Overview

CHAPTER 1

MPF Overview

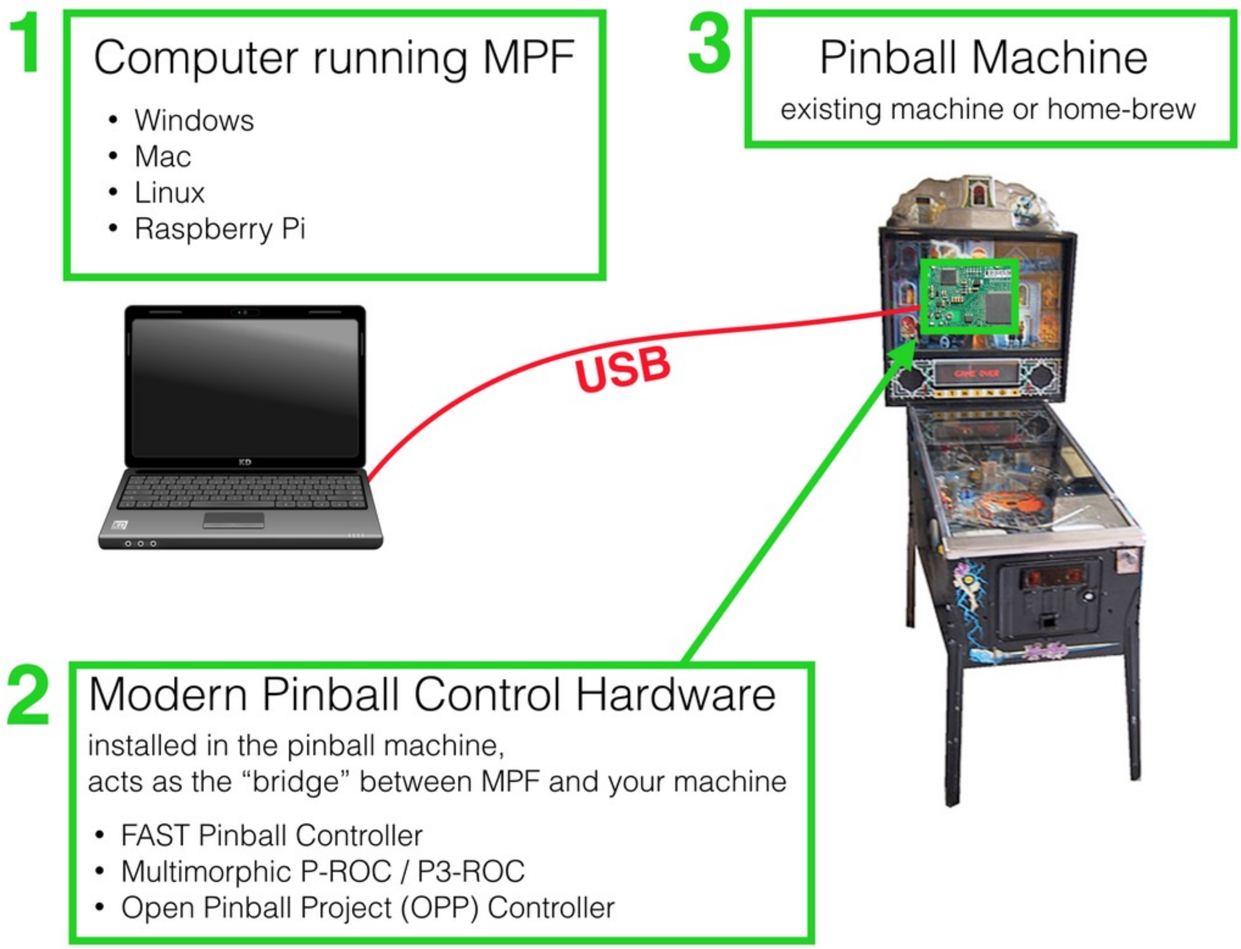
The Mission Pinball Framework (which we call "MPF") is free and open source software that you run on a computer (Windows, Mac, Linux, Raspberry Pi, etc.) to control a real, physical pinball machine. (More info one what MPF is here.)

Most people develop their game on their laptop, and then when they're done, transfer it to a smaller computer permanently installed in their pinball machine.

The computer running MPF is connected to a *modern pinball control system* via USB. (MPF supports several different control systems, including FAST Pinball, P-ROC, Open Pinball Project open source hardware, and Stern SPIKE hardware.)

You put that control system in your pinball machine, which can be a custom (home brew) machine or an existing machine you want to reprogram.

This diagram shows how it all fits together:



The MPF software is used to configure and control everything in your machine, including:

- Pinball mechanisms (switches, LEDs, lights, motors, coils, servos, steppers, flippers, ball locks, diverters, etc.)
- Pinball logic (ball locks, multiball, modes, tilt, high scores, ball saves, ball search, extra balls, etc.)
- The display (or displays): DMD, RGB LED, and/or LCD
- Audio & sounds
- Coordinated "shows" of actions which flash lights, fade LEDs, play sounds and video, etc.
- Player management, including player progress, scoring, tracking towards goals, etc.
- Plus lots of other little things that you probably aren't even thinking about yet :)

Note: MPF is a work-in-progress!

At this point MPF is a work-in-progress and not yet complete. It's being built by pinball-loving software developers in their spare time. There's a lot you can do with MPF today, but we also have a lot of work still to do. We're working hard though, typically adding 20-30 updates per week! And MPF is definitely "done" enough for you to use it today.

Read on to understand other important concepts about MPF:

MPF complete feature list

Even though MPF is a work-in-progress that's not yet complete, the core dev team has been working on it since 2014, with thousands of hours of combined effort.

Major Features & Concepts

- The vast majority of "programming" your game can be done with text-based config files that make it easy to get powerful and complex pinball features running in your game. They're also easy for non-programmers to use.
- MPF is "event-driven" meaning that everything that happens in a pinball machine generates an event, and you can use those events to trigger actions (scoring, lights, starting a mode, etc.)
- Advanced programmers and customization can be done via the API. (The API is fully documented at developer.missionpinball.org.)
- You can easily switch between hardware platforms, so if sometime down the road you want to switch hardware or the company whose hardware you're using goes out of business, all your effort is not lost as you can easily move everything to a new hardware platform with a few changed lines in your config file.

Compatible control systems / electronics

MPF currently interfaces with the following pinball control systems & electronics (which in turn control the physical pinball machine hardware):

- Multimorphic P-ROC & P3-ROC pinball controllers, with either PD-8x8, PD-16, PD-LED, and SW-16 driver and accessory boards or installation in existing WPC, Stern Whitestar, or Stern SAM machines.
- FAST Pinball Core, Nano & WPC controllers, with 3802, 1616, and 0804 I/O boards, FAST servo boards, or installation in existing WPC machines.
- Open Pinball Project (OPP) open source controllers with Gen2 driver boards.
- Stern SPIKE / SPIKE 2 pinball machines.
- LISY controllers for Gottlieb System 1 and System 80 machines.
- Mark Sunnucks's "Snux" System 11 driver board for use in System 11 and Data East machines, in concert with either a P-ROC or FAST WPC controller.
- Fadecandy RGB LED controllers.
- Open Pixel Control (OPC) LED and lighting controllers.
- I2C servo controllers.
- Pololu Maestro servo controllers.
- SmartMatrix RGB LED DMD controllers.
- RGB.DMD RGB LED-based DMD controllers.

- MyPinballs segment display controllers.
- Trinamics Steprocker stepper motor controllers.

See the *Control Systems / Electronics* documentation for full details.

Pinball mechanism support

MPF currently supports the following different types of pinball playfield mechanisms:

- Switches (normally open, normally closed, mechanical or opto, with configurable debounce settings)
- Coils / drivers / solenoids (pulse, enable, disable, PWM)
- Lamp matrix-based incandescent lights & LEDs
- LEDs (RGB, GRB, RGBA, RGBW, RGBAW)
- Accelerometers
- GI (general illumination)
- Flashers
- Flippers
- Pop bumpers / slingshots
- Drop targets and drop target banks
- Diverters
- All forms of troughs (modern, System 11, early WPC, early ’80s, Gottlieb System 3, etc.)
- Ball devices (scoops, VUKs, saucers, locks, etc.)
- Multiple playfields and playfield transfers (including head-to-head machines)
- Driver-enabled devices (like flippers and pop bumpers in System 11 machines)
- Mechanical and coil-fired plungers, ball launchers, and catapults
- EM score reels
- Kickbacks
- Magnets
- Rollover switches
- Servos
- Stepper motors
- Traditional motors

See the *Pinball Mechs* documentation for full details.

Game logic

MPF includes built-in support for all the pinball machine and game logic you need, inculding:

- Modes and a mode stack (start / stop / restart / stacked modes)

- Ball locks
- Multiball
- Ball saves
- Ball search
- Extra balls
- Tilt
- Credits / coin play
- Audits
- Bonus
- High score
- Full per-player variable and settings support. Save/restore anything on a per-player bases (shots, objectives, goals collected, targets hit, etc.)
- Player achievements & achievement groups (groups of modes to start which progress towards wizard mode, etc.)
- Ball tracking / automatic ball routing
- Shots & shot groups (with full per-player state management (e.g. lit, unlit, flashing, etc.)
- Shot rotation (lane change, etc.)
- Attract mode
- Logic blocks, which let you build complex pinball game logic out of reusable components via the config files
- Score controller to assign points (or other progress) per-player for different events, with mode integration for blocking and blending
- Timers (start / stop / pause / count down / count up)
- Video modes
- Switch combinations (flipper cancel, hold flipper button to start super skill shot, etc.)
- Timed switches (hold the flipper for 2 seconds to show game stats, etc.)

See the *Game Logic* documentation for full details.

Displays, DMDs, & Graphics

- On-screen LCD displays, either high-def or with a "dot" look
- Physical mono-color DMDs
- RGB LED DMDs
- Segmented displays
- Display "slides" with priorities, transitions in and out
- Display "widgets" (things you put on displays), including:
  - Text (with fonts, styles, colors, dynamic text based on game state, etc.)

    - Images & animated images
    - Videos
    - Shapes
    - "Picture-in-picture" style sub-displays
- Any property of any widget can be animated (opacity, size, position, etc.)

See the *Displays* documentation for full details.

Sounds & Audio

- Multi-track sound system with automatic volume and ducking (e.g. voice, sfx, and background music tracks)
- Per-track settings for simultaneous sounds and sound queues (e.g. let as many sfx sounds play at once as you want, but queue sounds on the voice track so only one plays at a time)
- Advanced per-sound "tuning", including attack, attenuation, ducking, etc.
- Sound pools and sound groups, so you can have multiple sounds for a single effect and cycle through them, with controls for whether they random, weighed random, rotation patterns, etc.

See the *Sounds* documentation for full details.

Shows

- A show controller which runs coordinated shows of LEDs, lights, coils, flashers, sounds, slides, videos, animations, etc.
- Start/stop/pause/resume shows
- Dynamic shows which change based on what's happening in the game.
- Change the playback speed of shows (even while they're playing)

See the *Shows* documentation for full details.

Machine Management

- Service mode / operator menus
- Operator-configurable "settings" which you can use to expose any setting anywhere in MPF to game operators.
- A data manager which handles reading and writing data from disk, including audits, earnings, machine variables, high scores, etc.
- Power supply management (map drivers to power supplies to make sure not too many things fire at once)

Tools

- The *MPF Monitor* standalone app which is a graphical tool that connects to a live running instance of MPF and shows the status of various devices. You can interact with it by clicking on switches and see your game in action on your computer.
- An "interactive" media controller which lets you interactively build and test display slides, widgets, and animations.
- A switch player which lets you build automatically scripts to "replay" switches for testing your game.
- A complete set of test functions which you can use to write your own automated tests for your machine.
- A keyboard interface which lets you simulate switch actions with your computer keyboard. (Great for testing!)
- Detailed logging, config file checking, and helpful error messages to help you troubleshoot issues.

Professional-level features

MPF contains hundreds of the "little" things most people never think about that help ensure machines running it are truly professional-level machines that can be placed in revenue service in public locations. Here are just a few random things that have caused people to say, "Hey, that's cool!" over the years:

- Power supply management: MPF knows how much current each power supply has and how much current various devices require, so it will intelligently manage and delay coil firings to ensure fuses don't blow. (For example, don't reset the drop targets at the same time the flippers are held on and a ball is being ejected.)
- Tilt-through prevention: A sliding time window ensures that the tilt plumb-bob has settled before the next player's ball is started.
- Automatic ball routing and retry logic:
- Asset pools: Sound effects, images, and videos can be "pooled" (with various settings for randomness, weightings, etc.), ensuring that each "hit" of a target produces a different sound instead of the same one over and over.
- Audio loops and break / resume points: Cue points for music and audio to ensure that music tracks are smoothly looped and advanced based on game play.
- Advanced multi-track audio: Automatic ducking of music and sfx when voice tracks play, etc.
- Auto leveling based on accelerometer: The machine knows when it's out of level and can post a credit dot or notify the operator.

Developer-friendly

- Fully open-source and well-documented code.
- A plugin architecture which allows you to write your own plugins to extend baseline functionality.
- Modular design that lets you write your own hardware interfaces.

- A "scriptlet" interface which can be used to easily add Python code snippets to a game to extend the functionality you can get with the configuration files.
- A mode "code" interface which lets you add custom Python code to game modes.

And the best part: Everything mentioned on this page (except for the developer stuff) can be done via the text-based configuration files. If you don't want to be a "coder," you don't have to be. (Though if you are a coder, we'd love to have you help us write MPF!

By the way, if you'd like to see what we have in store for the future, check out our *MPF Road Map, Vision & Future*.

The MPF "Media Controller"

All modern pinball machines use graphics and sound. MPF's architecture is build so that the core "game" engine is completely separate from the "media" engine.

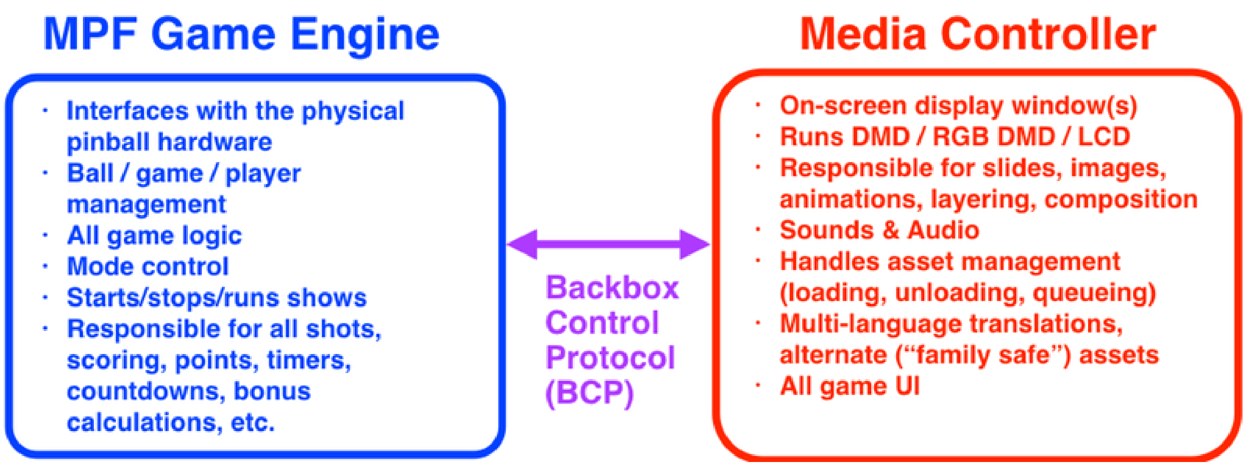
The "game" engine is the MPF software itself, and the "media" engine is something called the MPF Media Controller (which we often abbreviate as "MPF-MC").

When you run MPF, these two components are two separate processes that talk to each other via something called the "Backbox Control Protocol".

The details and inner workings of this are not really important, (and frankly they're mostly hidden from you).

But as you start to learn about MPF, just keep in mind that the part of MPF that runs your game and controls the hardware is separate from the part that shows the graphics and plays the sounds.

Here's a diagram that shows what each piece does:



More details about MPF's media controller architecture, as well as guides which show you how to run them on separate computers, or even to replace MPF's Media Controller with one based on Unity 3D or something you write yourself, are available in the *Displays, DMDs, & Graphics* section of the documentation.

Understanding MPF config files

MPF uses text-based config files to control the bulk of your game logic. In a sense, your MPF "code" is actually these config files.

There are machine-wide config files which control machine-wide things (such as hardware mappings, switches, lights, etc.) as well as mode-specific config files that control what happens when a specific mode is running. (And you can stack modes so you have a lot of them all doing different things at once.)

MPF also uses text-based files to control the "shows" which are the coordinated sequences of lights, sounds, displays, etc.

The MPF config files use a file format called YAML which is text-based and human readable. You can edit them in Notepad. YAML is kind of like XML, though easier to read and write. It's kind of like INI files, though more powerful.

