks\_per\_sec=32*, *blend=False*, *repeat=True*)

Adds a Show to this playlist. You have to add at least one show before you start playing the playlist.

**Parameters**

- **step\_num** – Interger of which step number you're adding this show to. You have to specify this since it's possible to add multiple shows to the same step (in cases where you want them both to play at the same time during that step). If you want the same show to play in multiple steps, then add it multiple times (once to each step). The show plays starting

with the lowest number step and then moving on. Ideally they'd be 1, 2, 3... but it doesn't matter. If you have step numbers of 1, 2, 5... then the player will figure it out.

- **show** – The Show object that you're adding to this step.
- **num\_repeats** – Integer of how many times you want this show to repeat within this step. Note this does not affect when the playlist advances to the next step. (That is controlled via `step_settings()` (page 71).) Rather, this is just how many loops this show plays. A value of 0 means it repeats indefinitely. (Well, until the playlist advances to the next step.) Note that you also have to have `repeat=True` for it to repeat here.
- **tocks\_per\_sec** – Integer of how fast you want this show to play. See `Show.play()` (page 72) for details.
- **blend** – Boolean of whether you want this show to blend with lower priority shows below it. See `Show.play()` (page 72) for details.
- **repeat** – Boolean which causes the show to keep repeating until the playlist moves on to the next step.

**start** (*priority=0, repeat=True, repeat\_count=0, reset=True*)

Starts playing a playlist. You can only use this after you've added at least one show via `add_show()` (page 70) and configured the settings for each step via `step_settings()` (page 71).

Args

**priority:** Integer of what priority you want the **Show** (page 72) shows in this playlist to play at. These shows will play "on top" of lower priority stuff, but "under" higher priority things.

**repeat:** Controls whether this playlist to repeats when it's finished. **repeat\_count:** How many times you want this playlist to

repeat before it stops itself. (Must be used with `repeat=True` above.) A value of 0 here means that this playlist repeats forever until you manually stop it. (This is ideal for attract mode.)

**reset:** Boolean which controls whether you want this playlist to start at the beginning (True) or you want it to pick up where it left off (False). You can also use `reset` to restart a playlist that's currently running.

**step\_settings** (*step, time=0, trigger\_show=None, hold=False*)

Used to configure the settings for a step in a **Playlist** (page 70). This configuration is required for each step. The main thing you use this for is to specify how the playlist knows to move on to the next step.

Args:

**step:** Integer for which step number you're configuring time: Integer of the time in seconds that you want this step to run

before moving on to the next one.

**trigger\_show:** If you want to move to the next step after one of the Shows in this step is done playing, pass that show's object here. This is required because if there are multiple Shows in this step of the playlist which all end at different times, we wouldn't know which one to watch in order to know when to move on.

Note that you can have repeats with a trigger show, but in that case you also need to have the `num_repeats` specified. Otherwise if you have your trigger show repeating forever then the playlist will never move on. (In that case use the `time` parameter to move on based on time.)

**stop** (*reset=True, hold=None*)

Stops a playlist. Pretty simple.

### Parameters

- **reset** – If *True*, it resets the playlist tracking counter back to the beginning. You can use *False* here if you want to stop and then restart a playlist to pick up where it left off.
- **hold** – Boolean which specifies whether this playlist should hold the lights and LEDs in their current states. Default is *None* which means it inherits whatever the shows or playlist settings were, but you can force it *True* or *False* if you want here.

**class** `mpf.system.light_controller.Show`(*machine*, *config*, *file\_name*, *asset\_manager*, *actions=None*)

Bases: `mpf.system.assets.Asset` (page 52)

**add\_loaded\_callback** (*loaded\_callback*, *\*\*kwargs*)

**advance** ()

**change\_speed** (*ticks\_per\_sec=1*)

Changes the playback speed of a running Show.

**Parameters** *ticks\_per\_sec* – The new ticks\_per\_second play rate.

If you want to change the playback speed by a percentage, you can access the current *ticks\_per\_second* rate via *Show*’s *ticks\_per\_second* variable. So if you want to double the playback speed of your show, you could do something like:

```
self.your_show.change_speed(self.your_show.ticks_per_second*2)
```

Note that you can’t just update the show’s *ticks\_per\_second* directly because we also need to update *self.ticks\_per\_tock*.

**do\_load** (*callback*, *show\_actions=None*)

**load\_show\_from\_disk** ()

**play** (*repeat=False*, *priority=0*, *blend=False*, *hold=None*, *ticks\_per\_sec=30*, *start\_location=None*, *callback=None*, *num\_repeats=0*, *sync\_ms=0*, *reset=True*, *\*\*kwargs*)

Plays a Show. There are many parameters you can use here which affect how the show is played. This includes things like the playback speed, priority, whether this show blends with others, etc. These are all set when the show plays. (For example, you could have a Show file which lights a bunch of lights sequentially in a circle pattern, but you can have that circle “spin” as fast as you want depending on how you play the show.)

### Parameters

- **repeat** – Boolean of whether the show repeats when it’s done.
- **priority** – Integer value of the relative priority of this show. If there’s ever a situation where multiple shows want to control the same item, the one with the higher priority will win. (“Higher” means a bigger number, so a show with priority 2 will override a priority 1.)
- **blend** – Boolean which controls whether this show “blends” with lower priority shows and scripts. For example, if this show turns a light off, but a lower priority show has that light set to blue, then the light will “show through” as blue while it’s off here. If you don’t want that behavior, set *blend* to be *False*. Then off here will be off for sure (unless there’s a higher priority show or command that turns the light on). Note that not all item types blend. (You can’t blend a coil or event, for example.)
- **hold** – Boolean which controls whether the lights or LEDs remain in their final show state when the show ends. Default is *None* which means *hold* will be *False* if the show has more than one step, and *True* if there is only one step.

- **ticks\_per\_sec** – Integer of how fast your show runs (“Playback speed,” in other words.) Your Show files specify action times in terms of ‘ticks’, like “make this light red for 3 ticks, then off for 4 ticks, then a different light on for 6 ticks. When you play a show, you specify how many ticks per second you want it to play. Default is 30, but you might even want ticks\_per\_sec of only 1 or 2 if your show doesn’t need to move than fast. Note this does not affect fade rates. So you can have ticks\_per\_sec of 1 but still have lights fade on and off at whatever rate you want. Also the term “ticks” was chosen so as not to confuse it with “ticks” which is used by the machine run loop.
- **start\_location** – Integer of which position in the show file the show should start in. Usually this is 0 but it’s nice to start part way through. Also used for restarting shows that you paused. A negative value will count backwards from the end (-1 is the last position, -2 is second to last, etc.).
- **callback** – A callback function that is invoked when the show is stopped.
- **num\_repeats** – Integer of how many times you want this show to repeat before stopping. A value of 0 means that it repeats indefinitely. Note this only works if you also have repeat=True.
- **sync\_ms** – Number of ms of the show sync cycle. A value of zero means this show will also start playing immediately. See the full MPF documentation for details on how this works.
- **reset** – Boolean which controls whether this show will reset to its first position once it ends. Default is True.
- **\*\*kwargs** – Not used, but included in case this method is used as an event handler which might include additional kwargs.

#### **resync ()**

Causes this show to do a one-time update to resync all the LEDs and lights in the show with where they should be now. This is used when a higher priority show stops so lower priority shows can put all the lights back to how they want them.

#### **stop (reset=True, hold=None)**

Stops the Light Show.

Note you can also use this method to clear a stopped show’s held lights and LEDs by passing hold=False.

#### **Parameters**

- **reset** – Boolean which controls whether the show will reset its current position back to zero. Default is True.
- **hold** – Boolean which controls whether the show will hold its current lights and LEDs in whatever state they are now, including their priorities. Default is None which will just use whatever the show setting was when you played it, but you can force it to hold or not with True or False here.

## **mpf.system.logic\_blocks module**

MPF plugin which implements Logic Blocks

**class** `mpf.system.logic_blocks.Accrual` (*machine, name, player, config*)

Bases: `mpf.system.logic_blocks.LogicBlock` (page 74)

A type of LogicBlock which tracks many different events (steps) towards a goal, with the steps being able to happen in any order.

**enable** (*\*\*kwargs*)

Enables this accrual. Automatically called when one of the 'enable\_events' is posted. Can also manually be called.

**hit** (*step, \*\*kwargs*)

Increases the hit progress towards completion. Automatically called when one of the *count\_events* is posted. Can also manually be called.

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

**reset** (*\*\*kwargs*)

Resets the hit progress towards completion

**class** `mpf.system.logic_blocks.Counter` (*machine, name, player, config*)

Bases: `mpf.system.logic_blocks.LogicBlock` (page 74)

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

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

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

**enable** (*\*\*kwargs*)

Enables this counter. Automatically called when one of the 'enable\_event's is posted. Can also manually be called.

**hit** (*\*\*kwargs*)

Increases the hit progress towards completion. Automatically called when one of the 'count\_events's is posted. Can also manually be called.

**reset** (*\*\*kwargs*)

Resets the hit progress towards completion

**stop\_ignoring\_hits** (*\*\*kwargs*)

Causes the Counter to stop ignoring subsequent hits that occur within the 'multiple\_hit\_window'. Automatically called when the window time expires. Can safely be manually called.

**class** `mpf.system.logic_blocks.LogicBlock` (*machine, name, player, config*)

Bases: `object`

Parent class for each of the logic block classes.

**complete** ()

Marks this logic block as complete. Posts the 'events\_when\_complete' events and optionally restarts this logic block or disables it, depending on this block's configuration settings.

**create\_control\_events** ()

**disable** (*\*\*kwargs*)

Disables this logic block. Automatically called when one of the *disable\_event* events is posted. Can also manually be called.

**enable** (*\*\*kwargs*)

Enables this logic block. Automatically called when one of the *enable\_event* events is posted. Can also manually be called.

**reset** (*\*\*kwargs*)

Resets the progress towards completion of this logic block. Automatically called when one of the *reset\_event* events is called. Can also be manually called.

**restart** (*\*\*kwargs*)

Restarts this logic block by calling `reset()` and `enable()`. Automatically called when one of the `restart_event` events is called. Can also be manually called.

**unload** ()

**class** `mpf.system.logic_blocks.LogicBlocks` (*machine*)

Bases: `object`

LogicBlock Manager.

**class** `mpf.system.logic_blocks.Sequence` (*machine, name, player, config*)

Bases: `mpf.system.logic_blocks.LogicBlock` (page 74)

A type of `LogicBlock` which tracks many different events (steps) towards a goal, with the steps having to happen in order.

**enable** (*step=0, \*\*kwargs*)

Enables this Sequence. Automatically called when one of the ‘`enable_events`’ is posted. Can also manually be called.

**Parameters** **step** – Step number this logic block will be at when it’s enabled. Default is 0.

Note the step numbers are zero-based.

**hit** (*\*\*kwargs*)

Increases the hit progress towards completion. Automatically called when one of the `count_events` is posted. Can also manually be called.

**reset** (*\*\*kwargs*)

Resets the sequence back to the first step.

## mpf.system.machine module

The main machine object for the Mission Pinball Framework.

**class** `mpf.system.machine.MachineController` (*options*)

Bases: `object`

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**

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

**config**

A dictionary of machine’s configuration settings, merged from various sources.

**physical\_hw**

Boolean as to whether there is physical pinball controller hardware attached.

**done**

Boolean. Set to True and MPF exits.

