MPF API Documentation

Release 0.32.8

The Mission Pinball Framework Team

May 04, 2017

Contents

This is the developer / API reference for The Mission Pinball Framework (MPF) 0.32.

Contents:

CHAPTER 1

mpf

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

Submodules

mpf.__main__

Command dispatcher of the Mission Pinball Framework.

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

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.

mpf.assets

Contains the asset classes.

mpf.commands

Cli commands in MPF.

class mpf.commands.**CommandLineUtility**(*path=None*)
Default cli entry point.

Initialise cli entry point.

classmethod check_python_version()
Check that we have at least python 3.

execute()
Actually run the command that was just set up.

get_external_commands()
Entry point to hook more commands.

This is used from mpf mc.

get_machine_path(*machine_path_hint*)
Return machine path.

parse_args()
Parse arguments.

mpf.commands.**run_from_command_line**(*args=None*)
Run cli command.

Submodules

mpf.commands.both

Runs mc and game.

class mpf.commands.both.**Command**(*mpf_path*, *machine_path*, *args*)
Command which runs game and mc.

Run game and mc.

mpf.commands.core

Deprecated command to start MPF game engine.

class mpf.commands.core.**Command**(*mpf_path*, *machine_path*, *args*)
Deprecated command.

Show error.

mpf.config_players

Config players which can control devices based on hardware.

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*, *calling_context*, *priority=0*, *mode=None*, ***kwargs*)
Callback for standalone player.

get_express_config(*value*)
Parse short config version.

get_full_config(*value*)
Return full config.

get_list_config(*value*)
Parse config list.

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

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

mpf.config_players.device_config_player

Base class for config players which have multiple entries.

class mpf.config_players.device_config_player.**DeviceConfigPlayer**(*machine*)
Base class for config players which have multiple entries.

Initialise config player.

clear_context(*context*)
Clear the context.

config_play_callback(*settings*, *calling_context*, *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.

get_list_config(*value*)
Parse config list.

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

mode_stop(*mode*)
Remove events for mode.

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

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

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*, *calling_context*, *priority=0*, *mode=None*, ***kwargs*)
Callback for standalone player.

get_express_config(*value*)
Parse express config.

get_full_config(*value*)
Return full config.

get_list_config(*value*)
Parse config list.

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

mode_stop(*mode*)
Remove events for mode.

play(*settings*, *context*, *calling_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*, *calling_context*, *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_config_entry(*settings*, *name*)
Validate one entry of this player.

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

mpf.config_players.flat_config_player

Base class for flat config players.

class mpf.config_players.flat_config_player.**FlatConfigPlayer**(*machine*)
Flat show players.

Initialise config player.

clear_context(*context*)
Clear the context.

config_play_callback(*settings*, *calling_context*, *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.

get_list_config(*value*)
Parse config list.

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

mode_stop(*mode*)
Remove events for mode.

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

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

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*, *calling_context*, *priority=0*, *mode=None*, ***kwargs*)
Callback for standalone player.

get_express_config(*value*)
Parse express config.

get_full_config(*value*)
Return full config.

get_list_config(*value*)
Parse config list.

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

mode_stop(*mode*)
Remove events for mode.

play(*settings*, *context*, *calling_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*, *calling_context*, *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_config_entry(*settings*, *name*)
Validate one entry of this player.

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 plugin player.

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

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

get_express_config(*value*)
Not supported.

get_full_config(*value*)
Return full config.

get_list_config(*value*)
Parse config list.

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

mode_stop(*mode*)
Remove events for mode.

play(*settings*, *context*, *calling_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*, *calling_context*, *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_config_entry(*settings*, *name*)
Validate one entry of this player.

mpf.config_players.queue_event_player

Queue Event Config Player.

class mpf.config_players.queue_event_player.**QueueEventPlayer**(*machine*)
Posts queue events based on config.

Initialise config player.

clear_context(*context*)
Clear the context.

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

get_express_config(*value*)
No express config.

get_full_config(*value*)
Return full config.

get_list_config(*value*)
Parse config list.

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

mode_stop(*mode*)
Remove events for mode.

play(*settings*, *context*, *priority=0*, ***kwargs*)
Post queue events.

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

mpf.config_players.queue_event_player.**player_cls**
alias of *QueueEventPlayer*

mpf.config_players.queue_relay_player

Queue Relay Config Player.

class mpf.config_players.queue_relay_player.**QueueRelayPlayer**(*machine*)
Blocks queue events and converts them to normal events.

Initialise config player.

clear_context(*context*)
Clear all queues.

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

get_express_config(*value*)
No express config.

get_full_config(*value*)
Return full config.

get_list_config(*value*)
Parse config list.