We have a *detailed config file reference* that explains all the options for all the files, but for now we just want to explain the basic concept of how these files work. (Feel free to browse through the config file reference, but remember that it's a just a reference. You'll actually learn how to use the config files via our tutorial and How To guides. Learning MPF by reading the config file reference is like learning a foreign language by reading a dictionary. :)

When you create your machine code in MPF, you'll actually create a folder which will contain your config files. A super-simple snippet might look like this:

```
game:
  balls_per_game: 3
```

Want a 5-ball game instead? Simple! Just change it:

```
game:
  balls_per_game: 5
```

Ultimately your config files will be thousands of lines long (though you can break them up into multiple files to help your sanity), but again, don't be overwhelmed now. The tutorial will walk you through them step-by-step, and in no time you'll have a playing pinball machine!

Config files versus "real" programming

When we talk about MPF, we really play up the fact that when you use MPF, you can do 90%+ of your of your "programming" with MPF's *YAML configuration files*.

We've received criticism of that over the past few years, typically falling into one of the following categories:

- Since everything in MPF is in config files, that's something new you have to learn. If you don't know MPF, you can't just look at a config file and know what's happening.
- Since config files insulate the game programmer from the code, when something doesn't work, you don't know if it's your config or a bug in MPF.
- Using config files limits game programmers in that they have to do everything the "MPF way."
- Coding is fun! MPF deprives people of that.

We understand the motivation behind all these thoughts, so we'd like to provide our perspective on these issues.

You can still code in MPF

MPF does not prevent you from coding. We provide two levels of abstraction to programmers: hardware abstraction and device abstract. If you use a flipper device in code it will expose methods to disable or enable a flipper and work on any hardware which is supported in MPF. Plus the device will manage all the game integration (e.g. disable flipper after the game).

Nevertheless, you might want to implement a different type of flipper (say with three coils each) and the flipper device might be a bad fit. Therefore, you can use the hardware abstraction interface and write rules in a hardware independent way (or overload the flipper device which does exactly that).

If you want to use a very specific feature of your hardware and we did not implement an abstraction for it you can also access the hardware directly but it will likely not work on other platforms anymore. E.g. this might be the case if you want to do advanced stuff with the AUX port on the P-Roc.

As you see, MPF offers you all kind of flexibility. You can access hardware directly or use abstractions. Plus, if you implement your own devices or extend existing devices (those can live inside your machine folder) you will be able to instantiate them using config (if you want that).

Code can be added either globally (using scriptlets or code hooks), per mode, as new/overloaded device or even as a custom platform. See the MPF developer documentation for more details about our APIs and interfaces.

Why config files?

At the most basic level, config files in MPF let you access hundreds or thousands of lines of code with a simple line or two in a config. The actual code that runs a pinball machine is really, really complex, especially when you think about all the logic around ball tracking, mode stacking, multiple things happening at once, etc.

By providing an interface like the config files, we allow you to have access and to be able to control all these complex things in a simple way.

MPF's config files are a form of something in computer science called a "domain-specific language. (DSL)"

In this context the "domain" is pinball, so the MPF config files could be thought of as a "pinball-specific language". This means that you can't use the MPF DSL to program a dart board machine or a self-driving car, but when it comes to programming pinball, they're darn good!

There are many advantages to DSLs, including:

- Increased productivity: Get a complex mode up and running in MPF with a half-page config file instead of writing 500 lines of Python code.
- Fewer bugs: The config files are used by lots of people, so we know they work the way they're supposed to, instead of every pinball maker writing their own stuff from scratch and re-solving the same problems over and over.
- Easier to read: You can look at a few lines of config file and know what you're looking at and what it's trying to do versus pages of Python code that you have to reverse engineer to understand.

- Ease of support: Same as above. If you are having a problem, it's easy to post a config to the forum and everyone can understand it, versus scanning through hundreds of lines of custom Python code.
- Ease of planning: Since everyone in the MPF community speaks the same language of config files, it's easy to ask for help and direction on how to do things.
- Insulation from future updates: The config files remain constant (or we provide migration tools to upgrade them, so we can make major changes to MPF under the hood without you having to re-write anything in your game.

Config files in MPF: use as much (or as little) as you want

Even though we just laid out the reasons we like "programming" your game via config files instead of "real" code, there's one important thing to know about the config files:

You don't have to use config files for everything.

There's a whole website dedicated to mixing custom code with MPF (at developer.missionpinball.org, and you can easily mix code (written in Python or the language of your choice) with existing MPF code and configs, so really you can use as much or as little of the config file interface as you want.

One way to think about MPF is that it's a solid set of pinball functionality with a nice API, and then the config file interface is a separate component that rides on top of that API and exposes it via easy-to-use config files.

So if you're a programmer and prefer to program against the API directly, go for it! The API is well-documented and fairly stable now, so if you don't want to use a single config file for anything, you can just use the MPF API and do whatever you want and still benefit from the thousands of hours of effort we put into MPF.

The reality, though, is that building a complete game in MPF is a balance between doing things in config files and writing code. At the end of the day, it doesn't matter whether your game is 90% configs and 10% code, or 80/20, 50/50, 20/80, etc. The exact balance depends on the personal preference of the person building the game.

In fact even we drop into "real" code to do certain things. There have been lots of times when we think, "Yeah, X action would be 20 confusing config lines or just two lines of Python, so I'm writing it in Python." That's perfectly fine.

The real power comes when you start to mix-and-match. For example, you could use the MPF config files to build out your base hardware interface and mode structures, then use your own Python code to do the logic within a mode, then use your mode code to post an event to use MPF's scoring system, etc.

If you don't use MPF, then you have to write everything yourself in code. If you do use MPF, then you get to choose what you write in code and what you don't have to write. (Seriously, ball tracking is a hard. Use our pre-written code via the config files!)

I already know Python. Why learn obscure config files?

Again, the software that runs pinball machines is complex. The complete MPF codebase is over 15,000 lines of code, with thousands of lines of code to do things that *seem* simple on the surface, like managing ball devices and tracking where all the balls are at all times.

MPF's config files provide a friendly interface to all that complexity. So yes, it's true that you have to spend a few hours learning about the `ball_devices:` section of the MPF config files in order to learn how to use them effectively. But the alternative is learning everything about how ball tracking works in a pinball machine and then writing all that from scratch yourself. That would take a lot longer than it would to learn about how to configure ball tracking in MPF. And besides, we already did that! :)

Aren't config files limiting?

Even though we've tried to envision many different scenarios and many different types of pinball machines as we built MPF, it's true that MPF does things a certain way, and the config files are a manifestation of the way MPF does things. So there could be scenarios where you want to do something differently than how MPF does it.

But this does not mean that MPF is not the right framework for you. Don't throw the baby out with the bath water! If you don't like the way something works in MPF's shot management tracking, you don't have to completely write your own shot management from scratch. Rather you can use MPF's shot sytem, subclass the methods and objects you want to change, and then tweak them to work in your specific scenario.

Even if you want to completely replace one component of MPF, there hundreds of different components, modules, and systems that go into a pinball machine that are already part of MPF. Unless you want to write all of those from scratch, using MPF lets you get a head start on many of the things that you need in your machine that you don't want to write yourself.

Coding is fun! Doesn't using config files deprive me of that?

Some people have said, "I like to code. I don't *want* to just build my machine quickly." Certainly we appreciate that, because we like to code too!

If you decide to write the software for your own pinball machine from scratch, you will spend hundreds of hours writing low-level pinball things, like hardware device management, ball tracking, a mode queue, player objects, a display and sound system, etc.

If you use MPF, even if you write your own game logic in Python code, then you can focus on the fun stuff while the MPF developers focus on the boring low-level pinball stuff.

Of course, if you're thinking, "But I *like* the low-level stuff, I want to write that," then we would love to have you on our team helping to make MPF better. :) We have a to-do list for MPF which will take years to complete, so if you like to code, we'd love to have you help!

If there's something that MPF does that you don't like and that you think you can do better, that's an even better reason to contribute back to MPF. Please, help us make MPF better!

We have success stories of this already. Brian Madden and Gabe Knuth started writing MPF in 2014. Since then, MPF user Jan Kantert started using MPF, and then he started tweaking things here and there (and submitting his changes back to the MPF project.) Now Jan has completely rewritten MPF's ball device code, our hardware platform interface, he's added multiball, ball lock, and ball search, extra balls, servos, tests. . . the list goes on.

Another MPF user, Quinn Capen, has rewritten MPF's RGB LED interface, written a complete pinball-focused advanced audio system, written an alternative media controller based on Unity 3D. . .

John Marsh said, "It would be cool if there was a GUI wizard to help people set up their machines," so now he's building that.

Hugh Spahr created his own pinball controller hardware (the Open Pinball Project), and then wrote a platform interface for MPF so MPF users can use OPP hardware too.

You get the idea.

The bottom line is that these are all MPF users who love to code, so rather than being scared away by MPF's config file interface, instead they embraced MPF, dug in, and are making MPF better. So now all the time they spend writing code isn't just limited to running on their machine which sits in their basement for 360 days a year; instead their code is running on pinball machines all over the world, which is very fulfilling and cool!

When something breaks, I don't know if it's my config or an MPF bug?

True, one of the limitations of using config files is that when things don't work the way you expect, you don't know if it's a problem with your config or a deeper bug in MPF.

However if you're someone who knows how to program, MPF is open source! You can go through the MPF code to see if it's a bug, and if so, you can fix it and submit a pull request to fix that bug for everyone.

And if it's a configuration error, you can also edit the MPF documentation to be more clear, and then submit a pull request to the docs, and now you've also helped fix this issue for everyone.

Again, don't not use MPF because it uses config files and you want to "know" what's happening under the hood. Instead learn MPF and the code behind it and share your programming and pinball passion with the world!

Using MPF means you have a team of programmers making your machine better

The MPF project was started in May 2014. Since then we have over 5,000 hours of time spent (both in code and documentation). More importantly, we're continuing to update and expand MPF, with dozens of commits to the core code and docs every week. (Probably an average of 60 hours a week of work.)

If you use MPF, you get all that work for free. :) It's like having a team of developers working 60 hours a week to make your game better. Pretty cool!

The bottom line

The creators of MPF are passionate about pinball, passionate about software development, and passionate about open source.

The beauty of MPF is that it's a bunch of people, from all over the world, writing software and documentation which helps more people create more pinball machines. As MPF grows in popularity, we love the fact that some day we will be able to walk into a bar, see a pinball machine, and know that some of the code we wrote is powering that machine. It warms our hearts.

If you decide to go your own way and not use MPF, that's great. We support you! (Feel free to rip off any ideas from MPF. We'd love it!) But don't write off MPF just because you want to do "real" programming and MPF is a "config-based" project. We could use the help of programmers like you. :)

CHAPTER 2

Compatible Pinball Machines

If you haven't done so already, be sure to read the *MPF Overview* page to understand how MPF talks to physical pinball machines.

There are three options when it comes to using MPF with a pinball machine:

- Build your own new machine completely from scratch.
- Rewrite the rules for an existing machine, which means you don't change the physical hardware at all, rather, you just update the software.
- "Retheme" an existing machine, which means you reuse all of the mechanical and electrical components of an existing machine, but you strip down and replace all the artwork to transform it into something else. (And you rewrite all the rules for your new theme.)

Here are more details on each option. The "rewrite the rules" and "retheme" options above are combined below into the "controlling an existing machine" section:

Controlling a custom "home brew" machine with MPF

Details for how to build custom machine hardware are covered on the PinballMakers.com Wiki. We cover some general areas here and suggest that you investigate those on your own. Contributions to the guide (and the rest of the documentation are *welcome*).

Control System

If you are "just" retheaming a machine have a look at the *Controlling an existing machine with MPF* section. If you want to use MPF to power a new *custom* pinball machine that you build yourself, you should buy new custom driver boards. There are a few common choices:

- *Multimorphic P3-Roc*
- *FAST Pinball*

- *Open Pinball Project (OPP)*
- *LISY Home (custom pinball version of LISY)*
- *Arduino Pinball Controller*
- *CobraPin Pinball Controller*

P3-Roc and FAST are both commercial systems at a similar price point but features vary slightly so compare them wisely. OPP is an open source/open hardware project and much cheaper but expect to invest some more time into the hardware itself. CobraPin is based on OPP with the goal of making OPP more accessible and provides somewhat of an all-in-one solution.

You might also want to some more control boards for servos, steppers and light. Common choices are:

- *Fadecandy* for WS2812 lights (FAST and P3-Roc offer this too)
- *Pololu Maestro* for servos

See the *Hardware Section* for all hardware supported by MPF.

Power and Wiring

You should invest some time into at the beginning of your custom pinball journey into your power supply and wiring.

- *Voltages and Power*
- *Wiring and Connectors in Pinball Machines*

Parts and Assemblies

MPF supports a *varity of pinball mechs*. You can have a look at manuals of existing machines to find numbers of mechs. For homebrew machines it is wise to buy assemblies of mechs. Mostly, because mechs consist of a lot of parts and you will likely fail to order all of them at once. Additionally, assemblies are often cheaper.

There are a few shops such as Pinballlife which offer assemblies. They also have a homebrew section which is worth checking out. Other shops such as Marcos Specialities offer more parts but are less focused on homebrew.

Controlling an existing machine with MPF

If you want to use MPF to write your own custom game code for an *existing* Williams or Stern pinball machine, you replace the original CPU board in the machine with a modern pinball controller board (called a *hardware controller*) such as a *P-ROC* Controller (but not P3-Roc). That hardware controller interfaces with the existing machine's driver boards to control the coils, lights, and DMD, and it provides a "bridge" (via USB) to a host computer running Python and the Mission Pinball Framework.

Machine Type	P-ROC	LISY	APC	Direct
Williams / Bally / Midway WPC	X	X		
Williams / Bally System 11	X		X	
Data East	X			
Stern S.A.M.	X			
Stern Whitestar	X			
Pinball 2000	X			
Stern SPIKE / SPIKE 2				X
Gottlieb System 1		X		
Gottlieb System 80		X		
Bally/Stern w/ AS-2518-17 or AS-2518-35 MPU		X		

Notes:

- "WPC" includes WPC-S and WPC-95, and machines made under the brands of Williams, Bally, and Midway. (A complete WPC game list is here.)
- System 11 and Data East machines require the "*Snux*" replacement driver board in addition to the P-ROC or FAST controller.
- Since Stern SPIKE systems have a linux-based computer inside them already, so *MPF can directly connect to and control them via USB*. No additional hardware is needed.
- Gottlieb System 1 and 80 can be controlled using the *LISY platform*
- Bally and Stern Games manufactured from 1977 to 1985 with MPU AS-2518-17 or AS-2518-35 can be controlled using *LISY35*

If you want to use MPF with an existing machine type that's not on the list above, that's still possible, but you'd have to rewire the entire machine and use modern control hardware. In other words, you strip the guts and keep all the hardware, and the machine essentially becomes a home-brew machine on the inside and a retheme or update on the outside. However, there might be an alternative not listed here so we recommend you to ask in our *user forum*.

CHAPTER 3

Downloading & Installing MPF (2023 Version)

The MPF Installers were rewritten from scratch and completely updated for MPF 0.56 in August 2022. (If you're using an older version of MPF, see the docs for the version you're using for installation instructions.)

MPF should work with following platforms:

- Windows 10 / Windows 11 (64-bit only)
- macOS 10.14+, up through macOS 13 Ventura (Intel & Apple Silicon)
- Linux (64-bit, lots of distros)
- Raspberry Pi 4B (eventually, not done yet, but no one can get Pis now anyway so no hurry.)

Python 3.9 is the latest version of Python supported. Some platforms also support Python 3.7 and 3.8. Python 3.10+ will not work.

Here links to the installation guides for each platform:

Installing MPF on Mac

This is the new process used to install MPF 0.56 on a Mac.

Overview of MPF on macOS

MPF works on macOS running on both Intel and Apple Silicon (M1/M2 processors). Requirements are:

- Apple Silicon Mac (M1/M2 processors) require macOS 12 Monterey or newer and Python 3.9.
- Intel processors require MacOS 10.14 or newer, and Python 3.7 - 3.9.
- MPF does not work with Python 3.10+

To install MPF on a Mac:

1. If you do not have Python, install Python 3.9.13 from python.org. If you have an M1/M2 Mac, be sure to get the Universal installer, not the Intel one.

   macOS 64-bit Intel-only installer
   (https://www.python.org/ftp/python/3.9.13/python-3.9.13-macosx10.9.pkg)

   macOS 64-bit universal installer
   (https://www.python.org/ftp/python/3.9.13/python-3.9.13-macos11.pkg)

   Choose the default installation location. If you want to install to a custom location, remember that location for later steps.

2. Install Homebrew (https://brew.sh/). Open your command terminal and paste in this command:

   `/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"`

   This will also install the Xcode command line tools if you don't have them. This process might take some time, so be patient.

3. Use Homebrew (or 'brew') to install the libraries and other support files MPF needs:

   `brew install SDL2 SDL2_mixer SDL2_image gstreamer gst-plugins-base gst-plugins-good gst-plugins-bad gst-plugins-ugly`

4. Verify that pip is installed. If you installed Python from python.org, then pip should have been installed as well. You can verify this by running `pip --version` or `pip3 --version` in your terminal. If it's not installed, you can install it using `brew install pip`.