**machineflow\_index**

What machineflow position the machine is currently in.

**machine\_path**

The root path of this machine\_files folder

**display**

**plugins**

**scriptlets**

**tilted**

**platform**

**events**

**add\_platform** (*name*)

Makes 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).

**configure\_debugger** ()

**get\_debug\_status** (*debug\_path*)

**log\_loop\_rate** ()

**log\_system\_info** ()

Dumps information about the Python installation to the log.

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

**power\_off** ()

Attempts to perform a power down of the pinball machine and ends MPF.

This method is not yet implemented.

**quit** ()

Performs a graceful exit of MPF.

**register\_monitor** (*monitor\_class*, *monitor*)

Registers 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.

**reset** ()

Resets 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.

Note: This method is not yet implemented.

**run** ()

Starts the main machine run loop.



**set\_default\_platform** (*name*)

Sets the default platform which is used if a device class-specific or device-specific platform is not specified. The default platform also controls whether a platform timer or MPF's timer is used.

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

**string\_to\_class** (*class\_string*)

Converts a string like `mpf.system.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>

**timer\_tick** ()

Called to “tick” MPF at a rate specified by the machine Hz setting.

This method is called by the MPF run loop or the platform run loop, depending on the platform. (Some platforms drive the loop, and others let MPF drive.)

**mpf.system.machine\_mode module****mpf.system.modes module****mpf.system.platform module**

Contains the parent classes Platform

**class** `mpf.system.platform.Platform` (*machine*)

Bases: `object`

Parent class for the a hardware platform interface.

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

This is the class that each hardware controller (such as P-ROC or FAST) will subclass to talk to their hardware.

**clear\_hw\_rule** (*sw\_name*)

Subclass this method in a platform module to clear a hardware switch rule for this switch.

Clearing a hardware rule means actions on this switch will no longer affect coils.

Another way to think of this is that it ‘disables’ a hardware rule. This is what you’d use to disable flippers and autofire\_coils during tilt, game over, etc.

**configure\_dmd** ()

Subclass this method in a platform module to configure the DMD.

This method should return a reference to the DMD’s platform interface object which will be called to access the hardware.

**configure\_driver** (*config*, *device\_type*=‘coil’)

Subclass this method in a platform module to configure a driver.

This method should return a reference to the driver’s platform interface object which will be called to access the hardware.

**configure\_gi** (*config*)

Subclass this method in a platform module to configure a GI string.

This method should return a reference to the GI string's platform interface object which will be called to access the hardware.

**configure\_led** (*config*)

Subclass this method in a platform module to configure an LED.

This method should return a reference to the LED's platform interface object which will be called to access the hardware.

**configure\_matrixlight** (*config*)

Subclass this method in a platform module to configure a matrix light.

This method should return a reference to the matrix lights's platform interface object which will be called to access the hardware.

**configure\_switch** (*config*)

Subclass this method in a platform module to configure a switch.

This method should return a reference to the switch's platform interface object which will be called to access the hardware.

**get\_hw\_switch\_states** ()

Subclass this method in a platform module to return the hardware states of all the switches on that platform. of a switch.

This method should return a dict with the switch numbers as keys and the hardware state of the switches as values. (0 = inactive, 1 = active) This method should not compensate for NO or NC status, rather, it should return the raw hardware states of the switches.

**run\_loop** ()

Subclass this method in a platform module if the platform will control the run loop rather than MPF controlling it.

If you want to use this method and let your platform control the machine's run loop, set *self.features['hw\_timer'] = True*.

If your platform controls the loop, it should call *self.machine.timer\_tick()* periodically based on the *self.machine.HZ* rate.

Also the loop should continue running until *self.machine.done* is True. For example, it could run in *while not self.machine.done: loop*.

Your loop can call *self.machine.switch\_controller.process\_switch()* if any switch events come in "off cycle", but the *timer\_tick* should be called consistently.

This loop can safely block. If the call to the hardware does not block, there should be a small pause in the loop (e.g. *time.sleep(.001)*) to prevent 100% CPU utilization.)

**set\_hw\_rule** (*sw\_name*, *sw\_activity*, *coil\_name=None*, *coil\_action\_ms=0*, *pulse\_ms=0*, *pwm\_on=0*, *pwm\_off=0*, *delay=0*, *recycle\_time=0*, *debounced=False*, *drive\_now=False*)

Writes a hardware rule to the controller.

**Parameters**

- **sw\_name** – String name of the switch.
- **sw\_activity** – Int representing the switch state this rule will be set for. 1 is active, 0 is inactive.
- **coil\_name** – String name of the coil.
- **coil\_action\_ms** – Total time in ms the coil should activate for.
- **pulse\_ms** – How long in ms the coil should activate for. Default is 0.

- **pwn\_on** – The ‘on’ portion, in ms of a pwm-based patter. Default is 0.
- **pwm\_off** – The ‘off’ portion, in ms, of a pwm-based patter. Default is 0.
- **delay** – The delay, in ms, the coil should wait before firing. Default is 0.
- **recycle\_time** – How long the coil must be inactive, in ms, before it can be fired again via this rule. Default is 0.
- **debounced** – Boolean which specifies whether this coil should activate on a debounced or non-debounced switch change state. Default is False (non-debounced).
- **drive\_name** – Boolean which controls whether the coil should activate immediately when this rule is applied if the switch currently in in the state set in this rule.

Note that this method provides several convenience processing to convert the incoming parameters into a format that is more widely-used by hardware controls. It’s intended that platform interfaces subclass `write_hw_rule()` instead of this method, though this method may be subclassed if you wish.

#### **tick()**

Subclass this method in a platform module to perform periodic updates to the platform hardware, e.g. reading switches, sending driver or light updates, etc.

If you want to use this method and let MPF control the machine’s run loop, set `self.features['hw_timer'] = False`.

This method is only used when MPF controls the game loop. Each platform interface either needs to implement this method or the `run_loop` method.

This method will be called every 1ms.

#### **timer\_initialize()**

Run this before the machine loop starts. I want to do it here so we don’t need to check for initialization on each machine loop. (Or is this premature optimization?)

#### **write\_hw\_rule(*sw, sw\_activity, coil\_action\_ms, coil, pulse\_ms, pwm\_on, pwm\_off, delay, recycle\_time, debounced, drive\_now*)**

Subclass this method in a platform interface to write a hardware switch rule to the controller.

Game programmers will typically use `set_hw_rule` instead of this method because `set_hw_rule` takes switch NC and NO settings into account, so it’s a bit more convenient.

## **mpf.system.scoring module**

MPF plugin for a score controller which handles all scoring and bonus tracking.

**class** `mpf.system.scoring.ScoreController(machine)`

Bases: `object`

**add(*points, force=False*)**

Adds to the current player’s score.

Use this method instead of changing the value of the player attribute directly because this method will post the scoring events that other modules use for effects and stuff.

**Parameters** **points** – Integer of points to add to the current player’s score. Note this value can also be negative to subtract points from their score.

**process\_config(*config, mode=None, priority=0*)**

**register\_score\_event(*event\_name, points, priority=0, block=False*)**

Used to register a score event which adds to a player’s score when a certain event is posted.

**Parameters**

- **event\_name** – The string name of the event that should cause a score change to take place.
- **points** – The integer number of points that should be added or subtracted to the current player’s score when this event is posted.
- **priority** – Integer priority which is used in conjunction with block.
- **block** – Boolean which specifies whether this event should block lower priority events. If True, lower priority events will not score as long as this event is registered. If False then lower priority events will score as normal.

**Returns:** A “key” which can be used to later unregister this event via the ‘unregister\_score\_event’ method.

**unload\_score\_events** (*key\_list*)

Unloads and removes several score events at once.

**Parameters** *key\_list* – A list of keys of the score events you want to remove.

**unregister\_score\_event** (*score\_entry\_key*)

Removes a score event.

**Parameters** *score\_entry\_key* – The key of the score event to remove. This is the key that’s returned by the ‘register\_score\_event()’ method.

**mpf.system.scriptlet module**

Contains the parent class for Scriptlets.

**class** `mpf.system.scriptlet.Scriptlet` (*machine, name*)

Bases: `object`

**on\_load** ()

Automatically called when this Scriptlet loads. It’s the intention that the Scriptlet writer will overwrite this method in the Scriptlet.

**mpf.system.shots module****mpf.system.switch\_controller module**

Contains the SwitchController class which is responsible for reading switch states and posting events to the framework.

**class** `mpf.system.switch_controller.SwitchController` (*machine*)

Bases: `object`

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://missionpinball.com/docs/system-components/switch-controller/>

**add\_monitor** (*monitor*)

**add\_switch\_handler** (*switch\_name, callback, state=1, ms=0, return\_info=False, call-back\_kwargs=None*)

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 nearest 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.

**is\_active** (*switch\_name*, *ms=None*)

Queries 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*)

Queries 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*)

Queries 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** = <logging.Logger object at 0x1049b4f90>

**log\_active\_switches** ()

Writes 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*)

Returns the number of ms that have elapsed since this switch last changed state.

**process\_switch** (*name=None*, *state=1*, *logical=False*, *num=None*, *obj=None*, *debounced=True*)

Processes a new switch state change.

#### Parameters

- **name** – The string name of the switch. This is optional if you specify the switch via the ‘num’ or ‘obj’ parameters.

- **state** – Boolean or int of state of the switch you’re processing, True/1 is active, False/0 is inactive.
- **logical** – Boolean which specifies whether the ‘state’ argument represents the “physical” or “logical” state of the switch. If True, a 1 means this switch is active and a 0 means it’s inactive, regardless of the NC/NO configuration of the switch. If False, then the state parameter passed will be inverted if the switch is configured to be an ‘NC’ type. Typically the hardware will send switch states in their raw (logical=False) states, but other interfaces like the keyboard and OSC will use logical=True.
- **num** – The hardware number of the switch.
- **obj** – The switch object.
- **debounced** – Whether or not the update for the switch you’re sending has been debounced or not. Default is True

Note that there are three different parameter options to specify the switch: ‘name’, ‘num’, and ‘obj’. You only need to pass one of them.

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’.)

**remove\_switch\_handler** (*switch\_name, callback, state=1, ms=0*)

Removes 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.

**secs\_since\_change** (*switch\_name*)

Returns the number of ms that have elapsed since this switch last changed state.

**set\_state** (*switch\_name, state=1, reset\_time=False*)

Sets the state of a switch.

**update\_switches\_from\_hw** ()

Updates the states of all the switches by 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** ()

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

Throws logging warnings if anything doesn’t match.

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

## **mpf.system.target\_controller module**

## **mpf.system.tasks module**

**class** `mpf.system.tasks.DelayManager`

Bases: `object`

Parent class for a delay manager which can manage multiple delays.

**add** (*name, ms, callback, \*\*kwargs*)

Adds a delay.

**Parameters**

- **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.
- **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.
- **\*\*kwargs** – Any other (optional) kwarg pairs you pass will be passed along as kwargs to the callback method.

**check** (*delay*)

Checks 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** ()

Removes (clears) all the delays associated with this DelayManager.

**dead\_delay\_managers** = set([])

**delay\_managers** = set([])

**remove** (*name*)

Removes a delay. (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** (*name, ms, callback, \*\*kwargs*)

Resets a delay, first deleting the old one (if it exists) and then adding new delay with the new settings.