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

mode_stop(*mode*)
Remove events for mode.

play(*settings*, *context*, *priority=0*, ***kwargs*)
Block queue event.

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

mpf.config_players.queue_relay_player.**player_cls**
alias of *QueueRelayPlayer*

mpf.config_players.score_player

Scoring Config Player.

class mpf.config_players.score_player.**ScorePlayer**(*machine*)
Posts events based on config.

Initialise score player.

clear_context(*context*)
Clear context.

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

get_express_config(*value*)
Parse express config.

get_full_config(*value*)
Return full config.

get_list_config(*value*)
Parse list.

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

mode_stop(*mode*)
Remove events for mode.

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

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

mpf.config_players.score_player.**player_cls**
alias of *ScorePlayer*

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*, *calling_context*, *priority=0*, *mode=None*, ***kwargs*)
Callback for standalone player.

get_express_config(*value*)
Parse express config.

get_full_config(*value*)
Return full config.

get_list_config(*value*)
Parse config list.

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

mode_stop(*mode*)
Remove events for mode.

play(*settings*, *context*, *calling_context*, *priority=0*, *queue=None*, ***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*, *calling_context*, *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_config_entry(*settings*, *name*)
Validate one entry of this player.

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*, *calling_context*, *priority=0*, *mode=None*, ***kwargs*)
Callback for standalone player.

get_express_config(*value*)
Not supported.

get_full_config(*value*)
Return full config.

get_list_config(*value*)
Parse config list.

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

mode_stop(*mode*)
Remove events for mode.

play(*settings*, *context*, *calling_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*, *calling_context*, *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_config_entry(*settings*, *name*)
Validate one entry of this player.

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

mpf.core

Core modules in MPF.

Submodules

mpf.core.ball_search

Implements the ball search procedure.

class mpf.core.ball_search.**BallSearch**(*machine*, *playfield*)
Ball search controller.

Initialise ball search.

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

disable()
Disable ball search.

Will stop the ball search if it is running.

enable()
Enable but do not start ball search.

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

give_up()
Give up the ball search.

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

register(*priority*, *callback*)
Register a callback for sequential ballsearch.

Callbacks are called by priority. Ball search only waits if the callback returns true.

Parameters

- **priority** – priority of this callback in the ball search procedure
- **callback** – callback to call. ball search will wait before the next callback, if it returns true

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

Returns false if the ball search is running.

reset_timer()
Reset the start timer.

Called by playfield.

run()
Run one iteration of the ball search.

Will schedule itself for the next run.

start()
Actually start ball search.

mpf.core.bcp

Contains the BCP communications classes.

Submodules

mpf.core.bcp.bcp_client

Base class for all bcp clients.

class mpf.core.bcp.bcp_client.**BaseBcpClient**(*machine*, *name*, *bcp*)
Base class for bcp clients.

Initialise client.

accept_connection(*receiver*, *sender*)
Created client for incoming connection.

connect(*config*)
Actively connect client.

read_message()
Read one message from client.

send(*bcp_command*, *kwargs*)
Send data to client.

stop()
Stop client connection.

mpf.core.bcp.bcp_socket_client

BCP socket client.

class mpf.core.bcp.bcp_socket_client.**BCPClientSocket**(*machine*, *name*, *bcp*)
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.
- **bcp** – The bcp object.

Initialise BCP client socket.

accept_connection(*receiver*, *sender*)
Create client for incoming connection.

connect(*config*)
Actively connect to server.

read_message()
Read the next message.

send(*bcp_command*, *bcp_command_args*)
Send a message to the BCP host.

Parameters

- **bcp_command** – command to send
- **bcp_command_args** – parameters to command

send_goodbye()
Send BCP 'goodbye' command.

send_hello()
Send BCP 'hello' command.

stop()
Stop and shut down the socket client.

class mpf.core.bcp.bcp_socket_client.**MpfJSONEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Encoder which by default encodes to string.

Constructor for JSONEncoder, with sensible defaults.

If skipkeys is false, then it is a TypeError to attempt encoding of keys that are not str, int, float or None. If skipkeys is True, such items are simply skipped.

If ensure_ascii is true, the output is guaranteed to be str objects with all incoming non-ASCII characters escaped. If ensure_ascii is false, the output can contain non-ASCII characters.

If check_circular is true, then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an OverflowError). Otherwise, no such check takes place.

If allow_nan is true, then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is true, then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If indent is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. None is the most compact representation.

If specified, separators should be an (item_separator, key_separator) tuple. The default is (', ', ': ') if *indent* is `None` and (',', ': ') otherwise. To get the most compact JSON representation, you should specify (',', ':') to eliminate whitespace.