5. Use pip to install MPF with the Text UI components:

   `pip install "mpf[cli]"`

6. Use pip to install the MPF Monitor (Note that the latest version requires PyQt6, priors required PyQt5):

   `pip install mpf-monitor`

7. Use pip to install MPF-MC:

   `pip install mpf-mc`

Note: For the commands that use pip, if you run into permission issues, try prefixing the command with sudo (i.e., `sudo pip install "mpf[cli]"`). Be aware that sudo allows the command to run with root permissions, which can pose a security risk if used carelessly.

Also, pip installs Python packages globally by default. If you'd prefer to keep your project and its dependencies isolated from your system's Python, consider using a Python virtual environment. There are several tools available for this, such as pipx, venv, or virtualenv.

Testing

May 2023 Note: Some of this might not work: The mc_demo and demo_man mentioned below might not work anymore as they haven't been updated in a while. Feel free to fix and/or update them and we'll merge your changes in!

To test, download the `mpf-examples` repo from here: https://github.com/missionpinball/mpf-examples. You can either clone it locally, or download the zip file and unzip it. Either is fine, just do what you're most comfortable with. Be sure to download / switch to the `dev` branch.

Then back in the terminal, change into the `mpf-examples` folder (or whatever folder you just unzipped that into), then change into the `mc_demo` folder, then run `mpf both`. That should launch the mc_demo code (which is Media Controller demo). A window should open with a red background and some text about slides, you should be able to use the right arrow key to advance to the next slide. You should be able to use the left arrow key to go back to the previous slide and you should hear a drum and cymbal sound when you change the slide.

You will see a bunch of warnings about some classes implemented in multiple locations, and how one will be used, but which one is undefined. It sounds scary, but this is normal. (For now.) We are investigating whether this is something we need to fix, and how we'll fix it if it is. But for now, it's fine.

You can also run the "demo_man" game from the `mpf-examples` folder. Change into the `demo_man` folder and run `mpf both -X`. You should see the DMD window pop up. The window you ran the command from will have some warnings which cover up the nice text UI display. Just grab a corner of the window with the mouse and resize the window (just make it a tiny bit bigger and smaller) and that will cause the window contents to completely refresh and you should see the expected MPF text UI display showing switch status, ball locations, etc. (See the screenshots below for details)

At this point, MPF is ready to go!

Notes, Caveats & Next Steps

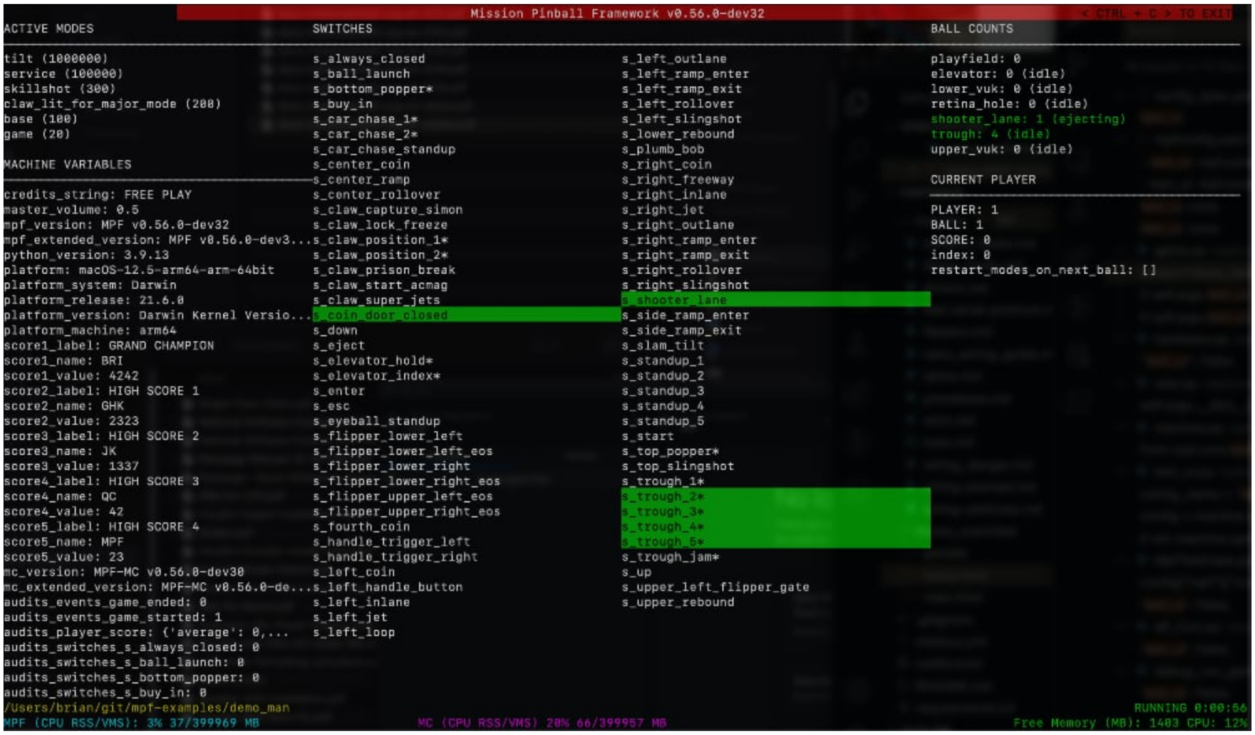
If have existing SDL and Gstreamer libraries installed (check the `/Library/Frameworks` folder), you can delete them. The versions that brew installs will go into the `/opt/homebrew` folder.

Do NOT use brew to install Python. Why? Because the Python in brew is meant to support other brew packages that need python, and as such it will automatically "upgrade" you to the latest Python, even on its own, which means your Python will flip to 3.10 and MPF won't work and you'll be sad. So that's why we install the "Framework Python" from python.org. (Why's it called "Framework Python"? Because it installs like a framework to that `/Library/Frameworks` folder.)

The Apple Silicon Macs need the Xcode command-line tools installed because the ruamel.yaml library that MPF uses to read YAML files doesn't have a pre-built version for M1/M2 Macs, so it has to be built locally. This process is automatic and transparent when the Xcode tools are installed.

If you do not see the "normal" MPF text UI display, and instead see something like this:

This is because those warnings mentioned above print on top of the nice MPF display. To fix this, just grab a corner of the window with the mouse and resize it to be a bit bigger and then smaller again. This will cause the entire window to update and you should see the expected MPF text UI display showing switch status, ball locations, etc. (See the screenshots below for details)



Alternately, if you don't want to resize the window every time, you can open two different terminal windows, and run `mpf -X` in one and `mpf mc` in the other.

Keeping MPF up-to-date

Once you have MPF installed via the procedure above, you can keep it up-to-date by running the final two pipx commands from above which you used to install MPF and MPF-MC.

Questions? Comments? Need help? You can post to the MPF Users Google Group:
https://groups.google.com/g/mpf-users/c/BIemtw17lx0

What if you borked it?

There aren't too many things that could go wrong, but if your environment is broken and you want to remove everything and start over, here are some notes:

To remove homebrew, run the following command:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/uninstall.sh)"
```

Homebrew installs everything to `/opt/homebrew`, which means if you just delete that folder, everything will be gone.

To fix Python versions:

Another problem is sometimes the system's default Python will be the homebrew one, and not that one that you installed from python.org. This can be a problem because MPF requires Python 3.7, 3.8, or 3.9 (3.9 only on M1/M2 Macs), but the homebrew python could be version 3.10 which won't work with MPF. So if you need to check or change this, you can use the following command:

```
which python3
```

You will see a path to the version of python that runs when you just type `python3` from the command line. Ideally you want it to be the version you installed, which will be:

```
/Library/Frameworks/Python.framework/Versions/3.9/bin/python3
```

If you see something else, then run `which -a python3` to see what other versions are installed. Then copy the path to the version you installed (which will be the `/Library/Frameworks/...` version), and use that in Step 5 (the initial pipx installation command) when you install MPF.

To fix errors about failing to load assets:

If you get an error about a failure while loading assets, and you see some references to PIL, there's a potential conflict with an image library that you can remove. To do that, use the following command:

```
pipx runpip mpf uninstall pillow
```

This command uses pipx to run a pip command inside the mpf environment to uninstall a package called pillow.

Installing MPF 0.56 on Windows

This process walks through installing MPF 0.56 a Windows machine.

Note that installing MPF is more complicated than a normal application. This is because MPF is a development tool you use to create your pinball software, not a finished app itself. So, like everything in pinball, there are a lot of steps.

Note: If you're an expert Python user, you can skip most of this page. Just know you need Python 3.9 (newer or older won't work, see below), and you can install MPF-MC via pip. `pipx install mpf-mc` This will also install MPF. You probably also want to install MPF Monitor via `pipx install mpf-monitor`.

Remove prior versions of MPF

MPF and the MPF-MC have many dependencies and requirements. If you have an older version of MPF installed, it may have installed some of these dependencies in a way that is incompatible with the latest version of MPF. So it's best to remove any prior versions of MPF before installing the latest version.

Uninstall MPF by using this command:

```
pip uninstall mpf
```

If you are unsure which version of MPF you have installed, you can run this command to check what is installed:

```
mpf --version
```

Windows System Requirements

MPF 0.56 requires Python 3.9. It does not run on Python 3.10 or newer. (If you can figure out how to get it to run on Python 3.10+, please submit the fixes!)

MPF can run on Python 3.7 and 3.8, but the MPF-MC audio doesn't work on those versions. So you should use Python 3.9.

Install Python

If you open a command prompt on a fresh Windows machine and type `python`, the Microsoft App Store will open and try to install Python 3.7. This is not the version of Python you want. So don't do that.

So instead, install Python from python.org. Here's the direct link to the newest installer for the final version of Python 3.9: https://www.python.org/ftp/python/3.9.13/python-3.9.13-amd64.exe

Run the installer, and choose the "Customize installation" option, and make sure to check the box for "py launcher" and "pip". (The others don't matter either way.) Then on the next screen, check the box for "Add Python to environment variables". Then click "Install".

Then open a command prompt (you can just run "cmd"), and type each of these commands one at a time (and hit enter after each one). First, make sure Python is installed and make sure when you run "python" that it's the proper version. (Sometimes Windows will have multiple versions of Python installed and it's not always clear which one will be used when you type "python".)

```
python --version
```

You want a result like "Python 3.9.13" or whatever version you just installed. If you see a version other than that, trying running `python3 --version` instead.

Install MPF

Now you're ready to install MPF. Open a new command window (cmd.exe) and type these commands and hit enter.

```
pip install "mpf[cli]"
```

A bunch of things will scroll by, and then hopefully MPF is installed. You can test it by typing this command:

```
mpf --version
```

This should print out something like *MPF 0.56.0*. If you get an error, something went wrong. If you get a different version, then you might have an older version of MPF which you need to uninstall first. (See the "Remove prior versions of MPF" section above.)

Install MPF Monitor

MPF Monitor the developers "god view" into your running MPF code. It has windows to show the states of every device, events, variables, modes, etc. You can even add a photo of your playfield and it will show all the lights and switches on it animated in real time to show you what's happening in your game.

MPF Monitor is also nice because it's a simple install, so you can visualize your game quickly. The MPF-MC (which will drive the displays in your machine) is a more complicated install, so you can get your game working with MPF Monitor first, and then add the MPF-MC later.

```
pip install mpf-monitor
```

You can now proceed with the getting started tutorials, or, go on to install the MPF Media Controller (MPF-MC).

Install the MPF Media Controller (MPF-MC)

The MPF Media Controller (MPF-MC) is a standalone package used to control the graphics, sounds, and music in a pinball machine. It's a separate package from MPF. Not every pinball machine uses MPF-MC, but most do. (There are also other media controllers that are not MPF-MC. For example, some people use Unity, the Unreal Engine, or Godot as their media controllers.)

To install MPF-MC, use the following command:

```
pip install mpf-mc
```

If you encounter permission issues during the installation, try running the commands with administrator privileges (Right-click on cmd.exe -> Run as Administrator).

Testing MPF-MC

Installing MPF-MC is pretty straightforward. Unfortunately just because it installs doesn't mean it works. :(

May 2023 Note: Some of this might not work: The mc_demo and demo_man mentioned below might not work anymore as they haven't been updated in a while. Feel free to fix and/or update them and we'll merge your changes in!

One way to test the MC is download the `mpf-examples` repo from here: https://github.com/missionpinball/mpf-examples. You can either clone it locally, or download the zip file and unzip it. Either is fine, just do what you're most comfortable with. Be sure to download / switch to the `dev` branch.

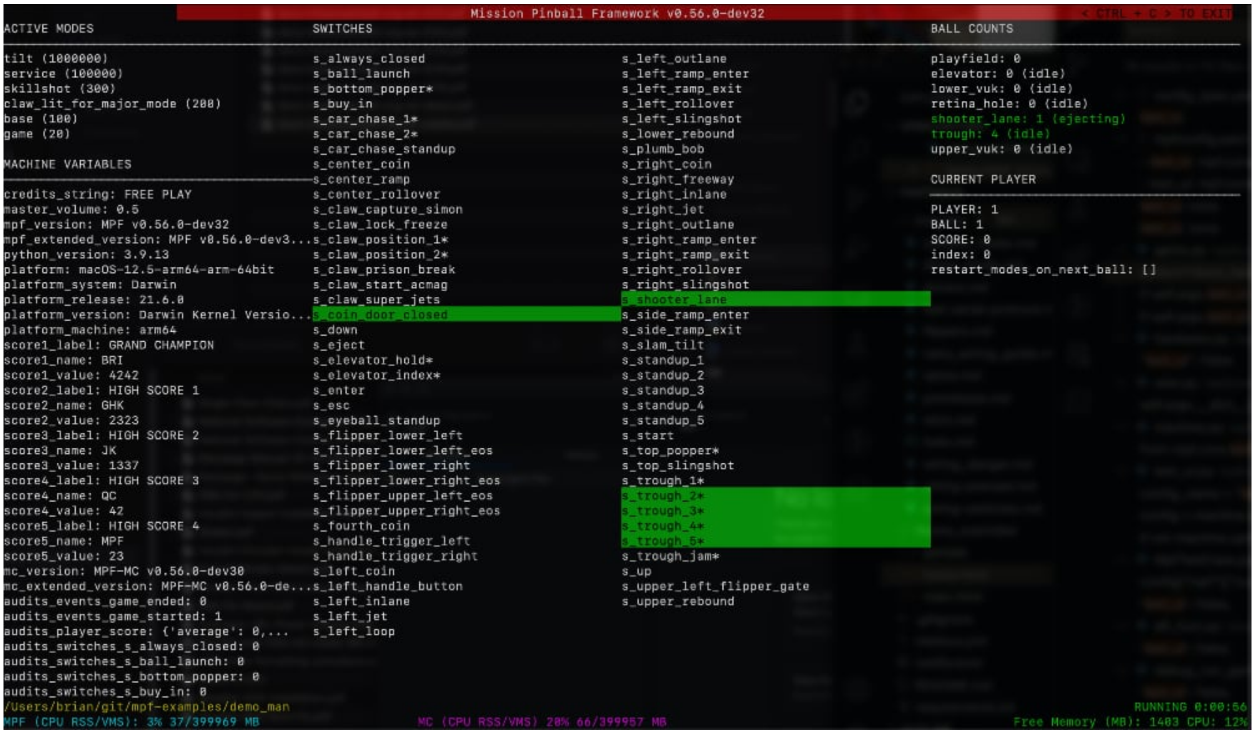
Then back in the command terminal, change into the `mpf-examples` folder (or whatever folder you just unzipped that into), then change into the `mc_demo` folder, then run `mpf both`. That should launch the mc_demo code (which is Media Controller demo). A window should open with a red background and some text about slides, you should be able to use the right arrow key to advance to the next slide. You should be able to use the left arrow key to go back to the previous slide and you should hear a drum and cymbal sound when you change the slide.

You will see a bunch of warnings about some classes implemented in multiple locations, and how one will be used, but which one is undefined. It sounds scary, but this is normal. (For now.) We are investigating whether this is something we need to fix, and how we'll fix it if so. But for now it's fine.

You can also run the "demo_man" game from the `mpf-examples` folder. Change into the `demo_man` folder and run `mpf both -X`. You should see the DMD window pop up. The window you ran the command from will have some warnings which cover up the nice text UI display. Just grab a corner of the window with the mouse and resize the window (just make it a tiny bit bigger and smaller) and that will cause the window contents to completely refresh and you should see the expected MPF text UI display showing switch status, ball locations, etc. (See the screenshots below for details)

If you do not see the "normal" MPF text UI display, and instead see something like this:

This is because those warnings mentioned above print on top of the nice MPF display. To fix this, just grab a corner of the window with the mouse and resize it to be a bit bigger or smaller, which will cause the entire window to update and you should see the expected MPF text UI display showing switch status, ball locations, etc. (See the screenshots below for details)



Alternately if you don't want to resize the window every time, you can open two different command prompt windows, and run `mpf -X` in one and `mpf mc` in the other.

At this point, MPF is ready to go!

Keeping MPF up-to-date

Once you have MPF installed via the procedure above, you can keep it up-to-date by running the final two pip commands from above which you used to install MPF and MPF-MC.

Questions? Comments? Need help? You can post a reply into the MPF new installers for macOS thread in the MPF Users Google Group: https://groups.google.com/g/mpf-users/c/BIemtw17lx0

Installing MPF on Linux

May 2023 Update

We need to update this page (we no longer suggest pipx, btw), but if you want the "real" linux install instructions, read through the GitHub Actions job scripts in the `.github/workflows/build_wheel.yml` file. You'll see the exact steps we use to build MPF-MC (MPF is simple to install, it's the MC part that's tricky). Various jobs in there use both apt-get and yum to set everything up before building the MC.

As part of our automated build process, we build and test MPF and MPF-MC against Ubuntu 20.04 & 22.04 and Debian Stretch & Buster. MPF 0.54 supports Python 3.5 to 3.7. MPF 0.55 supports Python 3.6 to 3.9. MPF 0.56 supports Python 3.7 to 3.9

Installing MPF Using Our Installer

Download the MPF Debian Installer (which is used for all of these) from https://github.com/missionpinball/mpf-debian-installer/archive/0.55.x.zip

Unzip it, and from a terminal run `chmod +x install && sudo ./install` from the folder you unzipped the files to. If you are using a P-Roc or P3-Roc also run `chmod +x install-proc && ./install-proc` (skip for other platforms). Consult the README for more information.

Installing MPF Manually

Current version these instructions are for v0.56.0, which is the current "dev" branch of MPF.

MPF 0.56 requires Python 3.7, 3.8, or 3.9. Various Linux distributions offer to install multiple versions of Python in parallel. First check what version of Python you might have already running on your computer.