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

**static timer\_tick** ()

**class** `mpf.system.tasks.Task` (*callback, args=None, name=None, sleep=0*)

Bases: `object`

A task/coroutine implementation.

Tasks are similar to timers except they can yield back to the main loop at any point, then be resumed later.

To wait from a Task, do *yield <ms>*, e.g. *yield 200*.

To exit from a Task, just return. This will raise a `StopIteration` exception which the scheduler will catch and remove the task from the run queue.

**NewTasks** = set([])

**Tasks** = set([])

**static create** (*callback, args=(), sleep=0*)

Creates a new task and insert it into the runnable set.

**restart ()**

Restarts the task.

**stop ()**

Stops the task.

This causes it not to run any longer, by removing it from the task set and then deleting it.

**static timer\_tick ()**

Scans all tasks now and run those that are ready.

## **mpf.system.timing module**

Contains Timing and Timer classes

**class** `mpf.system.timing.Timer (callback, args=(), frequency=None)`

Bases: `object`

Periodic timer object.

A timer defines a callable plus a frequency (in sec) at which it should be called. The frequency can be set to None so that the timer is not enabled, but it still exists.

### **Parameters**

- **callback** (*method*) – The method you want called each time this timer is fired.
- **args** (*tuple*) – Arguments you want to pass to the callback.
- **frequency** (*int or float*) – How often, in seconds, you want this timer
- **be called.** (*to*) –

**call ()**

**class** `mpf.system.timing.Timing (machine)`

Bases: `object`

System timing object.

This object manages timing for the whole system. Only one of these objects should exist. By convention it is called 'timing'.

The timing keeps the current time in 'time' and a set of Timer objects.

**HZ = None**

**add (timer)**

**static** `int_to_pwm (ratio, length)`

Converts a decimal between 0 and 1 to a pwm mask of whatever length you want.

For example, an input ratio of .5 with a result length of 8 returns 10101010. And input ratio of .7 with a result length of 32 returns 1101101110110110110110110110110.

Another way to think about this is this method converts a decimal percentage into the corresponding pwm mask.

### **Parameters**

- **ratio** (*float*) – A value between 0 and 1 that you want to convert.
- **length** (*int*) – How many digits you want in your result.

**ms\_per\_tick = None**



**static** `pwm_ms_to_byte_int` (*pwm\_on*, *pwm\_off*)

Converts a `pwm_on` / `pwm_off` ms times to a single byte pwm mask.

**remove** (*timer*)

**static** `secs` (*s*)

**secs\_per\_tick** = None

**static** `string_to_ms` (*time\_string*)

Decodes 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*)

Decodes a string of real-world time into an float of seconds.

See ‘string\_to\_ms’ for a description of the time string.

**static** `string_to_ticks` (*time\_string*)

Converts a string of real-world time into a float of how many machine ticks correspond to that amount of time.

See ‘string\_to\_ms’ for a description of the time string.

**tick** = 0

**timer\_tick** ()

## 1.6.2 Module contents



## MODULE CONTENTS



- .
- mpf.devices, 20
- mpf.devices.autofire, 1
- mpf.devices.ball\_device, 1
- mpf.devices.diverter, 3
- mpf.devices.driver, 5
- mpf.devices.drop\_target, 6
- mpf.devices.flasher, 7
- mpf.devices.flipper, 7
- mpf.devices.gi, 9
- mpf.devices.led, 10
- mpf.devices.matrix\_light, 11
- mpf.devices.new\_device\_template, 12
- mpf.devices.playfield, 12
- mpf.devices.playfield\_transfer, 14
- mpf.devices.score\_reel, 14
- mpf.devices.switch, 19
- mpf.media\_controller, 32
- mpf.media\_controller.core, 31
- mpf.media\_controller.core.keyboard, 20
- mpf.media\_controller.core.language, 21
- mpf.media\_controller.core.show\_controller, 22
- mpf.media\_controller.core.slide, 24
- mpf.media\_controller.core.slide\_builder, 27
- mpf.media\_controller.core.sound, 27
- mpf.media\_controller.decorators, 31
- mpf.media\_controller.display\_modules, 31
- mpf.media\_controller.elements, 32
- mpf.media\_controller.transitions, 32
- mpf.platform, 48
- mpf.platform.fast, 32
- mpf.platform.p3\_roc, 36
- mpf.platform.p\_roc, 41
- mpf.platform.virtual, 46
- mpf.plugins, 52
- mpf.plugins.auditor, 48
- mpf.plugins.ball\_save, 49
- mpf.plugins.ball\_search, 49
- mpf.plugins.info\_lights, 50
- mpf.plugins.osc, 50
- mpf.plugins.socket\_events, 51
- mpf.plugins.switch\_player, 52
- mpf.system, 85
- mpf.system.assets, 52
- mpf.system.ball\_controller, 55
- mpf.system.bcp, 56
- mpf.system.config, 60
- mpf.system.events, 63
- mpf.system.light\_controller, 67
- mpf.system.logic\_blocks, 73
- mpf.system.machine, 75
- mpf.system.platform, 77
- mpf.system.scoring, 79
- mpf.system.scriptlet, 80
- mpf.system.switch\_controller, 80
- mpf.system.tasks, 82
- mpf.system.timing, 84

**m**

- mpf, 87



## A

- Accrual (class in mpf.system.logic\_blocks), 73
- activate() (mpf.devices.diverter.Diverter method), 4
- active (mpf.media\_controller.core.slide.Slide attribute), 24
- add() (mpf.system.scoring.ScoreController method), 79
- add() (mpf.system.tasks.DelayManager method), 82
- add() (mpf.system.timing.Timing method), 84
- add\_ball() (mpf.devices.playfield.Playfield method), 12
- add\_element() (mpf.media\_controller.core.slide.Slide method), 25
- add\_handler() (mpf.devices.gi.GI method), 9
- add\_handler() (mpf.devices.matrix\_light.MatrixLight method), 11
- add\_handler() (mpf.system.events.EventManager method), 63
- add\_key\_map() (mpf.media\_controller.core.keyboard.Keyboard method), 20
- add\_light\_player\_show() (mpf.system.light\_controller.LightController method), 67
- add\_loaded\_callback() (mpf.media\_controller.core.show\_controller.Show method), 22
- add\_loaded\_callback() (mpf.system.light\_controller.Show method), 72
- add\_monitor() (mpf.system.events.EventManager method), 64
- add\_monitor() (mpf.system.switch\_controller.SwitchController method), 80
- add\_platform() (mpf.system.machine.MachineController method), 76
- add\_ready\_callback() (mpf.media\_controller.core.slide.Slide method), 25
- add\_show() (mpf.system.light\_controller.Playlist method), 70
- add\_show\_player\_show() (mpf.media\_controller.core.show\_controller.Show method), 24
- add\_switch\_handler() (mpf.system.switch\_controller.SwitchController method), 80
- add\_to\_bank() (mpf.devices.drop\_target.DropTarget method), 6
- add\_value() (mpf.devices.score\_reel.ScoreReelGroup method), 17
- advance() (mpf.devices.score\_reel.ScoreReel method), 15
- advance() (mpf.media\_controller.core.show\_controller.Show method), 22
- advance() (mpf.system.light\_controller.Show method), 72
- aliases (mpf.platform.p3\_roc.PDBConfig attribute), 39
- aliases (mpf.platform.p\_roc.PDBConfig attribute), 44
- are\_balls\_gathered() (mpf.system.ball\_controller.BallController method), 55
- Asset (class in mpf.system.assets), 52
- asset\_class (in module mpf.media\_controller.core.sound), 31
- AssetLoader (class in mpf.system.assets), 52
- AssetManager (class in mpf.system.assets), 52
- assumed\_value\_int (mpf.devices.score\_reel.ScoreReelGroup attribute), 18
- assumed\_value\_list (mpf.devices.score\_reel.ScoreReelGroup attribute), 18
- audit() (mpf.plugins.auditor.Auditor method), 48
- audit\_event() (mpf.plugins.auditor.Auditor method), 48
- audit\_player() (mpf.plugins.auditor.Auditor method), 48
- audit\_shot() (mpf.plugins.auditor.Auditor method), 48
- audit\_switch() (mpf.plugins.auditor.Auditor method), 48
- Auditor (class in mpf.plugins.auditor), 48
- AutofireCoil (class in mpf.devices.autofire), 1

## B

- ball\_drain() (mpf.plugins.ball\_save.BallSave method), 49
- ball\_found() (mpf.devices.playfield.Playfield method), 13
- ball\_lost() (mpf.devices.playfield.Playfield method), 13
- ball\_search\_begin() (mpf.devices.playfield.Playfield method), 13
- ball\_search\_disable() (mpf.devices.playfield.Playfield method), 13
- ball\_search\_end() (mpf.devices.playfield.Playfield method), 13
- ball\_search\_failed() (mpf.devices.playfield.Playfield method), 13
- ball\_search\_schedule() (mpf.devices.playfield.Playfield method), 13
- ball\_started() (mpf.plugins.info\_lights.InfoLights method), 50