If specified, default is a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`.

default(*o*)
Encode to string.

encode(*o*)
Return a JSON string representation of a Python data structure.

```
>>> from json.encoder import JSONEncoder
>>> JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode(*o*, *_one_shot=False*)
Encode the given object and yield each string representation as available.

For example:

```
for chunk in JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

mpf.core.bcp.bcp_socket_client.**decode_command_string**(*bcp_string*)
Decode 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.core.bcp.bcp_socket_client.**encode_command_string**(*bcp_command*, ***kwargs*)
Encode 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.core.bcp.bcp_transport

Classes which manage BCP transports.

class mpf.core.bcp.bcp_transport.**BcpTransportManager**(*machine*)
Manages BCP transports.

Initialise BCP transport manager.

add_handler_to_transport(*handler*, *transport: mpf.core.bcp.bcp_client.BaseBcpClient*)
Register client as handler.

get_named_client(*client_name*) → mpf.core.bcp.bcp_client.BaseBcpClient
Get a client by name.

get_transports_for_handler(*handler*)
Get clients which registered for a certain handler.

register_transport(*transport*)
Register a client.

remove_transport_from_handle(*handler*, *transport: mpf.core.bcp.bcp_client.BaseBcpClient*)
Remove client from a certain handler.

send_to_all_clients(*bcp_command*, ***kwargs*)
Send command to all bcp clients.

send_to_client(*client: mpf.core.bcp.bcp_client.BaseBcpClient*, *bcp_command*, ***kwargs*)
Send command to a specific bcp client.

send_to_clients(*clients*, *bcp_command*, ***kwargs*)
Send command to a list of clients.

send_to_clients_with_handler(*handler*, *bcp_command*, ***kwargs*)
Send command to clients which registered for a specific handler.

shutdown(***kwargs*)
Prepare the BCP clients for MPF shutdown.

unregister_transport(*transport: mpf.core.bcp.bcp_client.BaseBcpClient*)
Unregister client.

mpf.core.case_insensitive_dict

Case insensitive dict.

class mpf.core.case_insensitive_dict.**CaseInsensitiveDict**(**args*, ***kwargs*)
A dict which lowercases all keys.

Initialise case insensitve dict.

clear() → None. Remove all items from D.

copy() → a shallow copy of D

fromkeys()
Returns a new dict with keys from iterable and values equal to value.

get(*key*, **args*, ***kwargs*)
Return item for key.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

static lower(*key*)
Lowercase the key.

pop(*key*, **args*, ***kwargs*)
Retrieve and delete a value for a key.

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

setdefault(*key*, **args*, ***kwargs*)
Set defaults.

update(*e=None*, ***f*)
Update a value for a key.

values() → an object providing a view on D's values

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*, *calling_context*, *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.

get_list_config(*value*)
Parse config list.

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

mode_stop(*mode*)
Remove events for mode.

play(*settings*, *context*, *calling_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*, *calling_context*, *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_config_entry(*settings*, *name*)
Validate one entry of this player.

mpf.core.config_spec

Config spec for MPF.

mpf.core.data_manager

Contains the DataManager base class.

class mpf.core.data_manager.**DataManager**(*machine*, *name*)
Handles key value data loading and saving for the machine.

Initialise data manger.

The DataManager is responsible for reading and writing data to/from a file on disk.

Parameters

- **machine** – The main MachineController instance.
- **name** – A string name that represents what this DataManager instance is for. This name is used to lookup the configuration option in the machine config in the mpf:paths:<name> location. That's how you specify the file name this DataManager will use.

get_data(*section=None*)
Return the value of this DataManager's data.

Parameters **section** – Optional string name of a section (dictionary key) for the data you want returned. Default is None which returns the entire dictionary.

remove_key(*key*)
Remove key by name.

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

Parameters

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

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

Parameters

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

mpf.core.delays

Manages delays within a context.

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

Initialise delay manager.

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

Parameters

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

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

add_if_doesnt_exist(*ms*, *callback*, *name*, ***kwargs*)
Adds a delay only if a delay with that name doesn't exist already.

Parameters

- **ms** – Int of the number of milliseconds you want this delay to be for. Note that the resolution of this time is based on your machine's tick rate. The callback will be called on the first machine tick *after* the delay time has expired. For example, if you have a machine tick rate of 30Hz, that's 33.33ms per tick. So if you set a delay for 40ms, the actual delay will be 66.66ms since that's the next tick time after the delay ends.
- **callback** – The method that is called when this delay ends.
- **name** – String name of this delay. This name is arbitrary and only used to identify the delay later if you want to remove or change it.
- ****kwargs** – Any other (optional) kwarg pairs you pass will be passed along as kwargs to the callback method.

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

check(*delay*)
Check to see if a delay exists.