```
python3 --version
```

In some cases you might only get a Python version of 3.6, then run your admin tool and install a higher version of python, preferabyl the latest version of 3.9. If you don't want to remove the older version of Python you can keep it in parallel, just make sure to run the installer commands with the right version of python. If you have for example installed Python 3.9 try to running

```
python3.9 --version
```

You will need pip further down the line to complete the installation. Same as above for Python, check what version of pip you are running and choose the command to match the version you would like to use. Try the following commands to figure out what pip command works for you and has the right version.

```
pip --version
pip3 --version
pip3.9 --version
```

For the rest of the chapter I will always write python3.9 and pip3.9 as commands as reminder to use the right version, any of the above commands might work for you (or fail).

NOTE: If you already have an earlier version of MPF installed, uninstall that first by using this command:

```
sudo pip3.9 uninstall mpf-mc mpf
```

If you are unsure which version of MPF you have installed, you can run this command to check what is installed:

```
mpf --version
```

Now back to the installation, in a console run

```
pip3.9 install --user pipx
python3.9 -m pipx ensurepath
```

After this it might be necessary to restart the console. Now run the following command (obey that this is pipx and here there is no need for a version)

```
pipx install "mpf[cli]" --pip-args="--pre" --verbose --include-deps
pipx inject mpf mpf-mc --pip-args="--pre" --verbose --include-deps --include-apps
```

Updated MPF Monitor instructions (which work with pipx) are *here*.

At this point, MPF 0.56.0.devXX and MPF-MC 0.56.0.devXX are installed. (The "XX" in the version will be the dev build numbers.)

Download & run the "Demo Man" example game

Now that you have MPF installed, you probably want to see it in action. The easiest way to do that is to download a bundle of MPF examples and run our "Demo Man" example game. To do that, follow the instructions in the *How to run "Demo Man", an MPF example game* guide.

There's another example project you can also check out if you want called the "MC Demo" (for media controller demo) that lets you step through a bunch of example display things (slides, widgets, sounds, videos, etc). Instructions for running the MC Demo are *here*.

Running MPF

See the *How to start MPF and run your game* for details and command-line options.

If you see a Failed to initialize MPF exception when trying to start MPF with Multimorphic P-ROC/P3-ROC boards with Python 3.8.

Example Error:

Try Changing *Edit install-proc:*

From:

To:

Keeping MPF up-to-date

To upgrade MPF just re-run the installer which will make sure that you will also get updated dependencies:

```
sudo ./install
```

If you have MPF installed via the manual procedure above, you can keep it up-to-date by running the final two pipx commands from above which you used to install MPF and MPF-MC.

> **Warning:** If you are upgrading from MPF 0.33 to 0.50 you will need to manually perform several migration steps to modify your configuration files or they will not work in MPF 0.50. Please refer to Migrating from config version 4 to 5 of MPF for step-by-step instructions.

To install the latest dev release (not generally recommended) which allows you to try bleeding-edge features run:

```
pip3 install mpf[all] mpf-mc --pre --upgrade
```

To downgrade (or install a specific release x.yy.z) run:

```
pip3 install mpf[all]==x.yy.z
pip3 install mpf-mc==x.yy.z
```

Uninstalling MPF

To remove MPF either because it is no longer needed or to perform a clean install run:

```
sudo pip3 uninstall mpf-mc mpf
```

Specific Hardware Devices

We got some write-ups for specific hardware platforms. They follow the general linux installation schema but also cover some details about that hardware.

Installing MPF on a Raspberry Pi 3

Warning: Raspberry Pi support is experimental at this point. Users have found various issues with audio, and we're not sure whether the RPi has enough power to support MPF. So this document is more like a collection of notes versus a solid guide. We welcome your feedback or experience with other low-cost systems, though at this point if you're looking for a development platform, we'd probably recommend buying a more beefy x86 computer. For a "final" machine an inexpensive (<$200) Intel-based system running Linux or Windows might be better suited. However, it should be possible to run your final game on a RPi3+ if you tune your game accordingly. For example, this would include transcoding your videos to a format which can be played hardware accelerated on the RPi.

One first word: Don't try to install mpf on a Raspberry Pi B+ or Raspberry zero, it just won't work or will be very slow. Get yourself at a Raspberry Pi 3, they have a quad-core processor running with more than 900MHz. RPi3 also has better audio than RPi 1 (still not perfect). An HDMI audio adapter may be worthy for better audio. If you want to try this get at least a RPi4 with 2GB of RAM.

We previously recommended KivyPI but it only works for MPF < 0.54 and only the RPi3. Instead, we propose you install Raspbian and install all other parts yourself:

- Get the latest Raspberry Pi Imager from here: https://www.raspberrypi.org/software/
- Install Raspbian Lite onto your SD card.
- Boot your PI and connect keyboard + monitor
- Login with user:pi password:raspberry
- now type this:

```
sudo raspi-config
```

and choose 7. Advanced Options -> A1. Expand Filesystem to use the whole SD-Card Space, we will need it. You can change your username and localization settings too.

After that we will give the GPU a bit more of RAM:

Go to 7. Advanced Options -> A3 Memory split and change the value to 256.

Now reboot, login and type:

```
sudo apt update
sudo apt upgrade
sudo apt install git

git clone https://github.com/missionpinball/mpf-debian-installer.git
cd mpf-debian-install
sudo ./install
```

To checkout and run the *MPF Linux Debian installer*. It will install MPF, MPF-MC and all dependencies for you.

This will take some time as it may compile some drivers mpf-mc needs like the audio driver. Sometimes it looks like it hangs, but it does not. It will take up to half an hour, at least on a Raspberry 1 (which you should not use). Compiling is really slow on the Raspi.

Now copy your machine folder from your develop station or create a new one under your home directory (/home/pi/your_machine)

If you need a file-manager start mc (No, not the mpf mediacontroller, its the midnight commander ;-))

If you need to copy your folders from an usb-stick you have to manually mount it (we dont have X, so everything has to be done by hand).

```
sudo mount /dev/sda1 /mnt
```

This works in 90% otherwise your stick is not sda1, just look inside the /dev folder to find out which device you have to mount or type

```
lsblk
```

to list your block devices.

Now you find the contents of your stick in /mnt.

To tell mpf-mc and the underlying kivy to use the framebuffer via SDL2 you have to put this in your machine/config/config.yaml:

```
window:
  width: 1280
  height: 800
kivy_config:
  graphics:
    fbo: force-hardware
```

More or less important last steps:

Serial communication:

Linux always had and has the possibility to log in via a serial connection. If you run a hardware platform which uses the serial pin on the Raspberry you should disable the Linux login shell on that port. The device is called /dev/ttyAMA0 and you need to stop it from starting:

Type:

```
sudo systemctl disable serial-getty@ttyAMA0.service
```

Now you have to disable the console itself:

```
sudo mc
```

to start Midnight Commander as root (normally you should not do this, but this time you have to.)

Now go to /boot and press F4 over cmdline.txt.

Remove these entries:

```
console=ttyAMA0,115200 kgdboc=ttyAMA0, 115200
```

and save the file.

You have the possibility to connect RS 232 devices directly to the raspi but take care, the voltage levels are 3.3V on the raspi gpio. Further instructions here: http://elinux.org/RPi_Serial_Connection

Sound output:

Navigate to /boot/config.txt if you want to use audio out of the Raspberry built in “”soundcard””: edit this file as root and insert this line:

```
dtparam=audio=on
```

Inside this file you can change some settings that initialize on boot, its like a bios which the raspberry does not have.

Video Playback:

If you need video capability in your mpf-mc you need to install one player that kivy will use to play your videos:

```
sudo apt-get install omxplayer
```

You can try videoplayback with

```
omxplayer your_video.mp4
```

To test the video playback capability under kivy into the framebuffer just run this command:

```
python3 -m kivy.uix.videoplayer /usr/local/lib/python3.4/dist-packages/mpfmc/tests/machine_files/video/
↪videos/mpf_video_small_test.mp4
```

Troubleshooting:

More documentation about kivypie can be found here: http://kivypie.mitako.eu/kivy-faq.html

No sound:

If you have trouble getting sound out of your speakers or monitor have a look here:

https://www.raspberrypi.org/documentation/configuration/audio-config.md

If sound plays via omxplayer but not in MPF, set use_sdl_mixer_loader: False in your MPF configuration file.

Do a reboot:

```
sudo reboot
```

Remote log in:

To log in from your development machine into your raspberry you can do it easily via ssh. For windows I recommend putty: http://www.putty.org/

See whats going on on your pinball:

```
sudo dispman_vncserver
```

This starts a vncserver on your raspi and you can log in remotely from a RealVNCViewer https://www.realvnc.com/download/viewer/

Kivypie IP address, port 5900. It is not 100% reliable but fairly usable. Thanks to Peter Hanzel.

Start mpf and mpf-mc

To test your installation type

```
mpf
```

in your machine_folder.

Press (STRG+ALT F2) to change to the second terminal tty2.

Login and start mpf-mc inside your machine folder with

```
mpf mc
```

Enjoy!

What if it did not work?

In the following we list some common problems and solutions. If you got another problem please ask in our MPF User Forum.

YAML error on first start

You will see this error if there is already a different version of ruamel.yaml installed on your system:

```
pkg_resources.VersionConflict: (ruamel.yaml 0.15.37␣
↪(c:\users\robert\appdata\local\programs\python\python36\lib\site-packages), Requirement.parse('ruamel.
↪yaml<0.11,>=0.10')
```

This could have happened if you are upgrading to a newer version of MPF or you have incompatible versions of MPF, MPF-MC and/or the MPF-Monitor installed. The required ruamel.yaml version is different on newer MPF versions. We used to install ruamel 0.11 and switched to 0.15 in MPF 0.53+. MPF cannot start with two yaml libraries. To fix this check your versions `pip3 list` and check `mpf`, `mpf-mc` and `mpf-monitor`. Remove the wrong version and install the right one. All versions need to match (for instance all 0.56 or all dev).

The following command will remove all three and install the latest release:

```
pip3 uninstall mpf mpf-mc mpf-monitor
pip3 install mpf mpf-mc mpf-monitor
```

This error can also occur if you already have ruamel.yaml installed in your python system packages for a non-MPF package. Often times, you will have a newer version of ruamel.yaml than MPF requires. Unfortunately, MPF cannot use a newer version of this package because that caused issues in the past because newer versions dropped support (wheels for windows) for older python versions. In the case that you need a different version than the one MPF requires, it is advised to create a python virtual environment and install the required packages there, and use that virtual environment for running MPF.

Installing MPF on a Pine64 with Ubuntu

Note: This procedure for installing MPF on a Pine64 does not fully work. (MPF runs fine, Kivy installs fine, but MPF-MC does not run.) If you want to use MPF on a Pine64, maybe you can help figure out why this doesn't work and share your findings with us?)

Hardware Notes

- Spring for the fastest MicroSD card you can (Samsung Evo cards are reportedly the fastest), at least 16GB.
- The Pine64's video seems to only support 1080p and 4K resolutions, so make sure your display can do one or both of those at a proper 16:9 aspect ratio or else everything will be scaled and squished and it looks awful.
- If you find that your pine64 does not boot it maybe due to using a HDMI->DVI cable, try HDMI to HDMI first.

System Notes

There are a bunch of things that arrive broken with the current Ubuntu installer for Pine64 (as of this writing in November 2016). Some of them will prevent MPF from installing, and a few are just annoying.

Instructions

After installing the OS following the instructions on the Pine64 Wiki, expanding the volume to the full size of the SD card, and getting connected to the Internet, follow these steps. Don't try to update the installed system before following this.

Locale

Locale arrives broken and this wreaks all kinds of havoc, so here's how to fix it.

Assuming you want US English, substitute your preferred language if not:

```
$ sudo locale-gen "en_US.UTF-8"
Generating locales...
  en_US.UTF-8... done
Generation complete.

$ sudo dpkg-reconfigure locales
Generating locales...
  en_US.UTF-8... up-to-date
Generation complete.
```

That command will open a text-based dialog, we recommend that you don't choose "ALL" and only select the one or a few languages you want (generating them all takes a long time). Then reboot, then do the above reconfigure step AGAIN, then reboot, then run:

```
$ locale
```

And make sure it looks good. Mine says:

```
LANG=en_US.UTF-8
LANGUAGE=en
LC_CTYPE="en_US.UTF-8"
LC_NUMERIC=en_US.UTF-8
LC_TIME=en_US.UTF-8
LC_COLLATE="en_US.UTF-8"
LC_MONETARY=en_US.UTF-8
LC_MESSAGES="en_US.UTF-8"
LC_PAPER=en_US.UTF-8
LC_NAME=en_US.UTF-8
LC_ADDRESS=en_US.UTF-8
LC_TELEPHONE=en_US.UTF-8
LC_MEASUREMENT=en_US.UTF-8
LC_IDENTIFICATION=en_US.UTF-8
LC_ALL=
```

It took a few tries for this to stick for me, so do it again, including reboot, if your results here are wrong.

Fix the Software Boutique

This arrives broken, too. Oddly, running the Mate Welcome as root and clicking a button partly fixes it.

```
$ sudo ubuntu-mate-welcome
```

When it comes up, click on the "Subscribe to updates" button, then quit it.

Now go to System -> Administration -> Software Boutique. Click on the wrench, then do each repair option (after clicking one, wait for it to say it has finished).

Now go to System -> Administration -> Software Updater and get everything up to date. You will need to reboot again after that.

Install Missing pip3

```
$ apt-get install python3-pip
```

The path where `pip` puts executables is not in the system default path, so edit `~/.bashrc` to add the following path:

```
$ sudo nano ~/.bashrc
```

At the bottom of the file add the following:

```
export PATH=~/.local/bin:$PATH
```

Hit "control + x" to save and "y" then "return" to save the file as the same name.

Now start a fresh terminal so that this new PATH is included in your current environment. Then:

Install MPF

Download the MPF Debian Installer from https://github.com/missionpinball/mpf-debian-installer/archive/0.55.x.zip

To unzip the file navigate in your terminal to the location of the downloaded files.

Unzip the file:

```
$ unzip dev.zip .
```

If this does not run you may need to install unzip:

```
$ sudo apt-get install unzip
```

After unzip, run ./dev/install from the folder you unzipped the files to. Consult the README for more information.

Running MPF

See the *How to start MPF and run your game* page for details and command-line options.

What if it did not work?

In the following we list some common problems and solutions. If you got another problem please ask in our MPF User Forum.

YAML error on first start

You will see this error if there is already a different version of ruamel.yaml installed on your system:

```
pkg_resources.VersionConflict: (ruamel.yaml 0.15.37
↪(c:\users\robert\appdata\local\programs\python\python36\lib\site-packages), Requirement.parse('ruamel.
↪yaml<0.11,>=0.10')
```

This could have happened if you are upgrading to a newer version of MPF or you have incompatible versions of MPF, MPF-MC and/or the MPF-Monitor installed. The required ruamel.yaml version is different on newer MPF versions. We used to install ruamel 0.11 and switched to 0.15 in MPF 0.53+. MPF cannot start with two yaml libraries. To fix this check your versions `pip3 list` and check `mpf`, `mpf-mc` and `mpf-monitor`. Remove the wrong version and install the right one. All versions need to match (for instance all 0.56 or all dev).

The following command will remove all three and install the latest release:

```
pip3 uninstall mpf mpf-mc mpf-monitor
pip3 install mpf mpf-mc mpf-monitor
```

This error can also occur if you already have ruamel.yaml installed in your python system packages for a non-MPF package. Often times, you will have a newer version of ruamel.yaml than MPF requires. Unfortunately, MPF cannot use a newer version of this package because that caused issues in the past because newer versions dropped support (wheels for windows) for older python versions. In the case that you need a different version than the one MPF requires, it is advised to create a python virtual environment and install the required packages there, and use that virtual environment for running MPF.

Specific Linux Distributions

Specifics about certain linux distributions.

Installing MPF on Xubuntu/Lubuntu

Xubuntu is a Ubuntu-based linux distribution using the minimalist, yet still feature-packed, XFCE desktop manager. The focus of this guide will be for getting MPF up and running directly from power (unattended) for use in a production scenario.

1. Create Xubuntu/Lubuntu Installation Media

You will need:

- 4GB USB Flash Drive or larger

Write the ISO (Win/Mac/Linux)

Use UNetbootin

- Select LUbuntu or XUbuntu
- Select your USB Stick

2. Install Xubuntu/Lubuntu

Boot from the installation media (you may need to change something in your BIOS to enable booting from USB). It should be a fairly straight-forward linux installation. When it asks about partioning, choose the "Guided - entire hard disk" option (unless you have a specific reason not to). You will be asked to create a user account. When doing so, it's important that you: **DO NOT ELECT TO**

ENCRYPT THE HOME FOLDER. If you encrypt the home folder, the auto login will not work and will have to reinstall to fix.

3. Configure Xubuntu/Lubuntu

The system will reboot after installation. Login with your username and password then follow these steps:

- Launch a Terminal emulator
- Update the sources: `sudo apt-get update`
- Upgrade all the things: `sudo apt-get upgrade`
- **Setup auto-login to the XFCE desktop**
    - Create the file `/etc/lightdm/lightdm.conf.d/12-autologin.conf` and edit it to contain:

```
[Seat:*]
autologin-user=your_username
autologin-user-timeout=0
```

- Be sure to change `your_username` to the username you created during installation.
- **Optional: Reduce the Network Timeout**
    - You should do this if the system will not always be connected to the internet
    - Edit the file `/etc/systemd/system/network-online.targets.wants/networking.service`
    - Find the line `TimeoutStartSec=5min` and change to `TimeoutStartSec=10sec`

4. Install MPF

The existing Debian install script works perfectly on Ubuntu. The following commands will install the current versions of MPF and MPF-MC as well as each of their dependencies.