- BallController (class in mpf.system.ball\_controller), 55
- BallDevice (class in mpf.devices.ball\_device), 1
- balls (mpf.devices.playfield.Playfield attribute), 13
- balls (mpf.system.ball\_controller.BallController attribute), 55
- BallSave (class in mpf.plugins.ball\_save), 49
- BallSearch (class in mpf.plugins.ball\_search), 49
- bank() (mpf.platform.p3\_roc.PDBCoil method), 38
- bank() (mpf.platform.p\_roc.PDBCoil method), 43
- BCP (class in mpf.system.bcp), 56
- bcp\_mode\_start() (mpf.system.bcp.BCP method), 56
- bcp\_mode\_stop() (mpf.system.bcp.BCP method), 56
- bcp\_player\_added() (mpf.system.bcp.BCP method), 56
- bcp\_receive\_dmd\_frame() (mpf.system.bcp.BCP method), 56
- bcp\_receive\_error() (mpf.system.bcp.BCP method), 56
- bcp\_receive\_get() (mpf.system.bcp.BCP method), 56
- bcp\_receive\_set() (mpf.system.bcp.BCP method), 56
- bcp\_receive\_switch() (mpf.system.bcp.BCP method), 56
- bcp\_receive\_trigger() (mpf.system.bcp.BCP method), 56
- bcp\_reset() (mpf.system.bcp.BCP method), 56
- bcp\_trigger() (mpf.system.bcp.BCP method), 57
- BCPClientSocket (class in mpf.system.bcp), 59
- blit\_8bit\_alpha() (mpf.media\_controller.core.slide.Slide method), 25
- build\_slide() (mpf.media\_controller.core.slide\_builder.SlideBuilder method), 27
- C**
- call() (mpf.system.timing.Timer method), 84
- CaseInsensitiveDict (class in mpf.system.config), 60
- change\_speed() (mpf.media\_controller.core.show\_controller.Show method), 22
- change\_speed() (mpf.system.light\_controller.Show method), 72
- Channel (class in mpf.media\_controller.core.sound), 27
- check() (mpf.system.tasks.DelayManager method), 83
- check\_config\_file\_version() (mpf.system.config.Config static method), 61
- check\_for\_invalid\_sections() (mpf.system.config.Config method), 61
- check\_hw\_switches() (mpf.devices.score\_reel.ScoreReel method), 15
- chime() (mpf.devices.score\_reel.ScoreReelGroup method), 18
- class\_label (mpf.devices.autofire.AutofireCoil attribute), 1
- class\_label (mpf.devices.ball\_device.BallDevice attribute), 2
- class\_label (mpf.devices.diverter.Diverter attribute), 4
- class\_label (mpf.devices.driver.Driver attribute), 5
- class\_label (mpf.devices.drop\_target.DropTarget attribute), 6
- class\_label (mpf.devices.drop\_target.DropTargetBank attribute), 6
- class\_label (mpf.devices.flasher.Flasher attribute), 7
- class\_label (mpf.devices.flipper.Flipper attribute), 8
- class\_label (mpf.devices.gi.GI attribute), 9
- class\_label (mpf.devices.led.LED attribute), 10
- class\_label (mpf.devices.matrix\_light.MatrixLight attribute), 11
- class\_label (mpf.devices.new\_device\_template.YourNewDevice attribute), 12
- class\_label (mpf.devices.playfield.Playfield attribute), 13
- class\_label (mpf.devices.playfield\_transfer.PlayfieldTransfer attribute), 14
- class\_label (mpf.devices.score\_reel.ScoreReel attribute), 15
- class\_label (mpf.devices.score\_reel.ScoreReelGroup attribute), 18
- class\_label (mpf.devices.switch.Switch attribute), 19
- clear() (mpf.media\_controller.core.slide.Slide method), 26
- clear() (mpf.system.events.QueuedEvent method), 67
- clear() (mpf.system.tasks.DelayManager method), 83
- clear\_hw\_rule() (mpf.platform.fast.HardwarePlatform method), 33
- clear\_hw\_rule() (mpf.platform.p3\_roc.HardwarePlatform method), 36
- clear\_hw\_rule() (mpf.platform.p\_roc.HardwarePlatform method), 41
- clear\_hw\_rule() (mpf.platform.virtual.HardwarePlatform method), 46
- clear\_hw\_rule() (mpf.system.platform.Platform method), 77
- client\_send\_osc\_message() (mpf.plugins.osc.OSC method), 50
- client\_update\_all() (mpf.plugins.osc.OSC method), 50
- client\_update\_all\_switches() (mpf.plugins.osc.OSC method), 50
- client\_update\_light() (mpf.plugins.osc.OSC method), 50
- client\_update\_switch() (mpf.plugins.osc.OSC method), 50
- collection (mpf.devices.autofire.AutofireCoil attribute), 1
- collection (mpf.devices.ball\_device.BallDevice attribute), 2
- collection (mpf.devices.diverter.Diverter attribute), 4
- collection (mpf.devices.driver.Driver attribute), 5
- collection (mpf.devices.drop\_target.DropTarget attribute), 6
- collection (mpf.devices.drop\_target.DropTargetBank attribute), 7
- collection (mpf.devices.flasher.Flasher attribute), 7
- collection (mpf.devices.flipper.Flipper attribute), 8
- collection (mpf.devices.gi.GI attribute), 9
- collection (mpf.devices.led.LED attribute), 10
- collection (mpf.devices.matrix\_light.MatrixLight attribute), 11



- tribute), 11
- collection (mpf.devices.new\_device\_template.YourNewDevice attribute), 12
- collection (mpf.devices.playfield.Playfield attribute), 13
- collection (mpf.devices.playfield\_transfer.PlayfieldTransfer attribute), 14
- collection (mpf.devices.score\_reel.ScoreReel attribute), 15
- collection (mpf.devices.score\_reel.ScoreReelGroup attribute), 18
- collection (mpf.devices.switch.Switch attribute), 19
- color() (mpf.devices.led.LED method), 10
- color() (mpf.platform.fast.FASTDirectLED method), 33
- color() (mpf.platform.p3\_roc.PDBLED method), 39
- color() (mpf.platform.p\_roc.PDBLED method), 44
- color() (mpf.platform.virtual.VirtualLED method), 47
- compensate() (mpf.devices.led.LED method), 10
- complete() (mpf.system.logic\_blocks.LogicBlock method), 74
- Config (class in mpf.system.config), 60
- config (mpf.system.machine.MachineController attribute), 75
- config\_section (mpf.devices.autofire.AutofireCoil attribute), 1
- config\_section (mpf.devices.ball\_device.BallDevice attribute), 2
- config\_section (mpf.devices.diverter.Diverter attribute), 4
- config\_section (mpf.devices.driver.Driver attribute), 5
- config\_section (mpf.devices.drop\_target.DropTarget attribute), 6
- config\_section (mpf.devices.drop\_target.DropTargetBank attribute), 7
- config\_section (mpf.devices.flasher.Flasher attribute), 7
- config\_section (mpf.devices.flipper.Flipper attribute), 8
- config\_section (mpf.devices.gi.GI attribute), 9
- config\_section (mpf.devices.led.LED attribute), 10
- config\_section (mpf.devices.matrix\_light.MatrixLight attribute), 11
- config\_section (mpf.devices.new\_device\_template.YourNewDevice attribute), 12
- config\_section (mpf.devices.playfield.Playfield attribute), 13
- config\_section (mpf.devices.playfield\_transfer.PlayfieldTransfer attribute), 14
- config\_section (mpf.devices.score\_reel.ScoreReel attribute), 15
- config\_section (mpf.devices.score\_reel.ScoreReelGroup attribute), 18
- config\_section (mpf.devices.switch.Switch attribute), 19
- configure\_debugger() (mpf.system.machine.MachineController method), 76
- configure\_dmd() (mpf.platform.fast.HardwarePlatform method), 34
- configure\_dmd() (mpf.platform.p3\_roc.HardwarePlatform method), 37
- configure\_dmd() (mpf.platform.p\_roc.HardwarePlatform method), 42
- configure\_dmd() (mpf.platform.virtual.HardwarePlatform method), 46
- configure\_dmd() (mpf.system.platform.Platform method), 77
- configure\_driver() (mpf.platform.fast.HardwarePlatform method), 34
- configure\_driver() (mpf.platform.p3\_roc.HardwarePlatform method), 37
- configure\_driver() (mpf.platform.p\_roc.HardwarePlatform method), 42
- configure\_driver() (mpf.platform.virtual.HardwarePlatform method), 46
- configure\_driver() (mpf.system.platform.Platform method), 77
- configure\_eject\_targets() (mpf.devices.ball\_device.BallDevice method), 2
- configure\_gi() (mpf.platform.fast.HardwarePlatform method), 34
- configure\_gi() (mpf.platform.p3\_roc.HardwarePlatform method), 37
- configure\_gi() (mpf.platform.p\_roc.HardwarePlatform method), 42
- configure\_gi() (mpf.platform.virtual.HardwarePlatform method), 46
- configure\_gi() (mpf.system.platform.Platform method), 77
- configure\_globals() (mpf.platform.p3\_roc.PDBConfig method), 39
- configure\_globals() (mpf.platform.p\_roc.PDBConfig method), 44
- configure\_led() (mpf.platform.fast.HardwarePlatform method), 34
- configure\_led() (mpf.platform.p3\_roc.HardwarePlatform method), 37
- configure\_led() (mpf.platform.p\_roc.HardwarePlatform method), 42
- configure\_led() (mpf.platform.virtual.HardwarePlatform method), 46
- configure\_led() (mpf.system.platform.Platform method), 78
- configure\_matrixlight() (mpf.platform.fast.HardwarePlatform method), 34
- configure\_matrixlight() (mpf.platform.p3\_roc.HardwarePlatform method), 37
- configure\_matrixlight() (mpf.platform.p\_roc.HardwarePlatform method), 42
- configure\_matrixlight() (mpf.platform.virtual.HardwarePlatform method), 46
- configure\_matrixlight() (mpf.system.platform.Platform method), 78

`configure_switch()` (mpf.platform.fast.HardwarePlatform method), 34  
`configure_switch()` (mpf.platform.p3\_roc.HardwarePlatform method), 37  
`configure_switch()` (mpf.platform.p\_roc.HardwarePlatform method), 42  
`configure_switch()` (mpf.platform.virtual.HardwarePlatform method), 46  
`configure_switch()` (mpf.system.platform.Platform method), 78  
`count_balls()` (mpf.devices.ball\_device.BallDevice method), 2  
`count_balls()` (mpf.devices.playfield.Playfield method), 13  
`Counter` (class in mpf.system.logic\_blocks), 74  
`create()` (mpf.system.tasks.Task static method), 83  
`create_channel()` (mpf.media\_controller.core.sound.Track method), 30  
`create_control_events()` (mpf.system.logic\_blocks.LogicBlock method), 74  
`create_loader_thread()` (mpf.system.assets.AssetManager method), 53  
`create_show_from_script()` (mpf.system.light\_controller.LightController method), 67  
`create_socket_threads()` (mpf.system.bcp.BCPClientSocket method), 59  
`create_track()` (mpf.media\_controller.core.sound.SoundController method), 29  
`create_trigger_event()` (mpf.system.bcp.BCP method), 57

## D

`deactivate()` (mpf.devices.diverter.Diverter method), 4  
`dead_delay_managers` (mpf.system.tasks.DelayManager attribute), 83  
`decode()` (mpf.platform.p3\_roc.DriverAlias method), 36  
`decode()` (mpf.platform.p\_roc.DriverAlias method), 41  
`decode_command_string()` (in module mpf.system.bcp), 60  
`decode_pdb_address()` (in module mpf.platform.p3\_roc), 41  
`decode_pdb_address()` (in module mpf.platform.p\_roc), 46  
`decrease_volume()` (mpf.system.bcp.BCP method), 57  
`dedicated_bank()` (mpf.platform.p3\_roc.PDBLight method), 39  
`dedicated_bank()` (mpf.platform.p\_roc.PDBLight method), 44  
`dedicated_output()` (mpf.platform.p3\_roc.PDBLight method), 39  
`dedicated_output()` (mpf.platform.p\_roc.PDBLight method), 44  
`delay_managers` (mpf.system.tasks.DelayManager attribute), 83  
`DelayManager` (class in mpf.system.tasks), 82  
`depth` (mpf.media\_controller.core.slide.Slide attribute), 25  
`device_class_init()` (mpf.devices.led.LED class method), 10  
`device_class_init()` (mpf.devices.new\_device\_template.YourNewDevice class method), 12  
`device_class_init()` (mpf.devices.score\_reel.ScoreReelGroup class method), 18  
`dict_merge()` (mpf.system.config.Config static method), 61  
`disable()` (mpf.devices.autofire.AutofireCoil method), 1  
`disable()` (mpf.devices.diverter.Diverter method), 4  
`disable()` (mpf.devices.driver.Driver method), 5  
`disable()` (mpf.devices.flipper.Flipper method), 8  
`disable()` (mpf.devices.gi.GI method), 9  
`disable()` (mpf.devices.led.LED method), 10  
`disable()` (mpf.platform.fast.FASTDirectLED method), 33  
`disable()` (mpf.platform.fast.FASTDriver method), 33  
`disable()` (mpf.platform.p3\_roc.PDBLED method), 39  
`disable()` (mpf.platform.p3\_roc.PROCDriver method), 40  
`disable()` (mpf.platform.p\_roc.PDBLED method), 44  
`disable()` (mpf.platform.p\_roc.PROCDriver method), 45  
`disable()` (mpf.platform.virtual.VirtualDriver method), 47  
`disable()` (mpf.platform.virtual.VirtualLED method), 47  
`disable()` (mpf.plugins.auditor.Auditor method), 48  
`disable()` (mpf.system.logic\_blocks.LogicBlock method), 74  
`disable_bcp_switch()` (mpf.system.bcp.BCP method), 57  
`disable_bcp_switches()` (mpf.system.bcp.BCP method), 57  
`disable_held_coil()` (mpf.devices.diverter.Diverter method), 4  
`disable_hw_switch()` (mpf.devices.diverter.Diverter method), 4  
`disable_volume_keys()` (mpf.system.bcp.BCP method), 57  
`display` (mpf.system.machine.MachineController attribute), 75  
`Diverter` (class in mpf.devices.diverter), 3  
`dmd` (mpf.platform.p\_roc.PROCDMD attribute), 45  
`do_load()` (mpf.media\_controller.core.show\_controller.Show method), 23  
`do_load()` (mpf.media\_controller.core.sound.Sound method), 28  
`do_load()` (mpf.system.assets.Asset method), 52  
`do_load()` (mpf.system.light\_controller.Show method), 72  
`does_event_exist()` (mpf.system.events.EventManager method), 64  
`done` (mpf.system.machine.MachineController attribute), 75  
`Driver` (class in mpf.devices.driver), 5  
`DriverAlias` (class in mpf.platform.p3\_roc), 36