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

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

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

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

I.e. prevents the callback from being fired and cancels the delay.

Parameters **name** – String name of the delay you want to remove. If there is no delay with this name, that's ok. Nothing happens.

reset(*ms*, *callback*, *name*, ***kwargs*)
Reset a delay, first deleting the old one (if it exists) and then adding new delay with the new settings.

Parameters **as add()** (*same*) –

class mpf.core.delays.**DelayManagerRegistry**(*machine*)
Keeps references to all DelayManager instances.

Initialise delay registry.

add_delay_manager(*delay_manager*)
Add a delay manager to the list.

mpf.core.events

Classes for the EventManager and QueuedEvents.

class mpf.core.events.**EventHandlerKey**(*key*, *event*)
Create new instance of EventHandlerKey(key, event)

count(*value*) → integer – return number of occurrences of value

event
Alias for field number 1

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

key
Alias for field number 0

class mpf.core.events.**EventManager**(*machine*)
Handles all the events and manages the handlers in MPF.

Initialise EventManager.

add_handler(*event*, *handler*, *priority=1*, ***kwargs*)
Register an event handler to respond to an event.

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

Parameters

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

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

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

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

does_event_exist(*event_name*)
Check to see if any handlers are registered for the event name that is passed.

Parameters **event_name** – The string name of the event you want to check. This string will be converted to lowercase.

Returns True or False

post(*event*, *callback=None*, ***kwargs*)
Post an event which causes all the registered handlers to be called.

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

You can control the order the handlers will be called by optionally specifying a priority when the handlers were registed. (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*)

Post an boolean event which causes all the registered handlers to be called one-by-one.

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

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

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

Parameters

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

post_queue(*event*, *callback*, ***kwargs*)

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

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

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

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

Parameters

- **event** – A string name of the event you're posting. Note that you can post whatever event you want. You don't have to set up anything ahead of time, and if no handlers are registered for the event you post, so be it. Note that this event name will be converted to lowercase.
- **callback** – The method which will be called when the final handler is done processing this event and any handlers that registered waits have cleared their waits.

- ****kwargs** – One or more options keyword/value pairs that will be passed to each handler. (Just make sure your handlers are expecting them. You can add **kwargs to your handler methods if certain ones don't need them.)

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

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

Parameters

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

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

You can control the order the handlers will be called by optionally specifying a priority when the handlers were registed. (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, stanard 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_queue()
Check if there are any other events that need to be processed, and then process them.

remove_handler(*method*)
Remove an event handler from all events a method is registered to handle.

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

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

Parameters

- **event** – The name of the event you want to remove the handler from. This string will be converted to lowercase.
- **handler** – The handler method you want to remove.

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

remove_handler_by_key(*key: mpf.core.events.EventHandlerKey*)
Remove a registered event handler by key.

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

remove_handlers_by_keys(*key_list*)
Remove multiple event handlers based on a passed list of keys.

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

replace_handler(*event*, *handler*, *priority=1*, ***kwargs*)
Check to see if a handler (optionally with kwargs) is registered for an event and replaces it if so.

Parameters

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

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

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

wait_for_any_event(*event_names: [<class 'str'>]*)
Wait for any event from event_names.

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

class mpf.core.events.**PostedEvent**(*event*, *type*, *callback*, *kwargs*)
Create new instance of PostedEvent(event, type, callback, kwargs)

callback
Alias for field number 2

count(*value*) → integer – return number of occurrences of value

event
Alias for field number 0

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

kwargs
Alias for field number 3

type
Alias for field number 1

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

Initialise QueueEvent.

clear()
Clear a wait.

is_empty()
Return true if unlocked.

wait()
Register a wait for this QueueEvent.

class mpf.core.events.**RegisteredHandler**(*callback*, *priority*, *kwargs*, *key*, *condition*)
Create new instance of RegisteredHandler(callback, priority, kwargs, key, condition)

callback
Alias for field number 0

condition
Alias for field number 4

count (*value*) → integer – return number of occurrences of value

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

key
Alias for field number 3

kwargs
Alias for field number 2

priority
Alias for field number 1

mpf.core.file_manager

Contains the FileManager and FileInterface base classes.

class mpf.core.file_manager.**FileInterface**
Interface for config files.

Initialise file manager.

find_file (*filename*)
Test whether the passed file is valid.

If the file does not have an externsion, this method will test for files with that base name with all the extensions it can read.

Parameters **filename** – Full absolute path of a file to check, with or without an extension.

Returns False if a file is not found. Tuple of (full file with path, extension) if a file is found

static get_config_file_version (*filename*)
Get the config version number from a file.

Since this technique varies depending on the file type, it needs to be implemented in the child class

Parameters **filename** – The file with path to check.

