MPF API Documentation

Release 0.30.32

The Mission Pinball Framework Team

September 20, 2016

Contents

This is the developer / API reference for The Mission Pinball Framework (MPF) 0.30.

Contents:

CHAPTER 1

mpf

Contains "mpf" module path of the Mission Pinball Framework (MPF) .

1.1 Submodules

1.1.1 `mpf.__main__`

Command dispatcher of the Mission Pinball Framework.

`mpf.__main__.`**`main`**`(`*args=None*`)`
Dispatche commands to our handlers.

1.1.2 `mpf._version`

Version string of MPF.

This modules holds the MPF version strings, including the version of BCP it needs and the config file version it needs.

It's used internally for all sorts of things, from printing the output of the *mpf –version* command, to making sure any processes connected via BCP are the proper versions, to automatically triggering new builds and deployments to PyPI.

1.1.3 `mpf.assets`

1.1.4 `mpf.commands`

Submodules

`mpf.commands.both`

`mpf.commands.core`

Deprecated command to start MPF game engine.

1.1.5 `mpf.config_players`

Submodules

`mpf.config_players.coil_player`

Coil config player.

class `mpf.config_players.coil_player.`**`CoilPlayer`**(*machine*)
Triggers coils based on config.

Initialise config player.

`clear_context`(*context*)
Disable enabled coils.

`config_play_callback`(*settings*, *priority=0*, *mode=None*, ***kwargs*)
Callback for standalone player.

`get_express_config`(*value*)
Parse short config version.

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

`mode_start`(*config*, *priority*, *mode*)
Add events for mode.

`mode_stop`(*mode*)
Remove events for mode.

`play`(*settings*, *context*, *priority=0*, ***kwargs*)
Enable, Pulse or disable coils.

`process_config`(*config*, ***kwargs*)
Process system-wide config.

Called every time mpf starts, regardless of whether config was built from cache or config files.

`process_mode_config`(*config*, *root_config_dict*, *mode*, ***kwargs*)
Parse mode config.

`register_player_events`(*config*, *mode=None*, *priority=0*)
Register events for standalone player.

show_play_callback (*settings*, *priority*, *show_tokens*, *context*)
Callback if used in a show.

show_stop_callback (*context*)
Callback if show stops.

unload_player_events (*key_list*)
Remove event for standalone player.

validate_config (*config*)
Validate this player's section of a config file (either a machine-wide config or a mode config).

Parameters

- **config** – A dict of the contents of this config_player's section
- **the config file. It's assumed that keys are event names, and** (*from*) –
- **are settings for what this config_player does when that** (*values*) –
- **is posted.** (*event*) –

Returns: A dict in the same format, but passed through the config validator.

validate_show_config (*device*, *device_settings*)
Validate show config.

mpf.config_players.coil_player.**player_cls**
alias of `CoilPlayer`

mpf.config_players.flasher_player

Flasher config player.

class mpf.config_players.flasher_player.**FlasherPlayer** (*machine*)
Triggers flashers based on config.

Initialise config player.

clear_context (*context*)
Clear the context.

config_play_callback (*settings*, *priority=0*, *mode=None*, ***kwargs*)
Callback for standalone player.

get_express_config (*value*)
Parse express config.

get_full_config (*value*)
Return full config.

mode_start (*config*, *priority*, *mode*)
Add events for mode.

mode_stop (*mode*)
Remove events for mode.

play (*settings*, *context*, *priority=0*, ***kwargs*)
Flash flashers.

process_config(*config*, ***kwargs*)
Process system-wide config.

Called every time mpf starts, regardless of whether config was built from cache or config files.

process_mode_config(*config*, *root_config_dict*, *mode*, ***kwargs*)
Parse mode config.

register_player_events(*config*, *mode=None*, *priority=0*)
Register events for standalone player.

show_play_callback(*settings*, *priority*, *show_tokens*, *context*)
Callback if used in a show.

show_stop_callback(*context*)
Callback if show stops.

unload_player_events(*key_list*)
Remove event for standalone player.

validate_config(*config*)
Validate this player's section of a config file (either a machine-wide config or a mode config).