DriverAlias (class in mpf.platform.p\_roc), 41  
 DropTarget (class in mpf.devices.drop\_target), 6  
 DropTargetBank (class in mpf.devices.drop\_target), 6

## E

eject() (mpf.devices.ball\_device.BallDevice method), 2  
 eject() (mpf.devices.playfield.Playfield method), 13  
 eject\_all() (mpf.devices.ball\_device.BallDevice method), 2  
 eject\_all() (mpf.devices.playfield.Playfield method), 13  
 eject\_failed() (mpf.devices.ball\_device.BallDevice method), 2  
 elements (mpf.media\_controller.core.slide.Slide attribute), 24  
 enable() (mpf.devices.autofire.AutofireCoil method), 1  
 enable() (mpf.devices.diverter.Diverter method), 4  
 enable() (mpf.devices.driver.Driver method), 5  
 enable() (mpf.devices.flipper.Flipper method), 8  
 enable() (mpf.devices.gi.GI method), 9  
 enable() (mpf.platform.fast.FASTDirectLED method), 33  
 enable() (mpf.platform.fast.FASTDriver method), 33  
 enable() (mpf.platform.p3\_roc.PDBLED method), 39  
 enable() (mpf.platform.p3\_roc.PROCDriver method), 40  
 enable() (mpf.platform.p\_roc.PDBLED method), 44  
 enable() (mpf.platform.p\_roc.PROCDriver method), 45  
 enable() (mpf.platform.virtual.VirtualDriver method), 47  
 enable() (mpf.platform.virtual.VirtualLED method), 47  
 enable() (mpf.plugins.auditor.Auditor method), 48  
 enable() (mpf.plugins.ball\_save.BallSave method), 49  
 enable() (mpf.system.logic\_blocks.Accrual method), 73  
 enable() (mpf.system.logic\_blocks.Counter method), 74  
 enable() (mpf.system.logic\_blocks.LogicBlock method), 74  
 enable() (mpf.system.logic\_blocks.Sequence method), 75  
 enable\_bcp\_switch() (mpf.system.bcp.BCP method), 57  
 enable\_bcp\_switches() (mpf.system.bcp.BCP method), 57  
 enable\_hw\_switches() (mpf.devices.diverter.Diverter method), 4  
 enable\_no\_hold() (mpf.devices.flipper.Flipper method), 8  
 enable\_partial\_power() (mpf.devices.flipper.Flipper method), 8  
 enable\_volume\_keys() (mpf.system.bcp.BCP method), 57  
 encode\_command\_string() (in module mpf.system.bcp), 60  
 end() (mpf.plugins.ball\_search.BallSearch method), 49  
 event\_config\_to\_dict() (mpf.system.config.Config static method), 61  
 EventManager (class in mpf.system.events), 63  
 events (mpf.system.machine.MachineController attribute), 76  
 expire\_ms (mpf.media\_controller.core.slide.Slide attribute), 25

## F

fade() (mpf.platform.fast.FASTDirectLED method), 33  
 fade() (mpf.platform.p3\_roc.PDBLED method), 39  
 fade() (mpf.platform.p\_roc.PDBLED method), 44  
 fadeout() (mpf.media\_controller.core.sound.StreamTrack method), 29  
 FASTDirectLED (class in mpf.platform.fast), 32  
 FASTDMD (class in mpf.platform.fast), 32  
 FASTDriver (class in mpf.platform.fast), 33  
 FASTGString (class in mpf.platform.fast), 33  
 FASTMatrixLight (class in mpf.platform.fast), 33  
 FASTSwitch (class in mpf.platform.fast), 33  
 flash() (mpf.devices.flasher.Flasher method), 7  
 Flasher (class in mpf.devices.flasher), 7  
 Flipper (class in mpf.devices.flipper), 7  
 found\_new\_osc\_client() (mpf.plugins.osc.OSC method), 50  
 future\_pulse() (mpf.platform.p3\_roc.PROCDriver method), 40  
 future\_pulse() (mpf.platform.p\_roc.PROCDriver method), 45  
 future\_pulse() (mpf.platform.virtual.VirtualDriver method), 47

## G

game\_ended() (mpf.plugins.info\_lights.InfoLights method), 50  
 game\_starting() (mpf.devices.score\_reel.ScoreReelController method), 16  
 game\_starting() (mpf.plugins.info\_lights.InfoLights method), 50  
 gather\_balls() (mpf.system.ball\_controller.BallController method), 55  
 get\_additional\_ball\_capacity() (mpf.devices.ball\_device.BallDevice method), 3  
 get\_additional\_ball\_capacity() (mpf.devices.playfield.Playfield method), 13  
 get\_bcp\_messages() (mpf.system.bcp.BCP method), 58  
 get\_debug\_status() (mpf.system.machine.MachineController method), 76  
 get\_from\_socket() (mpf.system.bcp.BCPCClientSocket method), 59  
 get\_globals() (mpf.platform.p3\_roc.PDBConfig method), 39  
 get\_hw\_switch\_states() (mpf.platform.fast.HardwarePlatform method), 34  
 get\_hw\_switch\_states() (mpf.platform.p3\_roc.HardwarePlatform method), 38  
 get\_hw\_switch\_states() (mpf.platform.p\_roc.HardwarePlatform method), 42  
 get\_hw\_switch\_states() (mpf.platform.virtual.HardwarePlatform method), 46

- [get\\_hw\\_switch\\_states\(\)](#) (mpf.system.platform.Platform method), 78  
[get\\_key\\_press\\_string\(\)](#) (mpf.media\_controller.core.keyboard.Keyboard method), 21  
[get\\_language\(\)](#) (mpf.media\_controller.core.language.Language method), 21  
[get\\_physical\\_value\\_list\(\)](#) (mpf.devices.score\_reel.ScoreReelGroup method), 18  
[get\\_proc\\_number\(\)](#) (mpf.platform.p3\_roc.PDBConfig method), 39  
[get\\_proc\\_number\(\)](#) (mpf.platform.p\_roc.PDBConfig method), 44  
[get\\_sound\(\)](#) (mpf.media\_controller.core.sound.Track method), 30  
[get\\_state\(\)](#) (mpf.devices.led.LED method), 10  
[get\\_status\(\)](#) (mpf.devices.ball\_device.BallDevice method), 3  
[get\\_subsurface\(\)](#) (mpf.media\_controller.core.slide.Slide method), 26  
[get\\_text\(\)](#) (mpf.media\_controller.core.language.Language method), 21  
[get\\_volume\(\)](#) (mpf.media\_controller.core.sound.SoundController method), 29  
[GI](#) (class in mpf.devices.gi), 9
- ## H
- [HardwarePlatform](#) (class in mpf.platform.fast), 33  
[HardwarePlatform](#) (class in mpf.platform.p3\_roc), 36  
[HardwarePlatform](#) (class in mpf.platform.p\_roc), 41  
[HardwarePlatform](#) (class in mpf.platform.virtual), 46  
[height](#) (mpf.media\_controller.core.slide.Slide attribute), 25  
[hex\\_to\\_rgb\(\)](#) (mpf.platform.fast.FASTDirectLED method), 33  
[hexstring\\_to\\_int\(\)](#) (mpf.system.config.Config static method), 61  
[hexstring\\_to\\_list\(\)](#) (mpf.system.config.Config static method), 62  
[hit\(\)](#) (mpf.system.logic\_blocks.Accrual method), 74  
[hit\(\)](#) (mpf.system.logic\_blocks.Counter method), 74  
[hit\(\)](#) (mpf.system.logic\_blocks.Sequence method), 75  
[hold\(\)](#) (mpf.devices.ball\_device.BallDevice method), 3  
[HZ](#) (mpf.system.timing.Timing attribute), 84
- ## I
- [identify\\_connection\(\)](#) (mpf.platform.fast.SerialCommunicator method), 36  
[increase\\_volume\(\)](#) (mpf.system.bcp.BCP method), 58  
[indexes](#) (mpf.platform.p3\_roc.PDBConfig attribute), 39  
[indexes](#) (mpf.platform.p\_roc.PDBConfig attribute), 44  
[InfoLights](#) (class in mpf.plugins.info\_lights), 50  
[initialize\(\)](#) (mpf.devices.score\_reel.ScoreReelGroup method), 18  
[initialize\\_drivers\(\)](#) (mpf.platform.p3\_roc.PDBConfig method), 39  
[initialize\\_drivers\(\)](#) (mpf.platform.p\_roc.PDBConfig method), 44  
[int\\_to\\_hex\\_string\(\)](#) (mpf.system.config.Config static method), 62  
[int\\_to\\_pwm\(\)](#) (mpf.system.timing.Timing static method), 84  
[int\\_to\\_reel\\_list\(\)](#) (mpf.devices.score\_reel.ScoreReelGroup method), 18  
[invert\(\)](#) (mpf.devices.flipper.Flipper class method), 8  
[is\\_active\(\)](#) (mpf.system.switch\_controller.SwitchController method), 81  
[is\\_desired\\_valid\(\)](#) (mpf.devices.score\_reel.ScoreReelGroup method), 18  
[is\\_direct\\_coil\(\)](#) (mpf.platform.p3\_roc.PDBCoil method), 38  
[is\\_direct\\_coil\(\)](#) (mpf.platform.p\_roc.PDBCoil method), 43  
[is\\_direct\\_lamp\(\)](#) (mpf.platform.p3\_roc.PDBLight method), 39  
[is\\_direct\\_lamp\(\)](#) (mpf.platform.p\_roc.PDBLight method), 44  
[is\\_empty\(\)](#) (mpf.system.events.QueuedEvent method), 67  
[is\\_full\(\)](#) (mpf.devices.ball\_device.BallDevice method), 3  
[is\\_inactive\(\)](#) (mpf.system.switch\_controller.SwitchController method), 81  
[is\\_pdb\\_address\(\)](#) (in module mpf.platform.p3\_roc), 41  
[is\\_pdb\\_address\(\)](#) (in module mpf.platform.p\_roc), 46  
[is\\_pdb\\_coil\(\)](#) (mpf.platform.p3\_roc.PDBCoil method), 39  
[is\\_pdb\\_coil\(\)](#) (mpf.platform.p\_roc.PDBCoil method), 43  
[is\\_pdb\\_lamp\(\)](#) (mpf.platform.p3\_roc.PDBLight method), 39  
[is\\_pdb\\_lamp\(\)](#) (mpf.platform.p\_roc.PDBLight method), 44  
[is\\_playfield\(\)](#) (mpf.devices.ball\_device.BallDevice method), 3  
[is\\_state\(\)](#) (mpf.system.switch\_controller.SwitchController method), 81
- ## K
- [Keyboard](#) (class in mpf.media\_controller.core.keyboard), 20  
[keys\\_to\\_lower\(\)](#) (mpf.system.config.Config static method), 62  
[kill\(\)](#) (mpf.system.events.QueuedEvent method), 67  
[knockdown\(\)](#) (mpf.devices.drop\_target.DropTarget method), 6
- ## L
- [Language](#) (class in mpf.media\_controller.core.language), 21  
[LED](#) (class in mpf.devices.led), 10