Returns An int of the config file version

load (*filename*, *verify_version=True*, *halt_on_error=True*, *round_trip=False*)
Load file.

save (*filename*, *data*, ***kwargs*)
Save file.

class mpf.core.file_manager.**FileManager**
Manages file interfaces.

static get_file_interface (*filename*)
Return a file interface.

classmethod init ()
Initialise file manager.

static load (*filename*, *verify_version=False*, *halt_on_error=False*, *round_trip=False*)
Load a file by name.

static **locate_file**(*filename*) → str
Find a file location.

Parameters **filename** – Filename to locate

Returns: Location of file

static **save**(*filename*, *data*, ***kwargs*)
Save data to file.

mpf.core.mode_timer

Mode timers.

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.

Initialise mode timer.

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

Parameters

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

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

Parameters

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

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

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

Parameters

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

reset(***kwargs*)
Reset this timer based to the starting value that's already been configured.

Does not start or stop the timer.

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

restart(***kwargs*)
Restart the timer by resetting it and then starting it.

Essentially this is just a reset() then a start().

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

set_current_time(*timer_value*, ***kwargs*)
Set the current amount of time of this timer.

This value is expressed in "ticks" since the interval per tick can be something other than 1 second).

Parameters

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

set_tick_interval(*timer_value*, ***kwargs*)
Set the number of seconds between ticks for this timer.

This is an absolute setting. To apply a change to the current value, use the change_tick_interval() method.

Parameters

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

start(***kwargs*)
Start this timer based on the starting value that's already been configured.

Use 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*)
Stop the timer and posts the 'timer_<name>_stopped' event.

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

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

Parameters

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

timer_complete(***kwargs*)
Automatically called when this timer completes.

Posts the 'timer_<name>_complete' event. Can be manually called to mark this timer as complete.

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

mpf.core.mpf_controller

Base class for MPF controllers.

class mpf.core.mpf_controller.**MpfController**(*machine*)
Base class for MPF controllers.

Initialise controller.

Parameters **machine** (*mpf.core.machine.MachineController*) – the machine controller

Returns:

mpf.core.placeholder_manager

Templates and placeholders.

class mpf.core.placeholder_manager.**BaseTemplate**(*template*, *placeholder_manger*, *default_value*)
Base class for templates.

Initialise template.

evaluate(*parameters*, *fail_on_missing_params=False*)
Evaluate template.

class mpf.core.placeholder_manager.**BoolTemplate**(*template*, *placeholder_manger*, *default_value*)
Bool template.

Initialise template.

evaluate(*parameters*, *fail_on_missing_params=False*)
Evaluate template to bool.

class mpf.core.placeholder_manager.**FloatTemplate**(*template*, *placeholder_manger*, *default_value*)
Float template.

Initialise template.

evaluate(*parameters*, *fail_on_missing_params=False*)
Evaluate template to float.

class mpf.core.placeholder_manager.**IntTemplate**(*template*, *placeholder_manger*, *default_value*)
Float template.

Initialise template.

evaluate(*parameters*, *fail_on_missing_params=False*)
Evaluate template to float.

class mpf.core.placeholder_manager.**MachinePlaceholder**(*machine*)
Wraps the machine.

Initialise placeholder.

class mpf.core.placeholder_manager.**PlaceholderManager**(*machine*)
Manages templates and placeholders for MPF.

Initialise.

build_bool_template(*template_str*, *default_value=False*)
Build a bool template from a string.

build_float_template(*template_str*, *default_value=0.0*)
Build a float template from a string.

build_int_template(*template_str*, *default_value=0*)
Build a int template from a string.

evaluate_template(*template*, *parameters*)
Evaluate template.

get_global_parameters(*name*)
Return global params.

mpf.core.randomizer

Generic list randomizer.

class mpf.core.randomizer.**Randomizer**(*items*)
Generic list randomizer.

Initialise Randomizer.

get_current()
Return current item.

get_next()
Return next item.

loop
Return loop property.

static pick_weighted_random(*items*)
Pick a random item.

Parameters **items** – Items to select from

mpf.core.scriptlet

Contains the parent class for Scriptlets.

class mpf.core.scriptlet.**Scriptlet**(*machine*, *name*)
Baseclass for scriptlet which are simple scripts in a machine.

Initialise scriptlet.

on_load()
Automatically called when this Scriptlet loads.

It's the intention that the Scriptlet writer will overwrite this method in the Scriptlet.

mpf.core.service_controller

Controller for all service functionality.