Parameters

- **config** – A dict of the contents of this config_player's section
- **the config file. It's assumed that keys are event names, and** (*from*) –
- **are settings for what this config_player does when that** (*values*) –
- **is posted.** (*event*) –

Returns: A dict in the same format, but passed through the config validator.

validate_show_config(*device*, *device_settings*)
Validate show config.

mpf.config_players.flasher_player.**player_cls**
alias of *FlasherPlayer*

mpf.config_players.gi_player

GI config player.

class mpf.config_players.gi_player.**GiPlayer**(*machine*)
Enables GIs based on config.

Initialise config player.

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

config_play_callback(*settings*, *priority=0*, *mode=None*, ***kwargs*)
Callback for standalone player.

get_express_config(*value*)
Parse express config.

get_full_config (*value*)
Return full config.

mode_start (*config*, *priority*, *mode*)
Add events for mode.

mode_stop (*mode*)
Remove events for mode.

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

process_config (*config*, ***kwargs*)
Process system-wide config.

Called every time mpf starts, regardless of whether config was built from cache or config files.

process_mode_config (*config*, *root_config_dict*, *mode*, ***kwargs*)
Parse mode config.

register_player_events (*config*, *mode=None*, *priority=0*)
Register events for standalone player.

show_play_callback (*settings*, *priority*, *show_tokens*, *context*)
Callback if used in a show.

show_stop_callback (*context*)
Callback if show stops.

unload_player_events (*key_list*)
Remove event for standalone player.

validate_config (*config*)
Validate this player's section of a config file (either a machine-wide config or a mode config).

Parameters

- **config** – A dict of the contents of this config_player's section
- **the config file. It's assumed that keys are event names, and** (*from*) –
- **are settings for what this config_player does when that** (*values*) –
- **is posted.** (*event*) –

Returns: A dict in the same format, but passed through the config validator.

validate_show_config (*device*, *device_settings*)
Validate show config.

mpf.config_players.gi_player.**player_cls**
alias of *GiPlayer*

mpf.config_players.plugin_player

Plugin config player.

class mpf.config_players.plugin_player.**PluginPlayer** (*machine*)
Base class for a remote ConfigPlayer that is registered as a plug-in to MPF.

This class is created on the MPF side of things.

Initialise config player.

`clear_context` (*context*)
Clear the context at remote player via BCP.

`config_play_callback` (*settings*, *priority=0*, *mode=None*, ***kwargs*)
Callback for standalone player.

`get_express_config` (*value*)
Not supported.

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

`mode_start` (*config*, *priority*, *mode*)
Add events for mode.

`mode_stop` (*mode*)
Remove events for mode.

`play` (*settings*, *context*, *priority=0*, ***kwargs*)
Trigger remote player via BCP.

`process_config` (*config*, ***kwargs*)
Process system-wide config.

Called every time mpf starts, regardless of whether config was built from cache or config files.

`process_mode_config` (*config*, *root_config_dict*, *mode*, ***kwargs*)
Parse mode config.

`register_player_events` (*config*, *mode=None*, *priority=0*)
Register player events via BCP.

Override this method in the base class and registers the config_player events to send the trigger via BCP instead of calling the local play() method.

`show_play_callback` (*settings*, *priority*, *show_tokens*, *context*)
Callback if used in a show.

`show_stop_callback` (*context*)
Callback if show stops.

`unload_player_events` (*event_list*)
Unload player events via BCP.

`validate_config` (*config*)
Validate this player's section of a config file (either a machine-wide config or a mode config).

Parameters

- **`config`** – A dict of the contents of this config_player's section
- **`the config file. It's assumed that keys are event names, and`** (*`from`*) –
- **`are settings for what this config_player does when that`** (*`values`*) –
- **`is posted.`** (*`event`*) –

Returns: A dict in the same format, but passed through the config validator.

validate_show_config(*device*, *device_settings*)
Validate show config.

mpf.config_players.show_player

Show config player.

class mpf.config_players.show_player.**ShowPlayer**(*machine*)
Plays, starts, stops, pauses, resumes or advances shows based on config.

Initialise config player.

clear_context(*context*)
Stop running shows from context.

config_play_callback(*settings*, *priority=0*, *mode=None*, ***kwargs*)
Callback for standalone player.

get_express_config(*value*)
Parse express config.

get_full_config(*value*)
Return full config.

mode_start(*config*, *priority*, *mode*)
Add events for mode.

mode_stop(*mode*)
Remove events for mode.

play(*settings*, *context*, *priority=0*, ***kwargs*)
Play, start, stop, pause, resume or advance show based on config.

process_config(*config*, ***kwargs*)
Process system-wide config.