- light() (mpf.devices.score\_reel.ScoreReelGroup method), 18
- LightController (class in mpf.system.light\_controller), 67
- list\_of\_lists() (mpf.system.config.Config static method), 62
- load() (mpf.system.assets.Asset method), 52
- load\_asset() (mpf.system.assets.AssetManager method), 53
- load\_assets() (mpf.system.assets.AssetManager method), 53
- load\_config\_yaml() (mpf.system.config.Config static method), 62
- load\_from\_disk() (mpf.plugins.auditor.Auditor method), 48
- load\_show\_from\_disk() (mpf.media\_controller.core.show\_controller.Show method), 23
- load\_show\_from\_disk() (mpf.system.light\_controller.Show method), 72
- locate\_asset\_file() (mpf.system.assets.AssetManager method), 53
- log (mpf.system.switch\_controller.SwitchController attribute), 81
- log\_active\_switches() (mpf.system.switch\_controller.SwitchController method), 81
- log\_loop\_rate() (mpf.system.machine.MachineController method), 76
- log\_system\_info() (mpf.system.machine.MachineController method), 76
- logical\_to\_physical() (mpf.devices.score\_reel.ScoreReel method), 15
- LogicBlock (class in mpf.system.logic\_blocks), 74
- LogicBlocks (class in mpf.system.logic\_blocks), 75
- lookup\_invalid\_config\_setting() (mpf.system.config.Config method), 62
- M**
- machine (mpf.platform.p3\_roc.HardwarePlatform attribute), 36
- machine (mpf.platform.p\_roc.HardwarePlatform attribute), 41
- machine\_path (mpf.system.machine.MachineController attribute), 75
- machine\_type (mpf.platform.p3\_roc.HardwarePlatform attribute), 36
- machine\_type (mpf.platform.p\_roc.HardwarePlatform attribute), 41
- MachineController (class in mpf.system.machine), 75
- machineflow\_index (mpf.system.machine.MachineController attribute), 75
- make\_sure\_path\_exists() (mpf.plugins.auditor.Auditor method), 48
- map\_new\_score\_reel\_group() (mpf.devices.score\_reel.ScoreReelController method), 17
- match() (mpf.plugins.info\_lights.InfoLights method), 50
- matches() (mpf.platform.p3\_roc.DriverAlias method), 36
- matches() (mpf.platform.p\_roc.DriverAlias method), 41
- MatrixLight (class in mpf.devices.matrix\_light), 11
- member\_target\_change() (mpf.devices.drop\_target.DropTargetBank method), 7
- mpf (module), 87
- mpf.devices (module), 20
- mpf.devices.autofire (module), 1
- mpf.devices.ball\_device (module), 1
- mpf.devices.diverter (module), 3
- mpf.devices.driver (module), 5
- mpf.devices.drop\_target (module), 6
- mpf.devices.flasher (module), 7
- mpf.devices.flipper (module), 7
- mpf.devices.gi (module), 9
- mpf.devices.led (module), 10
- mpf.devices.matrix\_light (module), 11
- mpf.devices.new\_device\_template (module), 12
- mpf.devices.playfield (module), 12
- mpf.devices.playfield\_transfer (module), 14
- mpf.devices.score\_reel (module), 14
- mpf.devices.switch (module), 19
- mpf.media\_controller (module), 32
- mpf.media\_controller.core (module), 31
- mpf.media\_controller.core.keyboard (module), 20
- mpf.media\_controller.core.language (module), 21
- mpf.media\_controller.core.show\_controller (module), 22
- mpf.media\_controller.core.slide (module), 24
- mpf.media\_controller.core.slide\_builder (module), 27
- mpf.media\_controller.core.sound (module), 27
- mpf.media\_controller.decorators (module), 31
- mpf.media\_controller.display\_modules (module), 31
- mpf.media\_controller.elements (module), 32
- mpf.media\_controller.transitions (module), 32
- mpf.platform (module), 48
- mpf.platform.fast (module), 32
- mpf.platform.p3\_roc (module), 36
- mpf.platform.p\_roc (module), 41
- mpf.platform.virtual (module), 46
- mpf.plugins (module), 52
- mpf.plugins.auditor (module), 48
- mpf.plugins.ball\_save (module), 49
- mpf.plugins.ball\_search (module), 49
- mpf.plugins.info\_lights (module), 50
- mpf.plugins.osc (module), 50
- mpf.plugins.socket\_events (module), 51
- mpf.plugins.switch\_player (module), 52
- mpf.system (module), 85
- mpf.system.assets (module), 52
- mpf.system.ball\_controller (module), 55
- mpf.system.bcp (module), 56
- mpf.system.config (module), 60



mpf.system.events (module), 63  
mpf.system.light\_controller (module), 67  
mpf.system.logic\_blocks (module), 73  
mpf.system.machine (module), 75  
mpf.system.platform (module), 77  
mpf.system.scoring (module), 79  
mpf.system.scriptlet (module), 80  
mpf.system.switch\_controller (module), 80  
mpf.system.tasks (module), 82  
mpf.system.timing (module), 84  
ms\_per\_tick (mpf.system.timing.Timing attribute), 84  
ms\_since\_change() (mpf.system.switch\_controller.SwitchController method), 81

## N

name (mpf.media\_controller.core.slide.Slide attribute), 24  
NewTasks (mpf.system.tasks.Task attribute), 83  
normalize\_color() (mpf.platform.p3\_roc.PDBLED method), 39  
normalize\_color() (mpf.platform.p\_roc.PDBLED method), 44  
normalize\_hex\_string() (mpf.system.config.Config static method), 62  
null\_dmd\_sender() (mpf.platform.fast.HardwarePlatform method), 34  
num\_balls\_ejectable (mpf.devices.ball\_device.BallDevice attribute), 3  
num\_balls\_known (mpf.system.ball\_controller.BallController attribute), 55

## O

off() (mpf.devices.led.LED method), 10  
off() (mpf.devices.matrix\_light.MatrixLight method), 11  
off() (mpf.platform.fast.FASTGString method), 33  
off() (mpf.platform.fast.FASTMatrixLight method), 33  
off() (mpf.platform.p3\_roc.PROCMatrixLight method), 41  
off() (mpf.platform.p\_roc.PROCMatrixLight method), 46  
off() (mpf.platform.virtual.VirtualGI method), 47  
off() (mpf.platform.virtual.VirtualMatrixLight method), 47  
ok\_to\_confirm\_ball\_via\_playfield\_switch() (mpf.devices.playfield.Playfield method), 13  
on() (mpf.devices.led.LED method), 10  
on() (mpf.devices.matrix\_light.MatrixLight method), 11  
on() (mpf.platform.fast.FASTGString method), 33  
on() (mpf.platform.fast.FASTMatrixLight method), 33  
on() (mpf.platform.p3\_roc.PROCMatrixLight method), 41  
on() (mpf.platform.p\_roc.PROCMatrixLight method), 46  
on() (mpf.platform.virtual.VirtualGI method), 47

on() (mpf.platform.virtual.VirtualMatrixLight method), 47  
on\_load() (mpf.system.scriptlet.Scriptlet method), 80  
options (mpf.system.machine.MachineController attribute), 75  
OSC (class in mpf.plugins.osc), 50  
output() (mpf.platform.p3\_roc.PDBCoil method), 39  
output() (mpf.platform.p\_roc.PDBCoil method), 43

## P

palette (mpf.media\_controller.core.slide.Slide attribute), 25  
parse\_matrix\_num() (mpf.platform.p3\_roc.PDBSwitch method), 40  
parse\_matrix\_num() (mpf.platform.p\_roc.PDBSwitch method), 45  
pause() (mpf.media\_controller.core.sound.StreamTrack method), 30  
PDBCoil (class in mpf.platform.p3\_roc), 38  
PDBCoil (class in mpf.platform.p\_roc), 43  
PDBConfig (class in mpf.platform.p3\_roc), 39  
PDBConfig (class in mpf.platform.p\_roc), 43  
PDBLED (class in mpf.platform.p3\_roc), 39  
PDBLED (class in mpf.platform.p\_roc), 44  
PDBLight (class in mpf.platform.p3\_roc), 39  
PDBLight (class in mpf.platform.p\_roc), 44  
PDBSwitch (class in mpf.platform.p3\_roc), 40  
PDBSwitch (class in mpf.platform.p\_roc), 45  
persist (mpf.media\_controller.core.slide.Slide attribute), 25  
physical\_hw (mpf.system.machine.MachineController attribute), 75  
Platform (class in mpf.system.platform), 77  
platform (mpf.system.machine.MachineController attribute), 76  
play() (mpf.media\_controller.core.show\_controller.Show method), 23  
play() (mpf.media\_controller.core.sound.Channel method), 28  
play() (mpf.media\_controller.core.sound.Sound method), 28  
play() (mpf.media\_controller.core.sound.StreamTrack method), 30  
play() (mpf.media\_controller.core.sound.Track method), 30  
play() (mpf.system.light\_controller.Show method), 72  
play\_show() (mpf.media\_controller.core.show\_controller.ShowController method), 24  
play\_show() (mpf.system.light\_controller.LightController method), 67  
player\_added() (mpf.plugins.info\_lights.InfoLights method), 50  
player\_eject\_request() (mpf.devices.playfield.Playfield method), 14