Controller provides all service information and can perform service tasks. Displaying the information is performed by the service mode or other components.

class mpf.core.service_controller.**CoilMap**(*board*, *coil*)
Create new instance of CoilMap(board, coil)

board
Alias for field number 0

coil
Alias for field number 1

count(*value*) → integer – return number of occurrences of value

index(*value*[, *start*[, *stop*]])) → integer – return first index of value.
Raises ValueError if the value is not present.

class mpf.core.service_controller.**ServiceController**(*machine*)
Provides all service information and can perform service tasks.

Initialise service controller.

get_coil_map() → [<class 'mpf.core.service_controller.CoilMap'>]
Return a map of all coils in the machine.

get_switch_map()
Return a map of all switches in the machine.

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

start_service()
Start service mode.

stop_service()
Stop service mode.

mpf.core.shot_profile_manager

Contains the ShotProfileManager class.

class mpf.core.shot_profile_manager.**ShotProfileManager**(*machine*)
Controller for show profiles.

Initialise shot profile manager.

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

Parameters

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

- **kwargs** – unused

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

Called on mode start.

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

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

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

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

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

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

Parameters

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

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

Parameters

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

mpf.devices

Module which contains all devices in MPF.

Submodules

mpf.devices.ball_device

Ball device and ejectors.

Submodules

mpf.devices.ball_device.ball_device_ejector

Baseclass for ball device ejectors.

class mpf.devices.ball_device.ball_device_ejector.**BallDeviceEjector**(*ball_device*)
Ejector for a ball device. It has to implement at least one of eject_one_ball or eject_all_balls.

Initialise ejector.

ball_search(*phase*, *iteration*)
Search ball in device.

eject_all_balls()
Eject all balls.

eject_one_ball()
Eject one ball.

mpf.devices.ball_device.hold_coil_ejector

Hold coil ejector.

class mpf.devices.ball_device.hold_coil_ejector.**HoldCoilEjector**(*ball_device*)
Hold balls by enabling and releases by disabling a coil.

Initialise hold coil ejector.

ball_search(*phase*, *iteration*)
Run ball search.

eject_all_balls()
Eject all balls.

eject_one_ball()
Eject one ball by disabling hold coil.

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

mpf.devices.ball_device.pulse_coil_ejector

Standard pulse ejector.

class mpf.devices.ball_device.pulse_coil_ejector.**PulseCoilEjector**(*ball_device*)
Pulse a coil to eject one ball.

Initialise ejector.

ball_search(*phase*, *iteration*)
Run ball search.

eject_all_balls()
Cannot eject all balls.

eject_one_ball()
Pulse eject coil.

mpf.devices.score_reel_controller

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.

Initialise score reel controller.

active_scorereelgroup = None
Pointer to the active ScoreReelGroup for the current player.

game_starting(*queue*, *game*, ***kwargs*)
Reset 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()
Create 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.

mpf.file_interfaces

Contains config file interfaces.

mpf.migrator

Contains the migrator classes.

Submodules

mpf.migrator.config_version_3

Migrate to config version 3.

mpf.modes

Contains MPF default modes.

Submodules

mpf.modes.attract

Default attract mode.

Submodules

mpf.modes.attract.code

Code of the default attract mode.

mpf.modes.bonus

Default bonus mode.

Submodules

mpf.modes.bonus.code

Code of the default bonus mode.

mpf.modes.carousel

Default carousel mode.

Submodules

mpf.modes.carousel.code

Code of the default carousel mode.

mpf.modes.credits

Default credits mode.

Submodules

mpf.modes.credits.code

Code of the default credits mode.

mpf.modes.game

Default game mode.

Submodules

mpf.modes.game.code

Code of the default game mode.

mpf.modes.high_score

Default high score mode.

Submodules

mpf.modes.high_score.code

Code of the default high score mode.

mpf.modes.service

Default service mode.

Submodules

mpf.modes.service.code

Code of the default service mode.

mpf.modes.tilt

Default tilt mode.

Submodules

mpf.modes.tilt.code

Code of the default tilt mode.

mpf.platforms

Hardware platforms in MPF.

Submodules

mpf.platforms.base_serial_communicator

Base class for serial communicator.

class mpf.platforms.base_serial_communicator.**BaseSerialCommunicator**(*platform*, *port: str*, *baud: int*)

Basic Serial Communcator for platforms.

Initialise Serial Connection Hardware.

Parameters

- **platform** (*mpf.core.platform.BasePlatform*) – the platform
- **port** –
- **baud** –

readuntil(*separator*, *min_chars: int = 0*)

Read until separator.

Parameters

- **separator** – Read until this separator byte.
- **min_chars** – Minimum message length before separator

send(*msg*)

Send a message to the remote processor over the serial connection.

Parameters **msg** – Byes of the message you want to send.

stop()

Stop and shut down this serial connection.

mpf.platforms.fast

FAST hardware platform.