```
cd ~
wget https://github.com/missionpinball/mpf-debian-installer/archive/0.55.x.zip
unzip dev.zip
cd mpf-debian-installer-dev
sudo -H ./install
rm ~/dev.zip && rm -Rf ~/mpf-debian-installer-dev
```

If you want to make sure that MPF was installed, you can run:

```
mpf --version
```

This command can be run from anywhere and should produce output something like this:

```
username@host:~$ mpf --version
MPF v0.33.13
```

(Note that the actual version number of your MPF installation will be whatever version is the latest.)

5. Setup your Machine Config

- Copy your machine config root folder to `~/` which is the same as `/home/your_username/`.
- **Create a new file named `run.sh` in `/home/your_username/your_machine_folder/`**
    - Edit the file to contain:

```
#!/bin/bash
xterm -e "cd /home/your_username/your_machine_folder && mpf both -c config"
```

- Change `your_username` to the username you created during installation.
- Change `your_machine_folder` to the name of your specific machine folder.
- Change `config` part to reflect the name of your top-level config file in `~/your_machine_folder/config/`.

6. Setup your Machine Config to Auto-execute

When XFCE is executed, it runs all the *Desktop Entries* found within `~/.config/autostart`. We'll create one of our own to run the script we just added to our machine config.

- Create the file `~/.config/autostart/mpf.desktop` and edit it to contain:

```
[Desktop Entry]
Version=1.0
Name=MPF
Comment=Mission Pinball
Exec=/home/your_username/your_machine_folder/run.sh
Path=/home/your_username/your_machine_folder/
Terminal=false
Type=Application
```

- Change `your_username` to the username you created during installation.
- Change `your_machine_folder` to the name of your specific machine folder.

That's it. At this point, you should be able to reboot and watch the system auto-login to XFCE and then launch MPF using the script we added to your machine config.

Other Considerations

If using the SmartMatrix RGB DMD with this setup, you need to add the system user running your game to the `dialout` group.

```
sudo usermod -a -G dialout your_username
```

What if it did not work?

In the following we list some common problems and solutions. If you got another problem please ask in our MPF User Forum.

YAML error on first start

You will see this error if there is already a different version of ruamel.yaml installed on your system:

```
pkg_resources.VersionConflict: (ruamel.yaml 0.15.37 (c:\users\robert\appdata\local\programs\python\python36\lib\site-packages), Requirement.parse('ruamel.yaml<0.11,>=0.10')
```

This could have happened if you are upgrading to a newer version of MPF or you have incompatible versions of MPF, MPF-MC and/or the MPF-Monitor installed. The required ruamel.yaml version is different on newer MPF versions. We used to install ruamel 0.11 and switched to 0.15 in MPF 0.53+. MPF cannot start with two yaml libraries. To fix this check your versions `pip3 list` and check `mpf`, `mpf-mc` and `mpf-monitor`. Remove the wrong version and install the right one. All versions need to match (for instance all 0.56 or all dev).

The following command will remove all three and install the latest release:

```
pip3 uninstall mpf mpf-mc mpf-monitor
pip3 install mpf mpf-mc mpf-monitor
```

This error can also occur if you already have ruamel.yaml installed in your python system packages for a non-MPF package. Often times, you will have a newer version of ruamel.yaml than MPF requires. Unfortunately, MPF cannot use a newer version of this package because that caused issues in the past because newer versions dropped support (wheels for windows) for older python versions. In the case that you need a different version than the one MPF requires, it is advised to create a python virtual environment and install the required packages there, and use that virtual environment for running MPF.

What if it did not work?

In the following we list some common problems and solutions. If you got another problem please ask in our MPF User Forum.

YAML error on first start

You will see this error if there is already a different version of ruamel.yaml installed on your system:

```
pkg_resources.VersionConflict: (ruamel.yaml 0.15.37 (c:\users\robert\appdata\local\programs\python\python36\lib\site-packages), Requirement.parse('ruamel.yaml<0.11,>=0.10')
```

This could have happened if you are upgrading to a newer version of MPF or you have incompatible versions of MPF, MPF-MC and/or the MPF-Monitor installed. The required ruamel.yaml version is different on newer MPF versions. We used to install ruamel 0.11 and switched to 0.15 in MPF 0.53+. MPF cannot start with two yaml libraries. To fix this check your versions `pip3 list` and check `mpf`, `mpf-mc` and `mpf-monitor`. Remove the wrong version and install the right one. All versions need to match (for instance all 0.56 or all dev).

The following command will remove all three and install the latest release:

```
pip3 uninstall mpf mpf-mc mpf-monitor
pip3 install mpf mpf-mc mpf-monitor
```

This error can also occur if you already have ruamel.yaml installed in your python system packages for a non-MPF package. Often times, you will have a newer version of ruamel.yaml than MPF requires. Unfortunately, MPF cannot use a newer version of this package because that caused issues in the past because newer versions dropped support (wheels for windows) for older python versions. In the case that you need a different version than the one MPF requires, it is advised to create a python virtual environment and install the required packages there, and use that virtual environment for running MPF.

CHAPTER 4

How to start MPF and run your game

MPF is a console-based application which you run from the command line.

The quick version

1. Open a command prompt
2. Switch to your machine folder
3. Run `mpf both`

Starting the MPF game engine and media controller together

You can start both the MPF game engine and the *media controller* at the same time with a single command.

Since this is done from the command line, you'll need to open a command line window. On Windows, you can right-click on the Start Button (or whatever it's called these days) and click the "Command Prompt". On Mac OS X you can run the Terminal app. On Linux, well, if you're using Linux, you know what a command line is. :)

From the command line, change to the directory which is the root of your machine folder. This is the folder that contains your machine's `config`, `modes`, `shows`, etc. folders.

Note: Prior to MPF 0.30, we recommended that you put your machine's folder in the `/machine_files` folder inside the MPF package. That is changed now, and you can put your machine folder(s) wherever you want. In fact now that MPF has a "real" installer, the MPF package folder is hidden deep inside your system.

Then run:

```
mpf both <enter>
```

The `mpf both` command is what we use and probably what you'll use 99% of the time.

Starting the MPF media controller

Alternately you can choose to run just the media controller by itself (still from within your machine folder) like this:

```
mpf mc <enter>
```

You should see a popup window and a bunch of stuff scroll by in the console.

Starting the MPF game engine

You can run the MPF game engine by itself by running:

```
mpf game <enter>
```

Note that if you do not have a media controller running, the game engine won't start fully because it will get stuck trying to connect to the media controller. To avoid this if you just want to run the game engine by itself, add the `-b` command line option. (Details below)

Specifying command-line options

There are several command-like options you can use when you run MPF. To use them, add them *after* the name of the MPF command you're running, like:

```
mpf game -x -v

mpf mc -xvV

mpf both -v -b
```

The full list of available commands is covered in the documentation for each command (discussed below).

Understanding how this works

When you install MPF, the command `mpf` is registered with your system. Then you can open a command prompt and run "mpf" from any folder.

There are several sub-commands you can specify when you run MPF. You specify a sub-command by running `mpf <command>`. (Some mpf commands take additional options).

Here's a list of valid MPF commands. Click on any one of them for full details and command-line options.

- *mpf* (Starts the MPF game engine and other commands)
- *mpf game* (Starts the MPF game engine)
- *mpf mc* (Starts the MPF Media Controller)
- *mpf both* (Starts both the MPF game engine and media controller at the same time)
- *mpf migrate* (Migrates older config and show files to the current version)
- *mpf hardware* (Scan, inspect and configure hardware)
- *mpf service* (Service command line interface)
- *mpf build* (Build production bundles)

Specifying BCP ports

By default, the MPF game engine and the MC will connect via TCP port 5050. *You can change that port to whatever you want though*.

MPF command-line utility

When you install MPF, it registers an executable called `mpf` and puts it in your system path. Everything you do with MPF will use this tool from the command line.

Simply running `mpf` by itself will start the MPF game engine and run whatever machine configuration is in the current folder. But you can also use `mpf` to launch other things, like `mpf mc` to start the media controller, or `mpf migrate` to migrate your config files to the current version of MPF.

A full list of all the available commands, along with the various command line options, is *here*.

Command line options

–version

Prints the version of MPF and exits:

```
$ mpf --version
MPF v0.xx.yy
```

<command>

Runs the MPF command (with or without additional options).

See the *MPF commands* documentation for options.

MPF commands

MPF offers multiple commandline commands. Almost all of those commands should executed from within your MPF machine folder.

Usually you start MPF using:

```
$ mpf both
```

Alternatively your can run MPF and MC separately:

```
$ mpf game
```

and:

```
$ mpf mc
```

To start the interactive service cli run (start mpf both before):

```
$ mpf service
```

If you want to see details about your hardware (do not run MPF in parallel):

```
$ mpf hardware scan
```

To update the firmware of your hardware controllers (if supported by your platform):

```
$ mpf hardware firmware_update
```

mpf both (command-line utility)

Starts both the MPF game engine and the MPF Media Controller from a single command window with a single command. This is effectively the same as running both `mpf game` and `mpf mc`, but more convenient.

When you run `mpf both`, the console log outputs from both MPF and MPF-MC will be mingled together in the console window. However the log files in your machine's `/logs` folder will still be separate.

Also note that you can pass command line options to both MPF and MPF-MC after the "both" command, like this:

```
mpf both -v

mpf both -v -V -b
```

etc. See the *mpf game (command-line utility)* and *mpf mc (command-line utility)* command references for a full list of command line options.

To quit MPF and MPF-MC, either click in the graphical pop up window (so it has focus) and hit `Esc`, or click in the console window and press `CTRL+C`.

Note: If you use the `-l` (lowercase L) option to specify a log file along with `mpf both`, you need to use `-l` to specify the MPF log and `-L` to specify the MC log.

---

mpf core (command-line utility)

Runs the MPF "core" modules without any game logic. This feature has not been fully implemented yet, but it's being put in place to facilitate using the MPF platform interface to be completely controlled by external sources such as PinMAME without any of MPF's game logic.

mpf diagnosis (command-line utility)

Prints the current installed versions of MPF and the MPF-MC.

mpf game (command-line utility)

Starts the MPF game engine (the main MPF process).

Command line options

There are several command-line options you can use when running MPF. Note that single commands that take no options can be combined, so `mpf game -vVa` is the same as `mpf game -v -V -a`.

-a (lowercase)

Forces MPF to reload the config from the actual YAML config files, rather than from cache.

MPF contains a caching mechanism that caches YAML config files, and if the original files haven't changed since the last time MPF was run, it loads them from cache instead. Cached files are stored in your machine's temp folder which varies depending on your system.

-A (uppercase)

Do not cache the config files.

-b

Disables MPF's BCP interface, meaning MPF will not try to connect to a media controller via BCP. This is used if for some reason you just want to run MPF without MPF MC. Without this option, MPF will not start because it will just sit there trying to connect to the media controller.

-c (lowercase)

Specifies the name of the config file (or files) to load. Default `config.yaml` is used if this option is omitted. You do not have to specify the .yaml extension.

Examples:

Run MPF and load the config file `config/config.yaml`:

```
$ mpf game
```

Run MPF and load the config file `config/nodisplay.yaml`:

```
$ mpf game -c nodisplay
```

You can also chain multiple config files together by specifying a comma-separated list (no spaces). For example, to load `config/config.yaml` first, and then once that's loaded, merge in changes from `config/fast.yaml`, run:

```
$ mpf game -c config,fast
```

To load a machine folder from some other location, such as `/home/brian/pinball/demo_man/config/config.yaml`:

```
$ mpf game -c /home/brian/pinball/demo_man/config/config.yaml
```

-C (uppercase)

Specify the name of the MPF default config file which is loaded before your before your machine config. (MPF includes a file `mpfconfig.yaml` which is inside the MPF package which sets up default things like which modules are loaded, paths used, etc. If for some reason you want to override this file, you can do so with the *-C* option.

-h

Displays the command line help and exits. (Pretty much what's on this page.)

-f

Forces MPF to load all assets at start (rather than the default behavior where some assets can be loaded only when modes start or based on other events). This is useful during development to ensure that all assets are valid and loadable.

-t

Disable Text UI. This can be helpful while debugging and is also recommended when running the machine in production.

-l (lowercase "L")

Specifies the name and path of the log file.

The default stores the log file in the `/logs` folder in your machine folder, with a file name of `<year>-<month>-<day>-<hour>-<min>-<sec>-mpf-<hostname>.log`.

Note that log files are standard log file formats that can be read and parsed with log file utilities. (The "Console" app is built-in to OS X, for example.)

–syslog_address

Log to the specified syslog address. This can be a domain socket such as `/dev/log` on Linux or `/var/run/syslog` on Mac. Alternatively, you an specify `host:port` for remote logging over UDP.

-v (lowercase)

Enables verbose logging to the log file. Warning: Your log files will be huge, perhaps 1MB per minute of game time. Definitely only use this when you're troubleshooting.

-V (uppercase)

Enables verbose logging to the console output.

Note that due to the way the command prompt console works on Windows, enabling verbose logging on Windows will significantly affect MPF (in a bad way). Windows computers can run MPF no problem, but because of their weird console slowness we recommend that you do not use the *-V* command line option from a Windows computer.

-x (lowercase)

Ignores all `platform:` settings in your config files and forces MPF to run using the *virtual platform* interface. This is nice for testing when you don't have your physical hardware attached.

-X (uppercase)

Like -x, except it forces the *smart virtual platform*.

–vpx

Like -x, except it forces the *Virtual Pinball (VPX) platform*.

mpf mc (command-line utility)

Starts the MPF Media Controller.

Command line options

There are several command-line options you can use when running the MPF MC. Note that single commands that take no options can be combined, so `mpf mc -vVb` is the same as `mpf mc -v -V -b`.

-c (lowercase)

Specifies the name of the config file (or files) to load. Default `config.yaml` is used if this option is omitted. You do not have to specify the .yaml extension.

Examples:

Run MPF MC and load the config file `config/config.yaml`:

```
$ mpf mc
```

Run MPF and load the config file `config/nodisplay.yaml`:

```
$ mpf mc -c nodisplay
```

You can also chain multiple config files together by specifying a comma-separated list (no spaces). For example, to load `config/config.yaml` first, and then once that's loaded, merge in changes from `config/fast.yaml`, run:

```
$ mpf mc -c config,fast
```

To load a machine folder from some other location, such as `/home/brian/pinball/demo_man/config/config.yaml`:

```
$ mpf mc -c /home/brian/pinball/demo_man/config/config.yaml
```

-C (uppercase)

Specify the name of the MPF MC default config file which is loaded before your before your machine config. (MPF MC includes a file `mcconfig.yaml` which is inside the MPF MC package which sets up default things like which modules are loaded, paths used, etc. If for some reason you want to override this file, you can do so with the *-C* option.

Note that the `-C` option is used by both `mpf game` and `mpf mc`, but these two packages use different default files. So if you want to override the default, you'll have to make one file that works for both or else launch the MPF game engine and MPF MC separately (e.g. not using `mpf both`.

-h

Displays the command line help and exits. (Pretty much what's on this page.)

-f

Forces MPF to load all assets at start (rather than the default behavior where some assets can be loaded only when modes start or based on other events). This is useful during development to ensure that all assets are valid and loadable.

-l (lowercase "L")

Specifies the name and path of the log file.

The default stores the log file in the `/logs` folder in your machine folder, with a file name of <year>-<month>-<day>-<hour>-<min>-<sec>-mpf-<hostname>.log.

Note that log files are standard log file formats that can be read and parsed with log file utilities. (The "Console" app is built-in to OS X, for example.)

-L (uppercase)

Specifies the name and path of the log file.

Note this is the same as `-l` (lowercase L), but it's included so if you use *mpf both* with manually specified log files that you can use `-l` for the MPF log and `-L` for the MC log.

-v (lowercase)

Enables verbose logging to the log file. Warning: Your log files will be huge, perhaps 1MB per minute of game time. Definitely only use this when you're troubleshooting.

-V (uppercase)

Enables verbose logging to the console output.

Note that on due to the way the command prompt console works on Windows, enabling verbose logging on Windows will significantly affect MPF (in a bad way). Windows computers can run MPF no problem, but because of their weird console slowness we recommend that you do not use the *-V* command line option from a Windows computer.

-x (lowercase)

Ignores all `platform:` settings in your config files and forces MPF to run using the *virtual* platform interface. This is nice for testing when you don't have your physical hardware attached.

-X (uppercase)

Like *-x*, except it forces the *smart virtual* platform.

Unused Options

Note that command line options `-a -A -x -X` are valid but ignored by the MPF MC. This is because these options are used with the MPF game engine, but if you start the MPF game engine and MPF MC at the same time via `mpf both`, all options will be sent to both the game engine and the MC, so the MC ignores these options which it doesn't use.

mpf imc (command-line utility)

Starts the MPF Media Controller with a slide editor.

Command line options

There are no commandline option. Just start it from within your machine folder.

mpf migrate (command-line utility)

Migrates config and show files built for prior versions of MPF to the current version.

mpf monitor (command-line utility)

Starts the *MPF Monitor*.

mpf hardware (command-line utility)

Starts the MPF game engine, scan/configure hardware platforms and exit.

Command line options

There are several command-line options you can use when running `mpf hardware`.

scan

Start MPF, scan all configured hardware platforms, dump their state and exit

firmware_update

Start MPF, scan all configured hardware platforms, update their firmware if not up to date and exit.

mpf service (command-line utility)

Start the command line service mode. Run this command from your machine folder. Service CLI will connect to MPF via BCP and put the machine into service mode.

Command line options

`mpf service` will spawn an interactive shell. See *Service Commandline* for details.

mpf build (command-line utility)

See *Build Command Line* for details.

mpf test (command-line utility)

See *Run Single File Tests*.

mpf format (command-line utility)

See *Format And Lint Config Files*.

How to change the TCP ports MPF uses

Note: The functionality for changing the BCP port in the MPF-MC was added in MPF-MC v0.32.10.

Various MPF components talk to each other via a TCP socket protocol called BCP (which we invented). By default, MPF and MPF-MC each listen for incoming BCP connections on the following two TCP ports:

- `5050` MPF-MC
- `5051` MPF

When MPF-MC starts up, it starts listening on port 5050. If the MPF game engine doesn't connect, MPF-MC will sit there and wait for it. No problem.

When the MPF game engine starts, it attempts to connect to the MC on port 5050. If it can't make a connection, it will try again, and keep trying until a connection is made. (Note that you can control the behavior of this in the config files.)

The MPF game engine also listens for incoming BCP connections on 5051. This is not used by MPF-MC, but is used by other things that need to connect to MPF, such as the MPF Monitor.

If you have a port conflict (because something else on your system is using port 5050 or 5051), then you can change the MPF and MPF-MC ports to whatever you want. Just add the following two sections to your machine-wide config file. Note that you have to change it in two places, the "bcp" section which is what the MPF game engine reads to know what port the MC is listening on, and the "mpf-mc" section which is what the MC reads to know what port it should listen on.

Valid port numbers are anything between 1024 and 65535.

```
# config_version=5

bcp:
  connections:
    local_display:
      port: 1234

mpf-mc:
  bcp_port: 1234
```

CHAPTER 5

MPF Tutorial

Let's learn by example!

This tutorial will walk you through using MPF to create a basic pinball machine config. Since MPF is just software that supports lots of different physical hardware, you don't actually need to have physical pinball machine hardware to complete the tutorial. You can create a "virtual" pinball machine for now and then hook up a real machine later.

The tutorial includes:

- Configuring switches, coils, flippers, sling shots, and your trough.
- Starting and playing a complete game with multiple players.
- Setting up attract mode light and display shows.
- Basic scoring and defining shots and lights.
- Using the display to show what's happening and the score.
- Setting up a "base" game mode.

The idea is that everyone should follow the tutorial, and complete every step, in order. (The tutorial steps all build off the previous steps.) Once that's done, you can then move browse through the rest of the documentation to read specific "How To" guides for everything else you need. (These are in the Control Systems / Hardware, Pinball Mechanisms, Game Logic, Displays & Graphics, Sound, and Shows sections.)

If you want to see us work through the tutorial you can also watch our walk-through video:

https://youtu.be/R4WugGNiezc

Now let's get started. . .

Tutorial step 1: Installing MPF on your computer

The first step to using MPF is to understand some basics about how it works and to actually get MPF installed on your computer. So that's what these next few steps will do.

1. You don't need a physical pinball machine yet

First, you do *not* need to have physical hardware to go through this tutorial. You can complete the entire thing via MPF's "virtual" hardware platform which lets you run MPF on your computer with no actual hardware attached.

We should point out that MPF's virtual platform is *not* pinball emulation software. There is no 3D-rendered playfield like Pinball Arcade, and you can't really "play" your game. (This is because MPF is not pinball emulation software, rather, it's software to control a real pinball machine!)

That said, MPF has tools which let you control switches and see lights flash on your computer screen, and you can arrange them onto an image of your playfield, so you can actually build a complete game in MPF before you ever start on a physical machine!

2. Read the overview of MPF

You certainly don't have to read through all the documentation to start this tutorial. However, the documentation is arranged in the order you should read it, so if you haven't read the stuff leading up to this point, please do that now. (If you're reading this online, start with the "*MPF Overview*" entry on the left. If you're reading a PDF, please turn to Page 1. :)

3. Install MPF

Obviously before you can start using MPF, you need to install it. At this point it's best to install MPF on whatever computer you use daily. There is certainly no need to try to put it on some small Linux single board computer or Raspberry Pi or whatever you ultimately plan to put in your pinball machine. For now just install it on your laptop.

MPF runs on Windows, Mac, and Linux. You can find the installation instructions *here*.

Go do that now, and make sure that you get both MPF and the MPF-MC installed.

3a. What if you already installed MPF?

If you already installed MPF, let's quickly make sure you're using the same version this tutorial is written for.

This tutorial is written for MPF versions 0.56.

To see what version you have, open a command prompt (like you did when you installed MPF) and run the following command:

```
mpf --version
```

That command should print something like `MPF v0.51.3`. Note that the version is three numbers, `x.y.z`. The last number (the "z") is the patch number and doesn't have any functional changes. (In other words, MPF 0.51.0 and 0.51.2 and 0.51.56 have the same functions and features.)

Tip: We highly recommend that you use the latest version of MPF, especially if you're starting out and don't have config files to migrate. To find out which version of MPF is the latest, visit the MPF Users Google Group and look at the banner welcome message at the top of the page.

If this command gives you an error, then go back to the *Downloading & Installing MPF (2023 Version)* section to make sure MPF is installed. If it prints a version number lower than 0.56, then install the latest version of MPF. And if it shows that you have a newer version of MPF (based on the first two numbers), then go to docs.missionpinball.org to get the version of this documentation that matches the version of MPF you have.

4. Let's go!

If you're reading this tutorial online, note that there are "Previous" and "Next" navigation buttons at the bottom of each page. You should be able to just click those to follow along. (And if you'd like to download a copy of this documentation to read offline, click the "Read the Docs" link at the bottom of the menu bar on the left for links to PDF, HTML, and Epub versions of this documentation.)

Tutorial step 2: Create your machine folder

Okay, so MPF is installed and you're able to run *Demo Man*. Great! Now it's time to create the folders and files for your own game.

1. Understand the "machine folder" concept

In MPF, we use the term *machine folder* to describe the folder that contains all the files for your game. This includes config files, images, sounds, videos, settings, audits, modes, and everything else.

The machine folder is organized into subfolders to keep everything straight.

MPF machine folders are portable, meaning you can grab a machine folder from one computer and run it on another—-even if it's a different platform. (Windows to Linux, Mac to Windows, etc.)

Note: In "MPF speak", we call these "machine" folders, not "game" folders. The reason is because in MPF, a "game" is an actual game-in-progress running on a machine (with players, balls, scores, etc.). So you're really creating a pinball *machine* config, not a pinball game config.

---

2. Create your machine folders

Okay, so let's get started with your own game's machine folder. The first step is to create an empty folder somewhere. (Anywhere you want.) You can name this folder whatever you want too.

Let's use the name "your_machine", and let's add that folder to the `C:\pinball` folder, like this:

```
C:\pinball\your_machine
```

Obviously if you're on Mac or Linux, you won't have a C: drive, but that doesn't matter for the tutorial. Just create a new folder empty folder somewhere and name it whatever you want.

Throughout this tutorial we'll refer to this as "your machine folder".

Next create a subfolder in your new machine folder called `\config`. This is where your machine configuration files will live. This folder should be inside your machine folder, like this:

```
C:\pinball\your_machine\config
```

3. Create your machine config file

Now let's actually create a file which will contain all the configuration for your pinball machine. To do that, create a file called `config.yaml` in your machine folder's */config* sub-folder. This file should be here:

```
C:\pinball\your_machine\config\config.yaml
```

Note that if you're on Windows and you just right-click and select *New > Text Document*, make sure that Windows Explorer is configured to show file extensions so you actually create a file called `config.yaml` and not `config.yaml.txt`. (That's in the "View" menu of Explorer.)

4. Add #config_version=5 to the top of your config file

The first thing you need to do when you create any new config file for MPF is to add an entry on the very top line that tells MPF what "version" of the MPF config spec you're using for the file you're creating.

So just open the file (with a text editor or a free tool like Atom, Sublime, or Notepad++) and then add that to the top of the file and save it. If you are familiar with an IDE such as VSCode or PyCharm/IntelliJ we suggest that you install the *MPF language server* which supports auto-completion, syntax and error highlighting, context help, go to definition and more.

You can also follow our video about the perfect IDE setup:

https://youtu.be/QdDHEe2aEJo

For MPF 0.56, that should look like this:

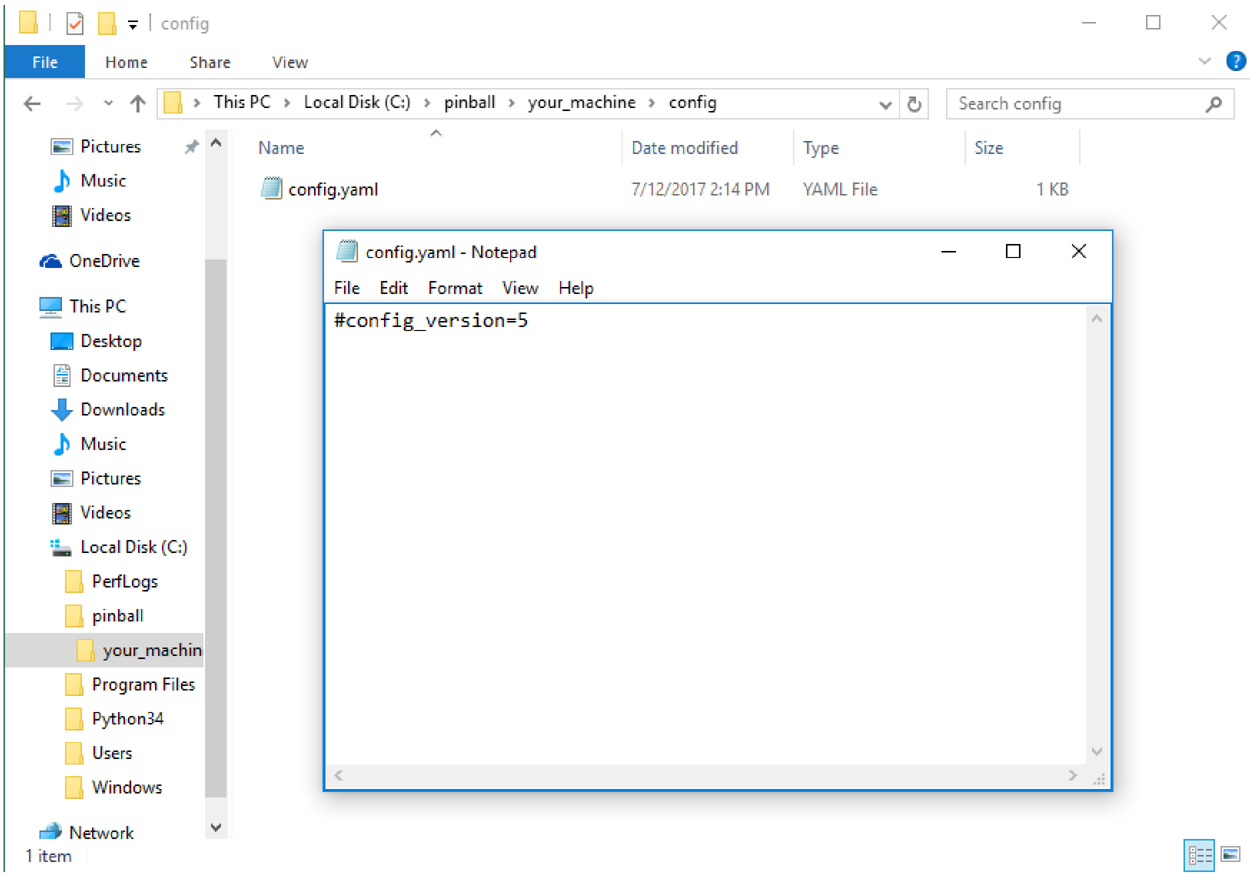
```
#config_version=5
```

Be sure to enter this exactly as it's shown here, with no spaces around the equal sign.

This line tells MPF which version of the config spec you have. That way if a future version of MPF requires changes to a config file, it can automatically recognize older files and update them.

The current version of the config files is 5 which is what's used with MPF 0.50 and newer, so that's what we're adding here.

At this point, your environment should look like this:



Note the folder structure, the location of the `config.yaml` file, and the `#config_version=5` as the only contents of that file.

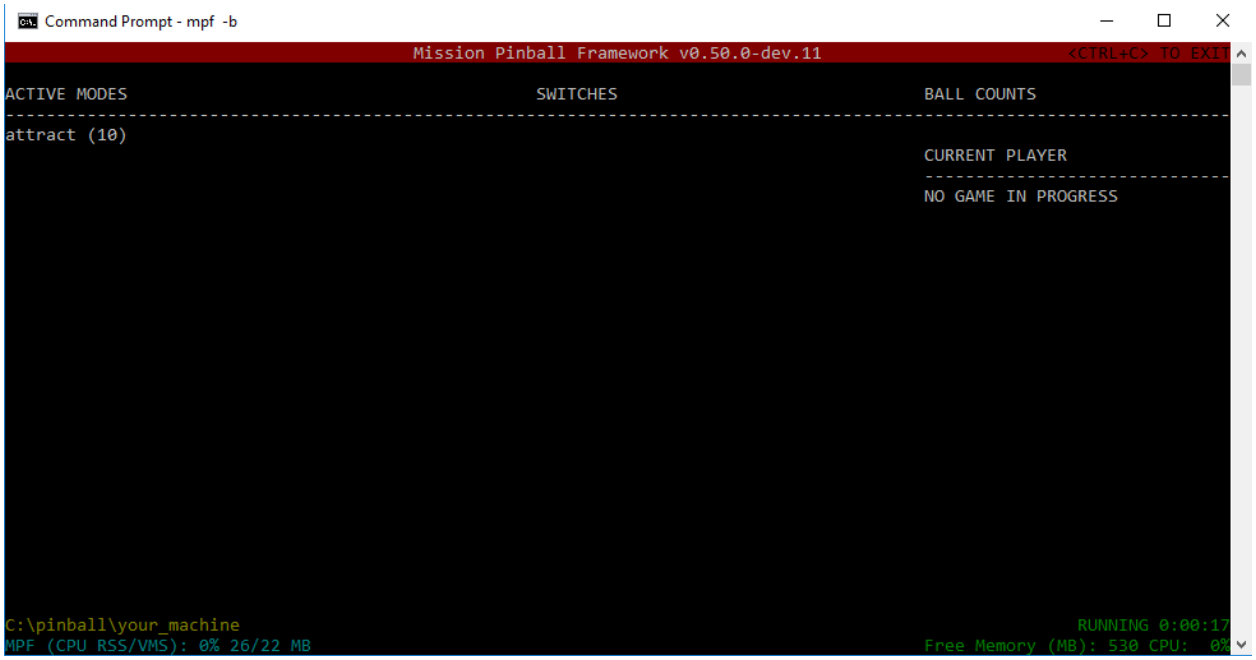
5. Run your game!

Believe it our not, it's time to run your game! Simply open a console window and change to your machine folder, and run `mpf -b`, like this:

```
C:\pinball\your_machine>mpf -b
```

Again, enter it as shown, with a space between `mpf` and `-b`. (The `-b` option tells MPF not to try to connect to a media controller for display and sound since we haven't set that up yet.)

You should get results that look something like this:



This is MPF's default display when it's running. Don't worry–this is not what your machine's players will see when they play! :) We'll set that up later. This is more for you while you're building your MPF config.

Notice a few things on this console display:

- The version of MPF that's running is in the red bar along the top.
- Any game modes that are running are in the "ACTIVE MODES" section (which is just the attract mode for now since we haven't set anything else up).
- A list of switches and their states in in the middle "SWITCHES" section (which is also empty since we don't have any switches setup yet).
- A list showing which devices are holding balls is in the "BALL COUNTS" section (also blank).
- The current player's number, score, and ball in the "CURRENT PLAYER" section (also blank).
- The machine folder path (in yellow in the lower left corner)
- How much CPU and memory MPF is using. (CPU is the percentage which is 0% in the screen shot which makes sense since your config is blank and MPF isn't doing anything!) The memory use is the memory used (RSS), then a slash, then the memory size (total that it could use). In the screen shot, we see MPF is using 4MB but could use as much as 22MB.
- How long MPF has been running (hours:minutes:seconds) in green in the lower right
- How much total free memory your computer has (530 MB in the screen shot)
- How much total CPU is busy on your computer overall (also 0% in the screen shot)

At this point you can pretty much just sit there and watch MPF forever, but it won't ever do anything until you add more to your config file.

To stop MPF, hit `CTRL+C` . That should take you back to the command window.

At this point you're all set! If your machine is working like this, go ahead and move on to the next step. However if you got something else on your display or some kind of error or crash, read on below. . .

What if it didn't work?

If you don't get an output that shows the attract mode running like the example above, there could be a few reasons for this, depending on the error.

If you get a crash with a message about a "Config file version mismatch", like this:

```
C:\pinball\your_machine>mpf -b
Config file version mismatch: C:\pinball\your_machine\config\config.yaml
Traceback (most recent call last):
  File "c:\python34\lib\site-packages\mpf\commands\game.py", line 202, in __init__
    MachineController(mpf_path, machine_path, vars(self.args)).run()
  File "c:\python34\lib\site-packages\mpf\core\machine.py", line 146, in __init__
    self._load_config()
  File "c:\python34\lib\site-packages\mpf\core\machine.py", line 405, in _load_config
    self._load_config_from_files()
  File "c:\python34\lib\site-packages\mpf\core\machine.py", line 425, in _load_config_from_files
    config_type='machine'))
  File "c:\python34\lib\site-packages\mpf\core\config_processor.py", line 24, in load_config_file
    config = FileManager.load(filename, verify_version, halt_on_error)
  File "c:\python34\lib\site-packages\mpf\core\file_manager.py", line 167, in load
    halt_on_error)
  File "c:\python34\lib\site-packages\mpf\file_interfaces\yaml_interface.py", line 255, in load
    raise ValueError("Config file version mismatch: {}".format(filename))
ValueError: Config file version mismatch: C:\pinball\your_machine\config\config.yaml
```

This means you don't have `#config_version=5` in the top line of your config file. (Make sure you include the hash mark as part of that.)

If the following line at the end of your log and nothing more happens you probably started mpf with mc (i.e. by omitting the *-b* switch). This can be fixed by either running *mpf -b* or by making sure that the media controller is running.

```
BCPClientSocket.local_display : Connecting BCP to 'local_display' at localhost:5050...
```

If you get an error that says `Could not find machine folder: 'None'`, that means that you ran MPF from the wrong folder. For example:

```
C:\pinball\your_machine\config>mpf
Error. Could not find machine folder: 'None'.
```

This happens because the command prompt is in the child "config" folder, rather than the base machine folder. So `cd ..` up one level and try again.

```
C:\>mpf
Error. Could not find machine folder: 'None'.
```

Again, same thing here. The example above is in the root of C: which is not a valid machine folder. (It is possible to run a machine from another folder via command line options which is why this error says it couldn't find the machine "None", but for now just know that you need to run MPF from the root of your machine folder.)

It's possible you might also get an error about "mpf" not being recognized. For example, on Windows:

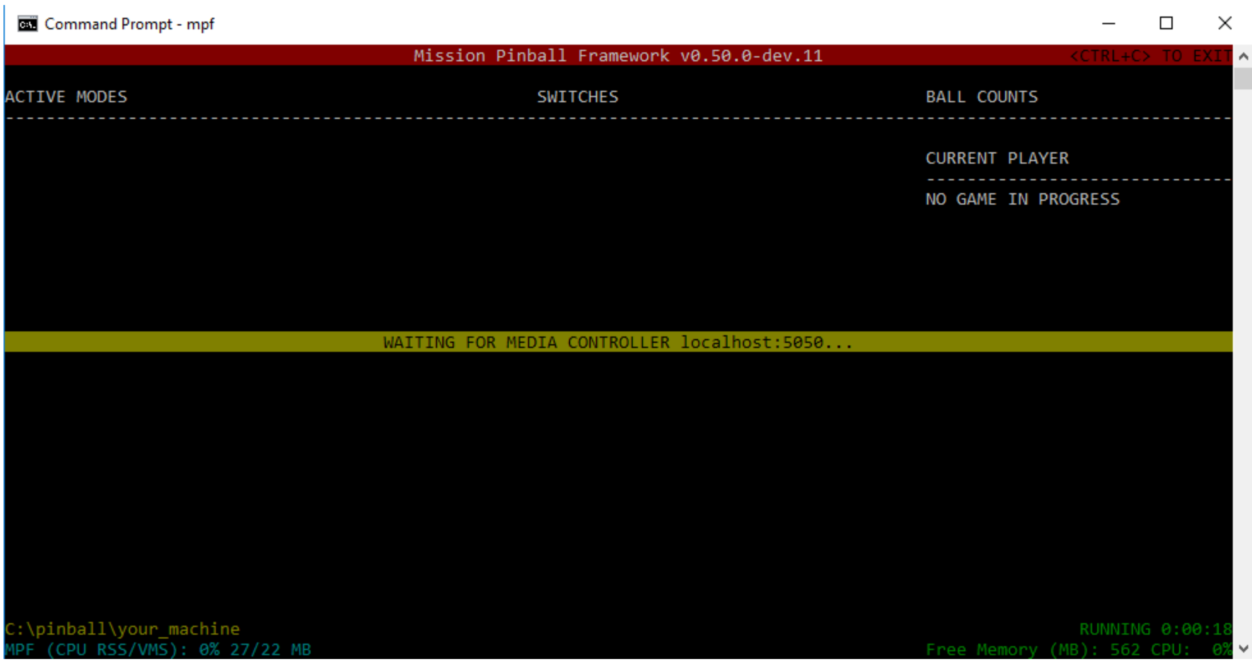
```
C:\pinball\your_machine>mpf
'mpf' is not recognized as an internal or external command,
operable program or batch file.
```

Or on Mac or Linux:

```
$ mpf
-bash: mpf: command not found
```

In this case you probably don't have MPF installed right, so jump back to the installation part of the docs and follow that again.

If you see a yellow bar and do not see the attract mode in the list of active modes, like this:



That means you did not including the `-b` option when you ran MPF. (e.g. you probably just ran `mpf` instead of `mpf -b`. In this case, MPF is trying to connect to the media controller (for your game's graphics and sounds), but since we haven't gotten that far in the tutorial, it doesn't exist and therefore MPF won't be able to connect to it.

Tutorial step 3: Get flipping!

There's something exciting about seeing the first flips from your own code, so in this step we're going to focus on getting your flippers working.

To do that, you have to add some entries to your config file to tell MPF about some coils and switches, then you have to group them together to tell MPF that they should act like flipper devices. So go ahead and open that `/config/config.yaml` file that you created in the previous step.

1. Add your flipper switches

The `switches:` section of your machine config file is where you list all the switches in your machine and map physical switch numbers to more friendly switch names. (This is what makes it possible to interact with switch names like "left_flipper" and "right_inlane" versus "switch 27" or "switch 19".)

So on the line after the `#config_version=5` entry from the previous tutorial step, write `switches:` (note the colon). Then on the next line, type four spaces (these must be spaces, not a tab), and write `s_left_flipper:`. Then on the next line, type eight spaces and add `number:`. Repeat that again for `s_right_flipper:`.

So now your `config.yaml` file should look like this:

```
#config_version=5

switches:
  s_left_flipper:
    number:
    tags: left_flipper
  s_right_flipper:
    number:
    tags: right_flipper
```

In case you're wondering why we preface each switch name with "s_", that's a little trick we learned that makes things easier as you get deeper into your configuration. We do this because most text editors and IDEs have "autocomplete" functions where it will pop up a list to autocomplete values as you type. So if you preface all your switches with "s_" (and your coils with "c_", your lights with "l_", etc.), then as soon as you type "s_" into your YAML file you should get a popup list with all your switches which you can use to select the right one. These saves lots of headaches later caused by not entering the name exactly right somewhere. :)

If you use Sublime as your editor, it just does this automatically. Other editors might require plugins. (For example, you can add this functionality to Atom with a free package called "autocomplete-plus".)

Notice that we added tags called `left_flipper` and `right_flipper`. These are optional, but recommended. The reason is that MPF includes a *combo switch* feature which posts events when player switches are held in combination. If you add these tags to your flipper switches, an event called *flipper_cancel* will be posted when the player hits both flipper buttons at the same time which you can use to cancel shows and other things you want the player to be able to skip.

When naming your switches (and most devices in MPF), your name can't start with a number and it should only be a combination of letters, numbers, and underscores.

Also, the names you enter here are the internal names that you'll use for these switches in your game code and configuration file. When it comes time to create "friendly" names for these switches which you'll expose via the service menu, you can create plain-English labels with spaces and capitalization everything. But that comes later.

In pre-0.50 versions, MPF was not case-sensitive and would internally convert most things to lowercase before comparison. This proved to be problematic, so MPF is now case-sensitive for all elements of your config files. Our configuration directives use only lowercase letters, underscores, and numbers. While you are free to format your tags as you wish, be aware that case-consistency is now required.

Speaking of formatting files, let's look at a few important things to know about YAML files (which is the format of the file we're creating here):

- You cannot use tabs to indent in YAML. (It is literally not allowed.) Most text editors can be configured to automatically insert spaces when you push the tab key, or you can just hit the space bar a bunch of times.
- The exact number of spaces you use for the indents doesn't matter (most people use groups of two or four), but what is absolutely important is that all items at the same "level" must be indented with the same number of spaces. In other words `s_left_flipper:` and `s_right_flipper:` need to have the same number of spaces in front of them. In a practical sense this shouldn't be a problem, because again most text editors let you use the tab key to automatically insert space characters.
- You cannot have a space between the setting name and the colon. GOOD: `switches:`. BAD: `switches :`
- You must must have a space after the colon and the setting value. GOOD: `balls: 3`. BAD: `balls:3`
- Anything on a line following a hash sign # is ignored, so you can use this to add comments and notes to yourself.

This all might seem kind of annoying, but that's just the way it is with YAML files. When we started building MPF, we weighed the pros and cons of lots of different config file formats (XML, INI, JSON, TOML, text, Python, etc.), and YAML was the best trade-off in terms of having the features we needed while being the easiest to use.

By the way, at some point we'll create GUI tools you can use to build your configs instead of having to hand-edit YAML files, but that's probably a few years away, so in the meantime, get used to YAML. :)

2. Enter the hardware numbers for your switches

The `config.yaml` file you have so far is completely valid. However, you'll notice that the `number:` setting for each switch is blank. If you are not using MPF with a physical pinball machine yet, you can keep these numbers blank. But if you want to control a real pinball machine, you need to enter values for each switch's `number:` setting.

The exact number you enter for each switch is dictated by which switch input on your pinball controller each switch is connected to. However, different controllers use different number formats.

The *How to configure "number:" settings* guide explains how hardware numbering works on each of the various hardware platforms MPF supports, so check that out now and enter your real numbers, not the made-up ones we use below.

```
switches:
  s_left_flipper:
    number: 0  # this can be blank if you don't have physical hw yet
  s_right_flipper:
    number: 1  # if you do have physical hw, most likely your number will be different
```

3. Add your flipper coils

Next you need to add entries for your flipper coils. These will be added to a section called `coils:`. If you're using dual-wound coils, you'll actually have four coil entries here—-both the main and hold coils for each flipper. If you're using single-wound coils, then you'll only have one coil for each flipper (which we'll configure to pulse-width modulation for the holds).

If you have no idea what we're talking about, read our *Flippers* documentation for an introduction to flipper concepts, dual-wound versus single- wound, holding techniques, end-of-stroke switches, and a bunch of other stuff that's important that you probably never thought about.

Here's an example of how you'd enter your coils for a machine with two dual-wound coils. If you have single-wound coils, or you have more than two flippers, refer to the *Flippers* documentation for examples of how to configure them.

```
coils:
  c_flipper_left_main:
    number: 0  # again, these numbers will probably be different for you
  c_flipper_left_hold:
    number: 1  # check your platform-specific documentation for the actual numbers
    allow_enable: true
  c_flipper_right_main:
    number: 2
  c_flipper_right_hold:
    number: 3
    allow_enable: true
```

Again, note each coil name is indented four spaces, and each "number" listed under them is indented eight spaces, there's no space before the colons, and there is a space after the colons. Like the switch numbers, the `number:` entry under each coil is the number that the pinball hardware controller uses for this coil. The exact number will depend on what type of controller hardware and driver boards you're using.

Also note that the two hold coils have `allow_enable:` entries added, with values of "true". Anyway, the purpose of the `allow_enable:` setting is that as a safety precaution, MPF does not allow you to enable (that is, to hold a coil in its "on" position) unless you specifically add `allow_enable: true` to that coil's config. This will help to prevent some errant config from enabling a coil that you didn't mean to enable and burning it up or starting a fire.

So in the case of your flippers, the "hold" coil of a flipper needs to have `allow_enable: true` since in order for it to act as a flipper, that coil need to be allowed to be enabled (held on).

4. Add your flipper "devices"

Okay, you have your coils and switches defined, but you can't flip yet because you don't have any flippers defined. Now you might be thinking, "Wait, but didn't I just configure the coils and switches?" Yes, you did, but now you have to tell MPF that you want to create a flipper device which links together one switch and one (or two) coils to become a "flipper". MPF supports dozens of different types of *Pinball Mechanisms*, some of which (like flippers), are created by combining other devices.

You create your flipper devices by adding a `flippers:` section to your config file, and then specifying the switch and coil(s) for each flipper. Since the flippers belong to a playfield we also create this now. Here's what you would create based on the switches and coils we've defined so far:

```
playfields:
  playfield:
    tags: default
    default_source_device: None  # use None in steps before 8

flippers:
  left_flipper:
```

(continues on next page)

(continued from previous page)

```
    main_coil: c_flipper_left_main
    hold_coil: c_flipper_left_hold
    activation_switch: s_left_flipper
  right_flipper:
    main_coil: c_flipper_right_main
    hold_coil: c_flipper_right_hold
    activation_switch: s_right_flipper
```

5. Try running MPF to make sure your config file is ok

At this point you should run your game to make sure it runs okay. Your flippers aren't going to work yet, but mainly we want to make sure MPF can read your config files and that there aren't any errors. Open a command prompt, switch to your machine folder, and run MPF again (like Step 2), also with the -b option. Additionally, we will add the -t option to disable the text UI and show the log on the console instead (you can also see it inside the logs folder inside your machine):

```
$ mpf -t -b
```

The console output will look similar to Step 2 as well, and it won't look like much is happening here. The main thing is to make sure that MPF starts and runs without giving you any errors–meaning that everything you setup in your config file is ok.