Called every time mpf starts, regardless of whether config was built from cache or config files.

process_mode_config(*config*, *root_config_dict*, *mode*, ***kwargs*)
Parse mode config.

register_player_events(*config*, *mode=None*, *priority=0*)
Register events for standalone player.

show_play_callback(*settings*, *priority*, *show_tokens*, *context*)
Callback if used in a show.

show_stop_callback(*context*)
Callback if show stops.

unload_player_events(*key_list*)
Remove event for standalone player.

validate_config(*config*)
Validate this player's section of a config file (either a machine-wide config or a mode config).

Parameters

- **config** – A dict of the contents of this config_player's section
- **the config file. It's assumed that keys are event names, and** (*from*) –

- **`are settings for what this config_player does when that`** (*`values`*) –
- **`is posted.`** (*`event`*) –

Returns: A dict in the same format, but passed through the config validator.

`validate_show_config`(*device*, *device_settings*)
Validate show config.

`mpf.config_players.show_player.`**`player_cls`**
alias of *`ShowPlayer`*

`mpf.config_players.trigger_player`

Trigger config player.

class `mpf.config_players.trigger_player.`**`TriggerPlayer`**(*machine*)
Executes BCP triggers based on config.

Initialise config player.

`clear_context`(*context*)
Clear the context.

`config_play_callback`(*settings*, *priority=0*, *mode=None*, ***kwargs*)
Callback for standalone player.

`get_express_config`(*value*)
Not supported.

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

`mode_start`(*config*, *priority*, *mode*)
Add events for mode.

`mode_stop`(*mode*)
Remove events for mode.

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

`process_config`(*config*, ***kwargs*)
Process system-wide config.

Called every time mpf starts, regardless of whether config was built from cache or config files.

`process_mode_config`(*config*, *root_config_dict*, *mode*, ***kwargs*)
Parse mode config.

`register_player_events`(*config*, *mode=None*, *priority=0*)
Register events for standalone player.

`show_play_callback`(*settings*, *priority*, *show_tokens*, *context*)
Callback if used in a show.

`show_stop_callback`(*context*)
Callback if show stops.

`unload_player_events`(*key_list*)
Remove event for standalone player.

validate_config(*config*)
Validate this player's section of a config file (either a machine-wide config or a mode config).

Parameters

- **config** – A dict of the contents of this config_player's section
- **the config file. It's assumed that keys are event names, and** (*from*) –
- **are settings for what this config_player does when that** (*values*) –
- **is posted.** (*event*) –

Returns: A dict in the same format, but passed through the config validator.

validate_show_config(*device*, *device_settings*)
Validate show config.

`mpf.config_players.trigger_player.`**player_cls**
alias of *TriggerPlayer*

1.1.6 mpf.core

- *Submodules*

Submodules

mpf.core.ball_search

Implements the ball search procedure

mpf.core.case_insensitive_dict

mpf.core.config_player

Base class used for things that "play" from the config files, such as WidgetPlayer, SlidePlayer, etc.

class `mpf.core.config_player.`**ConfigPlayer**(*machine*)
Base class for players which play things based on config.

Initialise config player.

clear_context(*context*)
Clear the context.

config_play_callback(*settings*, *priority=0*, *mode=None*, ***kwargs*)
Callback for standalone player.

get_express_config(*value*)
Parse short config version.

Implements "express" settings for this config_player which is what happens when a config is passed as a string instead of a full config dict. (This is detected automatically and this method is only called when the config is not a dict.)

For example, the led_player uses the express config to parse a string like 'ff0000-f.5s' and translate it into:

color: 220000 fade: 500

Since every config_player is different, this method raises a NotImplementedError and most be configured in the child class.

Parameters **`value`** – The single line string value from a config file.

Returns A dictionary (which will then be passed through the config validator)

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

`mode_start`(*config*, *priority*, *mode*)
Add events for mode.

`mode_stop`(*mode*)
Remove events for mode.

`play`(*settings*, *context*, *priority=0*, ***kwargs*)
Directly play player.

classmethod `process_config`(*config*, ***kwargs*)
Process system-wide config.

Called every time mpf starts, regardless of whether config was built from cache or config files.

`process_mode_config`(*config*, *root_config_dict*, *mode*, ***kwargs*)
Parse mode config.

`register_player_events`(*config*, *mode=None*, *priority=0*)
Register events for standalone player.