Submodules

mpf.platforms.fast.fast_defines

Defines for FAST WPC.

mpf.platforms.fast.fast_dmd

Fast DMD support.

class mpf.platforms.fast.fast_dmd.**FASTDMD**(*machine*, *sender*)
Object for a FAST DMD.

Initialise DMD.

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

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

mpf.platforms.fast.fast_io_board

Fast Io board.

class mpf.platforms.fast.fast_io_board.**FastIoBoard**(*node_id*, *model_string*, *firmware_version*, *switch_count*, *driver_count*)
A fast IO board on the NET processor.

Initialise FastIoBoard.

mpf.platforms.fast.fast_led

WS2812 LEDs on the fast controller.

class mpf.platforms.fast.fast_led.**FASTDirectLED**(*number*)
Represents a single RGB LED connected to the Fast hardware platform.

Initialise LED.

color(*color*)
Instantly set this LED to the color passed.

Parameters **color** – an RGBColor object

current_color
Return current color.

mpf.platforms.fast.fast_serial_communicator

Fast serial communicator.

class mpf.platforms.fast.fast_serial_communicator.**FastSerialCommunicator**(*platform*, *port*, *baud*)

Handles the serial communication to the FAST platform.

Initialise communicator.

Parameters

- **platform** (*mpf.platforms.fast.fast.HardwarePlatform*) – the fast hardware platform
- **port** – serial port
- **baud** – baud rate

query_fast_io_boards()

Query the NET processor to see if any FAST IO boards are connected.

If so, queries the IO boards to log them and make sure they're the proper firmware version.

readuntil(*separator*, *min_chars: int = 0*)

Read until separator.

Parameters

- **separator** – Read until this separator byte.
- **min_chars** – Minimum message length before separator

send(*msg*)

Send 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()

Stop and shut down this serial connection.

mpf.platforms.interfaces

Package with interfaces for hardware platform devices.

Submodules

mpf.platforms.interfaces.dmd_platform

Interface for monochrome and rgb platform devices.

class mpf.platforms.interfaces.dmd_platform.**DmdPlatformInterface**

Interface for monochrome DMDs in hardware platforms.

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

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

mpf.platforms.interfaces.driver_platform_interface

Interface for drivers.

class mpf.platforms.interfaces.driver_platform_interface.**DriverPlatformInterface**(*config*, *number*)
Interface for drivers in hardware platforms.

DriverPlatformInterface is an abstract base class that should be overridden for all driver interface classes on supported platforms. This class ensures the proper required methods are implemented to support driver operations in MPF.

Initialise driver.

disable(*coil*)
Disable the driver.

enable(*coil*)
Enable this driver, which means it's held "on" indefinitely until it's explicitly disabled.

get_board_name()
Return the name of the board of this driver.

pulse(*coil*, *milliseconds*)
Pulse a driver.

Pulse this driver for a pre-determined amount of time, after which this driver is turned off automatically. Note that on most platforms, pulse times are a max of 255ms. (Beyond that MPF will send separate enable() and disable() commands.

Parameters **milliseconds** – The number of ms to pulse this driver for. You should raise a ValueError if the value is out of range for your platform.

Returns A integer of the actual time this driver is going to be pulsed for. MPF uses this for timing in certain situations to make sure too many drivers aren't activated at once.

mpf.platforms.interfaces.gi_platform_interface

Interface for GIs.

class mpf.platforms.interfaces.gi_platform_interface.**GIPlatformInterface**
Interface for GIs in hardware platform.

GIPlatformInterface is an abstract base class that should be overridden for all GI interface classes on supported platforms. This class ensures the proper required methods are implemented to support GI operations in MPF.

off()
Turn off the GI instantly.

Returns None

on(*brightness=255*)
Set the GI to the specified brightness level.

Parameters **brightness** – Integer (0 to 255) that sets the brightness level of the GI

Returns None

mpf.platforms.interfaces.matrix_light_platform_interface

Interface for matrix lights.

class mpf.platforms.interfaces.matrix_light_platform_interface.**MatrixLightPlatformInterface**
Interface for matrix lights in hardware platforms.

MatrixLightPlatformInterface is an abstract base class that should be overridden for all matrix light interface classes on supported platforms. This class ensures the proper required methods are implemented to support matrix light operations in MPF.

off()
Turn off the matrix light instantly.

Returns None

on(*brightness=255*)
Set the matrix light to the specified brightness level.

Parameters **brightness** – Integer (0 to 255) that sets the brightness level of the light

Returns None

mpf.platforms.interfaces.rgb_led_platform_interface

Interface for RGB hardware devices/LEDs.

class mpf.platforms.interfaces.rgb_led_platform_interface.**RGBLEDPlatformInterface**
Interface for LEDs in hardware platforms.