- Playfield (class in mpf.devices.playfield), 12
- playfield\_switch\_hit() (mpf.devices.playfield.Playfield method), 14
- PlayfieldTransfer (class in mpf.devices.playfield\_transfer), 14
- Playlist (class in mpf.system.light\_controller), 70
- plugin\_class (in module mpf.plugins.auditor), 49
- plugin\_class (in module mpf.plugins.ball\_save), 49
- plugin\_class (in module mpf.plugins.ball\_search), 49
- plugin\_class (in module mpf.plugins.info\_lights), 50
- plugin\_class (in module mpf.plugins.osc), 51
- plugin\_class (in module mpf.plugins.socket\_events), 52
- plugin\_class (in module mpf.plugins.switch\_player), 52
- plugins (mpf.system.machine.MachineController attribute), 76
- pop\_coil() (mpf.plugins.ball\_search.BallSearch method), 49
- post() (mpf.system.events.EventManager method), 64
- post\_boolean() (mpf.system.events.EventManager method), 64
- post\_queue() (mpf.system.events.EventManager method), 65
- post\_relay() (mpf.system.events.EventManager method), 65
- power\_off() (mpf.system.machine.MachineController method), 76
- preload\_check() (in module mpf.media\_controller.core.keyboard), 21
- preload\_check() (in module mpf.media\_controller.core.sound), 31
- preprocess\_settings() (mpf.media\_controller.core.slide\_builder.SlideBuilder method), 27
- priority (mpf.media\_controller.core.slide.Slide attribute), 24
- proc (mpf.platform.p3\_roc.HardwarePlatform attribute), 36
- proc (mpf.platform.p3\_roc.PDBConfig attribute), 39
- proc (mpf.platform.p\_roc.HardwarePlatform attribute), 41
- proc (mpf.platform.p\_roc.PDBConfig attribute), 44
- proc\_num() (mpf.platform.p3\_roc.PDBSwitch method), 40
- proc\_num() (mpf.platform.p\_roc.PDBSwitch method), 45
- PROCMDM (class in mpf.platform.p\_roc), 45
- PROCDriver (class in mpf.platform.p3\_roc), 40
- PROCDriver (class in mpf.platform.p\_roc), 45
- process\_assets\_from\_disk() (mpf.system.assets.AssetManager method), 54
- process\_bcp\_events() (mpf.system.bcp.BCP method), 58
- process\_coil() (mpf.plugins.osc.OSC method), 50
- process\_config() (mpf.media\_controller.core.slide\_builder.SlideBuilder method), 27
- process\_config() (mpf.plugins.osc.OSC method), 50
- process\_config() (mpf.plugins.socket\_events.SocketClient method), 51
- process\_config() (mpf.system.config.Config static method), 63
- process\_config() (mpf.system.scoring.ScoreController method), 79
- process\_config2() (mpf.system.config.Config method), 63
- process\_event() (mpf.plugins.osc.OSC method), 50
- process\_event\_player() (mpf.system.events.EventManager method), 66
- process\_flipper() (mpf.plugins.osc.OSC method), 50
- process\_key\_press() (mpf.media\_controller.core.keyboard.Keyboard method), 21
- process\_key\_release() (mpf.media\_controller.core.keyboard.Keyboard method), 21
- process\_light() (mpf.plugins.osc.OSC method), 50
- process\_light\_player() (mpf.system.light\_controller.LightController method), 68
- process\_light\_scripts() (mpf.system.light\_controller.LightController method), 68
- process\_message() (mpf.plugins.osc.OSC method), 50
- process\_random\_event\_player() (mpf.system.events.EventManager method), 66
- process\_received\_message() (mpf.platform.fast.HardwarePlatform method), 34
- process\_shows\_from\_config() (mpf.media\_controller.core.show\_controller.ShowController method), 24
- process\_switch() (mpf.plugins.osc.OSC method), 51
- process\_switch() (mpf.system.switch\_controller.SwitchController method), 81
- PROCMatrixLight (class in mpf.platform.p3\_roc), 40
- PROCMatrixLight (class in mpf.platform.p\_roc), 46
- PROCSwitch (class in mpf.platform.p3\_roc), 41
- PROCSwitch (class in mpf.platform.p\_roc), 46
- pulse() (mpf.devices.driver.Driver method), 5
- pulse() (mpf.platform.fast.FASTDriver method), 33
- pulse() (mpf.platform.p3\_roc.PROCDriver method), 40
- pulse() (mpf.platform.p\_roc.PROCDriver method), 45
- pulse() (mpf.platform.virtual.VirtualDriver method), 47
- pulse\_ms (mpf.devices.score\_reel.ScoreReel attribute), 16
- pulsed\_patter() (mpf.platform.virtual.VirtualDriver method), 47
- pwm() (mpf.devices.driver.Driver method), 5
- pwm() (mpf.platform.fast.FASTDriver method), 33
- pwm() (mpf.platform.p3\_roc.PROCDriver method), 40
- pwm() (mpf.platform.p\_roc.PROCDriver method), 45
- pwm() (mpf.platform.virtual.VirtualDriver method), 47
- pin\_to\_byte\_int() (mpf.system.timing.Timing static method), 84

## Q

`query_fast_io_boards()` (mpf.platform.fast.SerialCommunicator method), 36

`queue_sound()` (mpf.media\_controller.core.sound.Track method), 31

`QueuedEvent` (class in mpf.system.events), 67

`quit()` (mpf.system.machine.MachineController method), 76

## R

`ready()` (mpf.media\_controller.core.slide.Slide method), 26

`receive_dx()` (mpf.platform.fast.HardwarePlatform method), 34

`receive_goodbye()` (mpf.system.bcp.BCPCClientSocket method), 59

`receive_hello()` (mpf.system.bcp.BCPCClientSocket method), 59

`receive_id()` (mpf.platform.fast.HardwarePlatform method), 34

`receive_local_closed()` (mpf.platform.fast.HardwarePlatform method), 34

`receive_local_open()` (mpf.platform.fast.HardwarePlatform method), 34

`receive_loop()` (mpf.system.bcp.BCPCClientSocket method), 59

`receive_lx()` (mpf.platform.fast.HardwarePlatform method), 34

`receive_ni()` (mpf.platform.fast.HardwarePlatform method), 34

`receive_nw_closed()` (mpf.platform.fast.HardwarePlatform method), 34

`receive_nw_open()` (mpf.platform.fast.HardwarePlatform method), 34

`receive_px()` (mpf.platform.fast.HardwarePlatform method), 34

`receive_rx()` (mpf.platform.fast.HardwarePlatform method), 34

`receive_sa()` (mpf.platform.fast.HardwarePlatform method), 34

`receive_sx()` (mpf.platform.fast.HardwarePlatform method), 34

`receive_wd()` (mpf.platform.fast.HardwarePlatform method), 34

`receive_wx()` (mpf.platform.fast.HardwarePlatform method), 35

`reconfigure()` (mpf.platform.virtual.VirtualDriver method), 47

`reel_list_to_int()` (mpf.devices.score\_reel.ScoreReelGroup method), 18

`refresh()` (mpf.media\_controller.core.slide.Slide method), 26

`register_and_load_machine_assets()` (mpf.system.assets.AssetManager method), 54

`register_asset()` (mpf.system.assets.AssetManager method), 54

`register_assets()` (mpf.system.assets.AssetManager method), 54

`register_data()` (mpf.plugins.osc.OSC method), 51

`register_lights()` (mpf.plugins.osc.OSC method), 51

`register_monitor()` (mpf.system.machine.MachineController method), 76

`register_mpfmc_trigger_events()` (mpf.system.bcp.BCP method), 58

`register_processor_connection()` (mpf.platform.fast.HardwarePlatform method), 35

`register_score_event()` (mpf.system.scoring.ScoreController method), 79

`register_sound_event()` (mpf.media\_controller.core.sound.SoundController method), 29

`register_sound_events()` (mpf.media\_controller.core.sound.SoundController method), 29

`register_switches()` (mpf.plugins.osc.OSC method), 51

`register_triggers()` (mpf.system.bcp.BCP method), 58

`removal_key` (mpf.media\_controller.core.slide.Slide attribute), 25

`remove()` (mpf.media\_controller.core.slide.Slide method), 26

`remove()` (mpf.system.tasks.DelayManager method), 83

`remove()` (mpf.system.timing.Timing method), 85

`remove_bcp_connection()` (mpf.system.bcp.BCP method), 58

`remove_element()` (mpf.media\_controller.core.slide.Slide method), 26

`remove_handler()` (mpf.devices.gi.GI method), 9

`remove_handler()` (mpf.devices.matrix\_light.MatrixLight method), 11

`remove_handler()` (mpf.system.events.EventManager method), 66

`remove_handler_by_event()` (mpf.system.events.EventManager method), 66

`remove_handler_by_key()` (mpf.system.events.EventManager method), 66

`remove_handlers_by_keys()` (mpf.system.events.EventManager method), 66

`remove_monitor()` (mpf.system.events.EventManager method), 66

`remove_player_controlled_eject()` (mpf.devices.playfield.Playfield method), 14

`remove_switch_handler()` (mpf.system.switch\_controller.SwitchController method), 82

`replace_handler()` (mpf.system.events.EventManager method), 66

`request_ball()` (mpf.devices.ball\_device.BallDevice method), 3



- request\_to\_start\_game() (mpf.system.ball\_controller.BallController method), 55
- reset() (mpf.devices.drop\_target.DropTarget method), 6
- reset() (mpf.devices.drop\_target.DropTargetBank method), 7
- reset() (mpf.system.logic\_blocks.Accrual method), 74
- reset() (mpf.system.logic\_blocks.Counter method), 74
- reset() (mpf.system.logic\_blocks.LogicBlock method), 74
- reset() (mpf.system.logic\_blocks.Sequence method), 75
- reset() (mpf.system.machine.MachineController method), 76
- reset() (mpf.system.tasks.DelayManager method), 83
- reset\_game\_lights() (mpf.plugins.info\_lights.InfoLights method), 50
- restart() (mpf.system.logic\_blocks.LogicBlock method), 74
- restart() (mpf.system.tasks.Task method), 83
- restore() (mpf.devices.led.LED method), 10
- restore() (mpf.devices.matrix\_light.MatrixLight method), 11
- restore\_lower\_lights() (mpf.system.light\_controller.LightController method), 68
- resync() (mpf.system.light\_controller.Show method), 73
- rgb\_to\_hex() (mpf.platform.fast.FASTDirectLED method), 33
- rotate\_player() (mpf.devices.score\_reel.ScoreReelController method), 17
- run() (mpf.system.assets.AssetLoader method), 52
- run() (mpf.system.machine.MachineController method), 76
- run\_loop() (mpf.system.platform.Platform method), 78
- run\_registered\_script() (mpf.system.light\_controller.LightController method), 68
- run\_script() (mpf.system.light\_controller.LightController method), 68
- ## S
- save\_to\_disk() (mpf.plugins.auditor.Auditor method), 48
- schedule() (mpf.platform.p3\_roc.PROCDriver method), 40
- schedule() (mpf.platform.p\_roc.PROCDriver method), 45
- schedule() (mpf.platform.virtual.VirtualDriver method), 47
- schedule\_deactivation() (mpf.devices.diverter.Diverter method), 4
- schedule\_removal() (mpf.media\_controller.core.slide.Slide method), 26
- score\_change() (mpf.devices.score\_reel.ScoreReelController method), 17
- ScoreController (class in mpf.system.scoring), 79
- ScoreReel (class in mpf.devices.score\_reel), 14
- ScoreReelController (class in mpf.devices.score\_reel), 16
- ScoreReelGroup (class in mpf.devices.score\_reel), 17
- Scriptlet (class in mpf.system.scriptlet), 80
- scriptlets (mpf.system.machine.MachineController attribute), 76
- secs() (mpf.system.timing.Timing static method), 85
- secs\_per\_tick (mpf.system.timing.Timing attribute), 85
- secs\_since\_change() (mpf.system.switch\_controller.SwitchController method), 82
- send() (mpf.platform.fast.SerialCommunicator method), 36
- send() (mpf.system.bcp.BCP method), 58
- send() (mpf.system.bcp.BCPClientSocket method), 59
- send\_goodbye() (mpf.system.bcp.BCPClientSocket method), 59
- send\_hello() (mpf.system.bcp.BCPClientSocket method), 60
- send\_message() (mpf.plugins.socket\_events.SocketClient method), 51
- send\_switch() (mpf.media\_controller.core.keyboard.Keyboard method), 21
- sending\_loop() (mpf.system.bcp.BCPClientSocket method), 60
- SerialCommunicator (class in mpf.system.logic\_blocks), 75
- SerialCommunicator (class in mpf.platform.fast), 35
- set\_brightness\_compensation() (mpf.devices.led.LED method), 10
- set\_default\_platform() (mpf.system.machine.MachineController method), 76
- set\_destination\_value() (mpf.devices.score\_reel.ScoreReel method), 16
- set\_hw\_rule() (mpf.system.platform.Platform method), 78
- set\_language() (mpf.media\_controller.core.language.Language method), 22
- set\_rollover\_reels() (mpf.devices.score\_reel.ScoreReelGroup method), 19
- set\_state() (mpf.system.switch\_controller.SwitchController method), 82
- set\_value() (mpf.devices.score\_reel.ScoreReelGroup method), 19
- set\_volume() (mpf.media\_controller.core.sound.SoundController method), 29
- set\_volume() (mpf.system.bcp.BCP method), 59
- setup\_client() (mpf.plugins.socket\_events.SocketClient method), 51
- setup\_client\_socket() (mpf.system.bcp.BCPClientSocket method), 60
- setup\_defaults() (mpf.system.assets.AssetManager method), 54
- setup\_osc\_client() (mpf.plugins.osc.OSC method), 51
- setup\_player\_controlled\_eject() (mpf.devices.playfield.Playfield method), 14
- Show (class in mpf.media\_controller.core.show\_controller), 22