`show_play_callback`(*settings*, *priority*, *show_tokens*, *context*)
Callback if used in a show.

`show_stop_callback`(*context*)
Callback if show stops.

`unload_player_events`(*key_list*)
Remove event for standalone player.

`validate_config`(*config*)
Validate this player's section of a config file (either a machine-wide config or a mode config).

Parameters

- **`config`** – A dict of the contents of this config_player's section
- **`the config file. It's assumed that keys are event names, and`** (*`from`*) –
- **`are settings for what this config_player does when that`** (*`values`*) –
- **`is posted.`** (*`event`*) –

Returns: A dict in the same format, but passed through the config validator.

validate_show_config(*device*, *device_settings*)
Validate show config.

mpf.core.delays

Delays MPF contains a Delay Manager.

class mpf.core.delays.**DelayManager**(*registry*)
Handles delays for one object

add(*ms*, *callback*, *name=None*, ***kwargs*)
Adds a delay.

Parameters

- **ms** – Int of the number of milliseconds you want this delay to be for. Note that the resolution of this time is based on your machine's tick rate. The callback will be called on the first machine tick *after* the delay time has expired. For example, if you have a machine tick rate of 30Hz, that's 33.33ms per tick. So if you set a delay for 40ms, the actual delay will be 66.66ms since that's the next tick time after the delay ends.
- **callback** – The method that is called when this delay ends.
- **name** – String name of this delay. This name is arbitrary and only used to identify the delay later if you want to remove or change it. If you don't provide it, a UUID4 name will be created.
- ****kwargs** – Any other (optional) kwarg pairs you pass will be passed along as kwargs to the callback method.

Returns String name of the delay which you can use to remove it later.

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.

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*) –

mpf.core.events

Contains the base classes for the EventManager and QueuedEvents

class mpf.core.events.**QueuedEvent**(*callback*, ***kwargs*)
Base class for an event queue which is created each time a queue event is called.

clear()
Clears a wait. If the number of waits drops to 0, the callbacks will be called.

is_empty()
Checks to see if this QueuedEvent has any waits.

Returns True is there are 1 or more waits, False if there are no more waits.

kill()
Kills this QueuedEvent by removing all waits. Does not process the callback.

wait()
Registers a wait for this QueueEvent.

mpf.core.mode_timer

class mpf.core.mode_timer.**ModeTimer**(*machine*, *mode*, *name*, *config*)
Parent class for a mode timer.

Parameters

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

add_time(*timer_value*, ***kwargs*)
Adds ticks to this timer.

Parameters

- **timer_value** – The number of ticks you want to add to this timer's current value.
- **kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

change_tick_interval(*change=0.0*, ***kwargs*)
Changes the interval for each "tick" of this timer.

Parameters

- **change** – Float or int of the change you want to make to this timer's tick rate. Note this value is added to the current tick interval. To set an absolute value, use the set_tick_interval() method. To shorten the tick rate, use a negative value.
- ****kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

kill()
Stops this timer and also removes all the control events.

pause(*timer_value=0*, ***kwargs*)
Pauses the timer and posts the 'timer_<name>_paused' event

Parameters

- **timer_value** – How many seconds you want to pause the timer for. Note that this pause time is real-world seconds and does not take into consideration this timer's tick interval.
- ****kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

reset(***kwargs*)
Resets this timer based to the starting value that's already been configured. Does not start or stop the timer.

Parameters ****kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

restart(***kwargs*)
Restarts the timer by resetting it and then starting it. Essentially this is just a reset() then a start()

Parameters ****kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

set_current_time(*timer_value*, ***kwargs*)
Sets the current amount of time of this timer. This value is expressed in "ticks" since the interval per tick can be something other than 1 second).

Parameters

- **timer_value** – Integer of the current value you want this timer to be.
- ****kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

set_tick_interval(*timer_value*, ***kwargs*)
Sets the number of seconds between ticks for this timer. This is an absolute setting. To apply a change to the current value, use the change_tick_interval() method.

Parameters

- **timer_value** – The new number of seconds between each tick of this timer. This value should always be positive.
- ****kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

start(***kwargs*)
Starts this timer based on the starting value that's already been configured. Use set_current_time() if you want to set the starting time value.

Parameters ****kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

stop(***kwargs*)
Stops the timer and posts the 'timer_<name>_stopped' event.