LEDPlatformInterface is an abstract base class that should be overridden for all LED interface classes on supported platforms. This class ensures the proper required methods are implemented to support LED operations in MPF.

color(*color*)
Set the LED to the specified color.

Parameters **color** – a list of int colors. one for each channel.

Returns None

mpf.platforms.interfaces.servo_platform_interface

Platform interface for servos.

class mpf.platforms.interfaces.servo_platform_interface.**ServoPlatformInterface**
Interface for servos in hardware platforms.

ServoPlatformInterface is an abstract base class that should be overridden for all servo interface classes on supported platforms. This class ensures the proper required methods are implemented to support servo operations in MPF.

go_to_position(*position*)
Move servo to a certain position.

mpf.platforms.interfaces.switch_platform_interface

Interface for switches.

class mpf.platforms.interfaces.switch_platform_interface.**SwitchPlatformInterface**(*config*, *number*)

Interface for switches in hardware platforms.

SwitchPlatformInterface is an abstract base class that should be overridden for all switches interface classes on supported platforms. This class ensures the proper required methods are implemented to support switch operations in MPF.

Initialise default attributes for switches.

mpf.platforms.opp

Open Pinball Project (OPP) hardware platform.

Submodules

mpf.platforms.opp.opp_incand

Support for incandescent wings in OPP.

class mpf.platforms.opp.opp_incand.**OPPIncand**(*incand_card*, *number*)

A driver of an incandescent wing card.

Initialise Incandescent wing card driver.

off()

Disable (turns off) this light.

on(*brightness: int = 255*)

Enable (turns on) this driver.

Parameters **brightness** – brightness 0 (off) to 255 (on) for this incandescent light. OPP only supports on (>0) or off.

class mpf.platforms.opp.opp_incand.**OPPIncandCard**(*chain_serial*, *addr*, *mask*, *incand_dict*)

An incandescent wing card.

Initialise OPP incandescent card.

mpf.platforms.opp.opp_neopixel

OPP WS2812 wing.

class mpf.platforms.opp.opp_neopixel.**OPPNeopixel**(*number*, *neo_card*)

One WS2812 LED.

Initialise LED.

color(*color*)
Instantly set this LED to the color passed.

Parameters

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

class mpf.platforms.opp.opp_neopixel.**OPPNeopixelCard**(*chain_serial*, *addr*, *neo_card_dict*, *platform*)
OPP Neopixel/WS2812 card.

Initialise OPP Neopixel/WS2812 card.

add_neopixel(*number*, *neo_dict*)
Add a LED channel.

mpf.platforms.opp.opp_rs232_intf

Defines for OPP platform.

class mpf.platforms.opp.opp_rs232_intf.**OppRs232Intf**
Constants for OPP serial protocol.

static calc_crc8_part_msg(*msg_chars*, *start_index*, *num_chars*)
Calculate CRC for part of a message.

static calc_crc8_whole_msg(*msg_chars*)
Calculate CRC for message.

mpf.platforms.opp.opp_switch

OPP input card.

class mpf.platforms.opp.opp_switch.**OPPInputCard**(*chain_serial*, *addr*, *mask*, *inp_dict*, *inp_addr_dict*)
OPP input card.

Initialise OPP input card.

class mpf.platforms.opp.opp_switch.**OPPSwitch**(*card*, *number*)
An OPP input on an OPP input card.

Initialise input.

mpf.plugins

Contains all 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.

class mpf.plugins.osc.**OSC**(*machine*)
OSC plugin.

Initialise OSC plugin.

client_send_osc_message(*category*, *name*, *data*)
Send 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.

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

process_config(*event*, *data*)
Send config data to the OSC client.

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

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

process_light(*light*, *data*)
Process a light event received from the OSC client.

process_message(*addr*, *tags*, *data*, *client_address*)
Receive OSC messages and act on them.

process_refresh(*name*, *data*)
Process refresh.

process_switch(*switch*, *data*)
Process a switch event received from the OSC client.

process_sync(*name*, *data*)
Process sync.

process_wpcsync(*name*, *data*)
Process wpc sync.

register_lights()
Add handlers to all lights so the OSC client can receive updates.

register_switches()
Add switch handlers to all switches so the OSC client can receive updates.

start(***kwargs*)
Start the OSC server.

stop()
Stop the OSC server.

mpf.plugins.osc.**plugin_class**
alias of *OSC*

mpf.tests

Unit tests for the MPF framework.

Submodules

mpf.tests.TestDataManager

In-memory DataManager.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

D

E

F

G

H

I

K

L

M

O

P