Show (class in mpf.system.light\_controller), 72  
show() (mpf.media\_controller.core.slide.Slide method), 26  
ShowController (class in mpf.media\_controller.core.show\_controller), 23  
shutdown() (mpf.system.bcp.BCP method), 59  
sink\_bank() (mpf.platform.p3\_roc.PDBLight method), 39  
sink\_bank() (mpf.platform.p\_roc.PDBLight method), 44  
sink\_board() (mpf.platform.p3\_roc.PDBLight method), 40  
sink\_board() (mpf.platform.p\_roc.PDBLight method), 44  
sink\_output() (mpf.platform.p3\_roc.PDBLight method), 40  
sink\_output() (mpf.platform.p\_roc.PDBLight method), 44  
Slide (class in mpf.media\_controller.core.slide), 24  
SlideBuilder (class in mpf.media\_controller.core.slide\_builder), 27  
SocketClient (class in mpf.plugins.socket\_events), 51  
Sound (class in mpf.media\_controller.core.sound), 28  
sound\_is\_done() (mpf.media\_controller.core.sound.Channel method), 28  
SoundController (class in mpf.media\_controller.core.sound), 28  
source\_bank() (mpf.platform.p3\_roc.PDBLight method), 40  
source\_bank() (mpf.platform.p\_roc.PDBLight method), 44  
source\_board() (mpf.platform.p3\_roc.PDBLight method), 40  
source\_board() (mpf.platform.p\_roc.PDBLight method), 44  
source\_output() (mpf.platform.p3\_roc.PDBLight method), 40  
source\_output() (mpf.platform.p\_roc.PDBLight method), 45  
split\_matrix\_addr\_parts() (mpf.platform.p3\_roc.PDBLight method), 40  
split\_matrix\_addr\_parts() (mpf.platform.p\_roc.PDBLight method), 45  
start() (mpf.plugins.ball\_search.BallSearch method), 49  
start() (mpf.plugins.osc.OSC method), 51  
start() (mpf.system.light\_controller.Playlist method), 71  
state() (mpf.platform.p3\_roc.PROCDriver method), 40  
state() (mpf.platform.p\_roc.PROCDriver method), 46  
state() (mpf.platform.virtual.VirtualDriver method), 47  
status\_dump() (mpf.devices.ball\_device.BallDevice method), 3  
step\_settings() (mpf.system.light\_controller.Playlist method), 71  
stop() (mpf.devices.ball\_device.BallDevice method), 3  
stop() (mpf.media\_controller.core.show\_controller.Show method), 23  
stop() (mpf.media\_controller.core.sound.Sound method), 28  
stop() (mpf.media\_controller.core.sound.StreamTrack method), 30  
stop() (mpf.media\_controller.core.sound.Track method), 31  
stop() (mpf.platform.fast.SerialCommunicator method), 36  
stop() (mpf.plugins.osc.OSC method), 51  
stop() (mpf.system.bcp.BCPCliSocket method), 60  
stop() (mpf.system.light\_controller.Playlist method), 71  
stop() (mpf.system.light\_controller.Show method), 73  
stop() (mpf.system.tasks.Task method), 84  
stop\_client() (mpf.plugins.socket\_events.SocketClient method), 51  
stop\_ignoring\_hits() (mpf.system.logic\_blocks.Counter method), 74  
stop\_script() (mpf.system.light\_controller.LightController method), 69  
stop\_show() (mpf.media\_controller.core.show\_controller.ShowController method), 24  
stop\_show() (mpf.system.light\_controller.LightController method), 69  
stop\_shows\_by\_key() (mpf.media\_controller.core.show\_controller.ShowController method), 24  
stop\_shows\_by\_key() (mpf.system.light\_controller.LightController method), 69  
stop\_shows\_by\_keys() (mpf.system.light\_controller.LightController method), 69  
StreamTrack (class in mpf.media\_controller.core.sound), 29  
string\_to\_class() (mpf.system.machine.MachineController method), 77  
string\_to\_list() (mpf.system.config.Config static method), 63  
string\_to\_lowercase\_list() (mpf.system.config.Config static method), 63  
string\_to\_ms() (mpf.system.timing.Timing static method), 85  
string\_to\_secs() (mpf.system.timing.Timing static method), 85  
string\_to\_ticks() (mpf.system.timing.Timing static method), 85  
surface (mpf.media\_controller.core.slide.Slide attribute), 24  
sw\_flip() (mpf.devices.flipper.Flipper method), 9  
sw\_release() (mpf.devices.flipper.Flipper method), 9  
Switch (class in mpf.devices.switch), 19  
SwitchController (class in mpf.system.switch\_controller), 80  
SwitchPlayer (class in mpf.plugins.switch\_player), 52  
sync\_ms\_next\_tick() (mpf.system.light\_controller.LightController method), 69

## T

Task (class in mpf.system.tasks), 83  
 Tasks (mpf.system.tasks.Task attribute), 83  
 text() (mpf.media\_controller.core.language.Language method), 22  
 tick (mpf.system.timing.Timing attribute), 85  
 tick() (mpf.devices.score\_reel.ScoreReelGroup method), 19  
 tick() (mpf.platform.fast.FASTDMD method), 32  
 tick() (mpf.platform.fast.HardwarePlatform method), 35  
 tick() (mpf.platform.p3\_roc.HardwarePlatform method), 38  
 tick() (mpf.platform.p3\_roc.PROCDriver method), 40  
 tick() (mpf.platform.p\_roc.HardwarePlatform method), 42  
 tick() (mpf.platform.p\_roc.PROCDMD method), 45  
 tick() (mpf.platform.p\_roc.PROCDriver method), 46  
 tick() (mpf.platform.virtual.VirtualDriver method), 47  
 tick() (mpf.plugins.ball\_search.BallSearch method), 49  
 tick() (mpf.system.platform.Platform method), 79  
 tilt() (mpf.plugins.info\_lights.InfoLights method), 50  
 tilted (mpf.system.machine.MachineController attribute), 76  
 timed\_pwm() (mpf.devices.driver.Driver method), 6  
 timed\_pwm() (mpf.platform.p3\_roc.PROCDriver method), 40  
 timed\_pwm() (mpf.platform.p\_roc.PROCDriver method), 46  
 Timer (class in mpf.system.timing), 84  
 timer\_initialize() (mpf.system.platform.Platform method), 79  
 timer\_tick() (mpf.system.machine.MachineController method), 77  
 timer\_tick() (mpf.system.tasks.DelayManager static method), 83  
 timer\_tick() (mpf.system.tasks.Task static method), 84  
 timer\_tick() (mpf.system.timing.Timing method), 85  
 Timing (class in mpf.system.timing), 84  
 Track (class in mpf.media\_controller.core.sound), 30

## U

unlight() (mpf.devices.score\_reel.ScoreReelGroup method), 19  
 unload() (mpf.system.assets.Asset method), 52  
 unload() (mpf.system.logic\_blocks.LogicBlock method), 75  
 unload\_assets() (mpf.system.assets.AssetManager method), 54  
 unload\_event\_player\_events() (mpf.system.events.EventManager method), 67  
 unload\_light\_player\_shows() (mpf.system.light\_controller.LightController method), 70

unload\_score\_events() (mpf.system.scoring.ScoreController method), 80  
 unload\_show\_player\_shows() (mpf.media\_controller.core.show\_controller.ShowController method), 24  
 unload\_slide\_events() (mpf.media\_controller.core.slide\_builder.SlideBuilder method), 27  
 unpause() (mpf.media\_controller.core.sound.StreamTrack method), 30  
 unregister\_score\_event() (mpf.system.scoring.ScoreController method), 80  
 unregister\_sound\_event() (mpf.media\_controller.core.sound.SoundController method), 29  
 unregister\_sound\_events() (mpf.media\_controller.core.sound.SoundController method), 29  
 update() (mpf.media\_controller.core.slide.Slide method), 26  
 update() (mpf.platform.fast.FASTDMD method), 32  
 update() (mpf.platform.p\_roc.PROCDMD method), 45  
 update() (mpf.platform.virtual.VirtualDMD method), 47  
 update\_audits() (mpf.plugins.osc.OSC method), 51  
 update\_ball() (mpf.plugins.osc.OSC method), 51  
 update\_config() (mpf.plugins.osc.OSC method), 51  
 update\_leds() (mpf.platform.fast.HardwarePlatform method), 35  
 update\_player() (mpf.plugins.osc.OSC method), 51  
 update\_score() (mpf.plugins.osc.OSC method), 51  
 update\_switches\_from\_hw() (mpf.system.switch\_controller.SwitchController method), 82

## V

validate() (mpf.devices.autofire.AutofireCoil method), 1  
 validate() (mpf.devices.score\_reel.ScoreReelGroup method), 19  
 validate\_config\_item() (mpf.system.config.Config static method), 63  
 validate\_config\_item2() (mpf.system.config.Config method), 63  
 validate\_item() (mpf.system.config.Config method), 63  
 validation\_error() (mpf.system.config.Config method), 63  
 verify\_switches() (mpf.system.switch\_controller.SwitchController method), 82  
 VirtualDMD (class in mpf.platform.virtual), 46  
 VirtualDriver (class in mpf.platform.virtual), 47  
 VirtualGI (class in mpf.platform.virtual), 47  
 VirtualLED (class in mpf.platform.virtual), 47  
 VirtualMatrixLight (class in mpf.platform.virtual), 47  
 VirtualSwitch (class in mpf.platform.virtual), 47

## W

wait() (mpf.system.events.QueuedEvent method), 67

`width` (`mpf.media_controller.core.slide.Slide` attribute),  
25  
`write_hw_rule()` (`mpf.platform.fast.HardwarePlatform`  
method), 35  
`write_hw_rule()` (`mpf.platform.p3_roc.HardwarePlatform`  
method), 38  
`write_hw_rule()` (`mpf.platform.p_roc.HardwarePlatform`  
method), 43  
`write_hw_rule()` (`mpf.platform.virtual.HardwarePlatform`  
method), 46  
`write_hw_rule()` (`mpf.system.platform.Platform` method),  
79

## Y

`YourNewDevice` (class in  
`mpf.devices.new_device_template`), 12