Parameters ****kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

subtract_time(*timer_value*, ***kwargs*)
Subtracts ticks from this timer.

Parameters

- **timer_value** – The number of ticks you want to subtract from this timer's current value.
- ****kwargs** – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

timer_complete(***kwargs*)
Automatically called when this timer completes. Posts the 'timer_<name>_complete' event. Can be manually called to mark this timer as complete.

Parameters `**kwargs` – Not used in this method. Only exists since this method is often registered as an event handler which may contain additional keyword arguments.

`mpf.core.player`

Contains the Player class which represents a player in a pinball game.

class `mpf.core.player.`**`Player`**(*machine*, *player_list*)

Base class for a player. One instance of this class is created for each player.

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

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

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

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

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

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

For the 'change' parameter, it will attempt to subtract the old value from the new value. If that works, it will return the result as the change. If it doesn't work (like if you're not storing numbers in this variable), then the change paramter will be True if the new value is different and False if the value didn't change.

Some examples:

player.score = 0

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

player.score += 500

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

player.score = 1200

Event posted: 'player_score' with Args: value=1200, change=700, prev_value=500

`monitor_enabled` = **False**

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

`mpf.core.scoring`

MPF plugin for a score controller which handles all scoring and bonus tracking.

`mpf.core.scriptlet`

Contains the parent class for Scriptlets.

`mpf.core.shot_profile_manager`

Contains the ShotProfileManager class.

`mpf.core.weakmethod`

Weak Method The *`WeakMethod`* is used by the `Clock` class to allow references to a bound method that permits the associated object to be garbage collected. Please refer to *examples/core/clock_method.py* for more information. This WeakMethod class is taken from the recipe http://code.activestate.com/recipes/81253/, based on the nicodemus version. Many thanks nicodemus!

class `mpf.core.weakmethod.`**`WeakMethod`**`(`*method*`)`

Implementation of a weakref for functions and bound methods.

`is_dead``()`

Returns True if the referenced callable was a bound method and the instance no longer exists. Otherwise, return False.

1.1.7 `mpf.devices`

Submodules

`mpf.devices.score_reel_controller`

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

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.

Known limitations of this module:

- Assumes all score reels include a zero value.
- Assumes all score reels count up or down by one.
- Assumes all score reels map their displayed value to their stored value in a 1:1 way. (i.e. value[0] displays 0, value[5] displays 5, etc.
- Currently this module only supports “incrementing” reels (i.e. counting up). Decrementing support will be added in the future.

`active_scorereelgroup` **= None**

Pointer to the active ScoreReelGroup for the current player.

`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.

player_to_scorereel_map = None
This is a list of ScoreReelGroup objects which corresponds to player indexes. The first element [0] in this list is the first player (which is player index [0], the next one is the next player, etc.

queue = None
Holds any active queue event queue objects

reset_queue = None
List of score reel groups that still need to be reset

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_scorereel_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(*value*, *change*, ***kwargs*)
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** – Interget value of the change to the score.

1.1.8 `mpf.file_interfaces`

1.1.9 `mpf.migrator`

Submodules

`mpf.migrator.config_version_3`

1.1.10 `mpf.modes`

Submodules

mpf.modes.attract

Submodules

mpf.modes.attract.code

mpf.modes.bonus

Contains a bonus mode.

Submodules

mpf.modes.bonus.code	Contains a bonus mode.

mpf.modes.carousel

Submodules

mpf.modes.carousel.code

mpf.modes.credits

Submodules

`mpf.modes.credits.code`

`mpf.modes.game`

Submodules

`mpf.modes.game.code`

`mpf.modes.high_score`

Submodules

`mpf.modes.high_score.code`

`mpf.modes.tilt`

Submodules

`mpf.modes.tilt.code`

1.1.11 `mpf.platforms`

Submodules

`mpf.platforms.fast`

Submodules

`mpf.platforms.fast.fast_defines`

`mpf.platforms.interfaces`

Package with interfaces for hardware platform devices.

`mpf.platforms.opp`

Submodules

`mpf.platforms.opp.opp_rs232_intf`

1.1.12 `mpf.plugins`

Submodules

`mpf.plugins.osc`

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

This mode requires pyOSC, https://trac.v2.nl/wiki/pyOSC It was written for pyOSC 0.3.5b build 5394, though later versions should work.

1.1.13 `mpf.tests`

CHAPTER 2

Indices and tables

- genindex
- modindex
- search