```
$ mpf -t -b
INFO : root : Loading config.
INFO : YamlMultifileConfigLoader : Machine config file #1: config.yaml
INFO : ConfigProcessor : Loading config from cache: /tmp/7146c817793475fbeb8d22f907d7bbbc.mpf_cache
INFO : ConfigProcessor : Loading config from cache: /tmp/49091ea856e626b51c4160f53a2ef744.mpf_cache
INFO : ConfigProcessor : Loading config from cache: /tmp/4cc7d3d11df84bb81fda7943558aba56.mpf_cache
INFO : Machine : Mission Pinball Framework Core Engine v0.54.0-dev.18
INFO : Machine : Command line arguments: {'no_load_cache': False, 'create_config_cache': True, 'bcp':␣
↪False, 'configfile': ['config.yaml'], 'force_assets_load': False, 'jsonlogging': False, 'logfile':
↪'logs/2020-04-01-21-45-55-mpf.log', 'pause': False, 'production': False, 'text_ui': False, 'loglevel
↪': 15, 'consoleloglevel': 20, 'force_platform': None, 'syslog_address': None, 'mc_file_name': None,
↪'no_sound': False}
INFO : Machine : MPF path: /pinball/src/mpf/mpf
INFO : Machine : Machine path: /mpf-examples/tutorial/step_3
INFO : Machine : Platform: linux
INFO : Machine : Python executable location: /usr/bin/python3
INFO : Machine : Python version: 3.6.9 (64-bit)
INFO : Machine : Initialise MPF.
INFO : EventManager : Event: ======'machine_var_credits_string'====== Args={'value': 'FREE PLAY', 'prev_
↪value': None, 'change': True}
INFO : EventManager : Event: ======'machine_var_mpf_version'====== Args={'value': 'MPF v0.54.0-dev.18',
↪'prev_value': None, 'change': True}
INFO : EventManager : Event: ======'machine_var_mpf_extended_version'====== Args={'value': 'MPF v0.54.0-
↪dev.18, Config version:5, Show version: 5, BCP version:1.1', 'prev_value': None, 'change': True}
INFO : EventManager : Event: ======'machine_var_python_version'====== Args={'value': '3.6.9', 'prev_
↪value': None, 'change': True}
INFO : EventManager : Event: ======'machine_var_platform'====== Args={'value': 'Linux-4.15.0-72-generic-
↪x86_64-with-Ubuntu-18.04-bionic', 'prev_value': None, 'change': True}
INFO : EventManager : Event: ======'machine_var_platform_system'====== Args={'value': 'Linux', 'prev_
↪value': None, 'change': True}
```

(continues on next page)

(continued from previous page)

```
INFO : EventManager : Event: ======'machine_var_platform_release'====== Args={'value': '4.15.0-72-
↪generic', 'prev_value': None, 'change': True}
INFO : EventManager : Event: ======'machine_var_platform_version'====== Args={'value': '#81-Ubuntu SMP␣
↪Tue Nov 26 12:20:02 UTC 2019', 'prev_value': None, 'change': True}
INFO : EventManager : Event: ======'machine_var_platform_machine'====== Args={'value': 'x86_64', 'prev_
↪value': None, 'change': True}
INFO : EventManager : Event: ======'init_phase_1'====== Args={}
INFO : EventManager : Event: ======'init_phase_2'====== Args={}
INFO : EventManager : Event: ======'init_phase_3'====== Args={}
INFO : EventManager : Event: ======'init_phase_4'====== Args={}
INFO : EventManager : Event: ======'machine_var_audits_switches_s_left_flipper'====== Args={'value': 0,
↪'prev_value': None, 'change': True}
INFO : EventManager : Event: ======'machine_var_audits_switches_s_right_flipper'====== Args={'value': 0,
↪ 'prev_value': None, 'change': True}
INFO : EventManager : Event: ======'init_phase_5'====== Args={}
INFO : EventManager : Event: ======'init_done'====== Args={}
INFO : EventManager : Event: ======'machine_reset_phase_1'====== Args={}
INFO : EventManager : Event: ======'machine_reset_phase_2'====== Args={}
INFO : EventManager : Event: ======'machine_reset_phase_3'====== Args={}
INFO : EventManager : Event: ======'reset_complete'====== Args={}
INFO : EventManager : Event: ======'mode_attract_will_start'====== Args={}
INFO : EventManager : Event: ======'mode_attract_starting'====== Args={}
INFO : Mode.attract : Started. Priority: 10
INFO : EventManager : Event: ======'mode_attract_started'====== Args={}
INFO : EventManager : Event: ======'collecting_balls_complete'====== Args={}
INFO : Machine : Starting the main run loop.
```

At this point you can stop it by making sure your console window has focus and then hitting `CTRL+C`.

What if it didn't work?

If your game ran fine, then you can skip down to Step 6 below. If something didn't work or you got an error, then there are a few things to try depending on what your error was.

If the last line in your console output was something like this:

```
ValueError: Found a "switchs:" section in config file C:\your_machine\config\config, but that section␣
↪is not valid in machine config files.
```

That means that it found a section in your config file that is not valid. Most likely this is due to a typo. For example, the above example has "switchs" instead of "switches".

Or maybe the error is more like this:

```
AssertionError: Config validation error: Entry flippers:left_flipper:main_coil:c_fliper_left_main is␣
↪not valid.
```

This is showing that the `flippers:left_flipper:main_coil:c_fliper_left_main` entry is not valid. Again this is a typo–the coil name is spelled wrong (one "p" in flipper instead of two).

Or something like this:

```
AssertionError: Your config contains a value for the setting "flippers:left_flipper:holdcoil", but this␣
↪is not a valid setting name.
```

Again pretty self-explanatory. The setting `flippers:left_flipper:holdcoil` is not valid. (It should actually be "hold_coil", not "holdcoil".)

So you can see that we've tried to be pretty helpful when it comes to typos and config file errors. The trick it just to read through the output in the logs and to trace down what they're complaining about.

You might also get errors saying there's some kind of YAML problem. For example, if you remove the colon after the `coils:` section and re-run MPF, you get the following error:

```
ValueError: YAML error found in file /Users/brian/git/mpf-examples/tutorial/config/config.yaml. Line 16,
↪ Position 24
```

Line 16, Position 24. Pretty straightforward, except the missing colon is actually on line 15. This is because removing the colon still produced valid YAML until it hit the next line. The point is that if you get a YAML error, look a few lines above and below the line number from the error.

Again, recapping the rules of YAML:

- Be sure to indent with spaces, not tabs.
- Make sure that all the "child" elements are indented the same. So your `s_left_flipper` and `s_right_flipper` both need to be indented the same number of spaces, etc.
- Make sure you *do not* have a space *before* each colon.
- Make sure you *do* have a space *after* each colon.
- Make sure you have the `#config_version=5` as the first line in your file.

If you struggle to spot the problem read our *Debugging YAML Parse Errors* guide.

6. Enabling your flippers

Just running MPF with your game's config file isn't enough to get your flippers working. By default, they are only turned on when a ball starts, and they automatically turn off when a ball ends. But the simple config file we just created doesn't have a start button or your ball trough or plunger lane configured, so you can't actually start a game yet. So in order to get your flippers working, we need to add a configuration into each flipper's entry in your config file that tells MPF that we just want to enable your flippers right away, without an actual game. (This is just a temporary setting that we'll remove later.) To do this, add the following entry to each of your flippers in your config file:

```
enable_events: machine_reset_phase_3
```

We'll cover exactly what this means later on. (Basically it's telling each of your flippers that they should enable themselves when MPF is booting up, rather than them waiting for a ball to start.) So now the `flippers:` section of your config file should look like this: (If you have single-wound coils, then you won't have the `hold_coil:` entries here.)

```
flippers:
  left_flipper:
    main_coil: c_flipper_left_main
    hold_coil: c_flipper_left_hold
    activation_switch: s_left_flipper
    enable_events: machine_reset_phase_3
  right_flipper:
    main_coil: c_flipper_right_main
```

(continues on next page)

(continued from previous page)

```
    hold_coil: c_flipper_right_hold
    activation_switch: s_right_flipper
    enable_events: machine_reset_phase_3
```

At this point the rest of the steps on this page are for getting your physical machine connected to your pinball controller. If you don't have a physical machine yet then you can skip directly to *Tutorial step 4: Adjust your flipper power*.

7. Configure MPF to use your physical pinball controller

If you have a physical pinball machine (or at least a something on your workbench) which is hooked up to a FAST, P-ROC, P3-ROC, OPP, or Stern SPIKE controller, then you need to add the hardware information to your config file so MPF knows which platform interface to use and how to talk to your hardware. To configure MPF to use a hardware pinball controller, you need to add a `hardware:` section to your config file, and then you add settings for `platform:` and `driverboards:`.

Don't worry if you do not have any hardware yet. You can run through the tutorial without hardware by using the *virtual* hardware platform.

Remember earlier in this step, we provided links to the documentation for each platform. Here is *a list of supported hardware platforms* and how to set them up.

You only need look at those docs for the specifics parts of the config that vary depending on your hardware. The good news is that 99.9% of the MPF config files are identical regardless of the hardware you're using.

Here are some various examples of different types of hardware configs. Please understand that these are just some examples! Do not copy them for your own use, rather, follow the instructions from the link to the list above.

FAST Pinball with FAST IO driver boards:

```
hardware:
  platform: fast
  driverboards: fast

fast:
  ports: com4, com5

switches:
  s_left_flipper:
    number: 00
```

P-ROC installed in an existing WPC machine:

```
hardware:
  platform: p_roc
  driverboards: wpc

switches:
  s_left_flipper:
    number: SF2
```

P3-ROC with P-ROC driver & switch boards:

```
hardware:
  platform: p3_roc
  driverboards: pdb

switches:
  s_left_flipper:
    number: 0-0
```

In case you are using *the Virtual Pinball (VPX) Platform* the config file will look like:

```
hardware:
  platform: virtual_pinball

switches:
  s_sling:
    number: 0
  s_flipper:
    number: 3

coils:
  c_sling:
    number: 0
  c_flipper:
    number: 1
    allow_enable: true
```

Video about developing your game without hardware:

https://youtu.be/7XmIIhzEREk

See? They're all different.

7a. Understand the "virtual" hardware

If you just added a `platform:` setting to your config file which specifies a physical hardware platform, now every time you run MPF with that config, it will try to connect to the physical hardware. But what happens if you want to use MPF without your physical pinball hardware attached? In that case, you can run MPF with either the `-x` or `-X` command line options. (Lowercase "x" is the "virtual" platform, and uppercase "X" is the "smart virtual" platform.)

We'll talk more about those later. The point is that if you have configured your machine for physical hardware and then you want to run MPF without the physical hardware, you need to add either `-x` or `-X` to your `mpf` command when you run it.

8. One last check before powering up

Okay, now we're really close to flipping. Before you proceed take a look at your config file to make sure everything looks good. It should look something like this one, though of course that will depend on what platform you're using, whether you have dual-wound or single- wound flipper coils, and what type of driver boards you have (which will affect your coil and switch numbers). But here's the general idea. (This is the exact file we use with a FAST WPC controller plugged into an existing *Demolition Man* machine.)

```
#config_version=5

hardware:
  platform: fast
  driverboards: wpc

switches:
  s_left_flipper:
    number: SF4
  s_right_flipper:
    number: SF6

coils:
  c_flipper_left_main:
    number: FLLM
  c_flipper_left_hold:
    number: FLLH
    allow_enable: true
  c_flipper_right_main:
    number: FLRM
  c_flipper_right_hold:
    number: FLRH
    allow_enable: true

playfields:
  playfield:
    tags: default
    default_source_device: None  # use None in steps before 8

flippers:
  left_flipper:
    main_coil: c_flipper_left_main
    hold_coil: c_flipper_left_hold
    activation_switch: s_left_flipper
    enable_events: machine_reset_phase_3
  right_flipper:
    main_coil: c_flipper_right_main
    hold_coil: c_flipper_right_hold
    activation_switch: s_right_flipper
    enable_events: machine_reset_phase_3
```

Note that the individual sections of the config file can be in any order. We put the `hardware:` section at the top, but that's just our personal taste. It really makes no difference.

9. Running your game and flipping!

At this point you're ready to run your game, and you should be able to flip your flippers! Run your game with the following command:

```
C:\your_machine\mpf -t -b
```

Watch the console log for the entry about the attract mode starting. Once you see that then you should be able to hit your flipper buttons and they should flip as expected! You might notice that your flippers seem weak. That's okay. The default flipper power settings are weak just to be safe. We'll show you how to adjust your flipper power settings in the next step of this tutorial. You'll also notice that switch events are posted to the console. `State:1` means the switch flipped from inactive to active, and `State:0` means it flipped from active to inactive.

```
INFO : SwitchController : <<<<< switch: s_left_flipper, State:1 >>>>>
INFO : SwitchController : <<<<< switch: s_left_flipper, State:0 >>>>>
INFO : SwitchController : <<<<< switch: s_right_flipper, State:1 >>>>>
INFO : SwitchController : <<<<< switch: s_right_flipper, State:0 >>>>>
```

Here's a companion video which shows running your game at this point in the tutorial based on the config file above: (Note that this companion video is showing *Judge Dredd*, and it's based on an older version of MPF, but the basic concepts are the same.)

https://youtu.be/SkxZxkHHmXw

What if it doesn't work?

If your game doesn't flip while you're running this config, there are a few things it could be: If the game software runs but you don't have any flipping, check the following:

- Make sure you're *not* using the `-x` or `-X` command line options, since those tells MPF to run in with the "virtual" hardware (e.g. software-only) mode meaning it won't talk to your actual physical hardware.
- Verify that your switch and coil numbers are set properly. Remember the values of "0" and "1" and stuff that we used here are just for the sake of this tutorial. In real life your coil numbers are going to be something like `A8` or `FLLH` or `C15` or `A1-B0-7`, and your switches will be something more like `E5` or `0/4` or `SD12`. Again look the how to guides for your specific platform for details on how their numbers should be set.
- Make sure you added `enable_events: machine_reset_phase_3` to each of your flipper configurations.
- Make sure your coin door is closed! If you're running MPF on an existing Williams or Stern machine, remember that when the coin door is open, there's a switch that cuts off the power to the coils. (Ask us how we knew to add this to the list. :)
- It's possible that your flippers are working, but their power level is so low that they're not actually moving. (In this case you might hear them click when you hit the flipper button.) In this case you can move on to the next step in the tutorial where we adjust the flipper power.

If MPF crashes or gives an error:

- If you're using a P-ROC and you get a bunch of really fast messages about *Error opening P-ROC device* and *Failed, trying again. . .* , this is because (1) your pinball machine is not turned on, (2) your P-ROC is not connected to your computer (via USB), or (3) you have a problem with the P-ROC drivers. If you're running MPF in a virtual machine, make sure the USB connection is set to go to the VM.
- If you're using FAST or OPP hardware and you get an error about a port configuration, or not being able to open a port, then make sure your port numbers are correct. If you were previously connecting to one of those ports via a terminal emulator, make sure you've disconnected from the port in that software before running MPF.

If a flipper gets stuck on:

- Really this shouldn't happen. :) But it did on our machine just now and we really really confused. :) It turns out it was our flipper button which was stuck in the "on" position. The *Judge Dredd* machine we were using at the time had those aftermarket magnetic sensor buttons with the little magnets on the button flags, one of them came unglued and slipped out of alignment, making the switch stuck in the "on" position.

If you're still running into trouble, feel free to post to the mpf-users Google group. We'll incorporate your issues into this tutorial to make it easier for everyone in the future!

If you get YAML errors either copy the complete example below or read our *Debugging YAML Parse Errors* guide.

Check out the complete config.yaml file so far

If you want to see a complete `config.yaml` file up to this point, there's a "tutorial" machine in the mpf-examples repo that you downloaded in Step 1. (This is the same repo that contains the Demo Man game that you ran in Step 1.)

The complete machine config is in the `mpf-examples/tutorial/step_3` folder.

```
C:\mpf-examples\tutorial>mpf -t
```

Tutorial step 4: Adjust your flipper power

We casually mentioned in the previous step that MPF uses a very low default power setting for coils–mainly because we don't want to risk blowing apart some 40-year-old coil mechanism with a power setting that's too high. (Ask us how we know this! :)

So at this step in the tutorial, we're going to look at how you can adjust and fine-tune the power of your flipper coils. The good news is that everything you learn here will 100% apply to all the other coils in your machine (slingshots, pop bumpers, ball ejects, the knocker, drop target resets, etc.)

1. Adjust coil pulse times

Modern pinball controllers that MPF uses have the ability to precisely control how long (in milliseconds) the full power is applied to a coil. (Longer time = more power.) This is called the "pulse time" of a coil, as it controls how long the coil is pulsed when it's fired.

You can set the default pulse time for each coil in the coil's entry in the `coils:` section of your config file. If you don't specify a time for a particular coil, then MPF will use a default pulse time of 10ms.

So in the last step, we got your flipper coils working, but as they are now, they each use 10ms for their pulse times. (Remember for flippers we're talking about the strong initial pulse to move the flipper from the down to up position. Then after that pulse is over, if you have dual-wound coils, the main winding is shut off while the hold winding stays on, and if you have single wound coils the pulse time specifies how long the coil is on solid for before it goes to the on/off pwm switching.)

So right now your flippers have a pulse time of 10ms. But what if that's too strong? In that case you risk breaking something. Or if your coil is too weak, then your ball will be too slow or not be able to make it to the top of the playfield or up all your ramps. So now you have to play with different settings to see what "feels" right.

Unfortunately there's no universal pulse time setting that will work on every machine. It depends on how many windings your coils have, how worn out your coils are, how clean your coil sleeves are, how tight your flipper bats are to the playfield, how free-moving your linkages are, and how much voltage you're using. Some machines have coil pulse times set really low, like 12 or 14ms. Others might be 60 or 70ms. Our 1974 Big Shot machine has several coils with pulse times over 100ms. It all really depends.

You adjust the pulse time for each coil by adding a `default_pulse_ms:` setting to the coil's entry in the `coils:` section of your config file. (Notice that you make this change in the `coils:` section of your config, not the `flippers:` section.) So let's try changing your flipper coils from the default of 10ms to 20ms. Change your config file so it looks like this:

```
coils:
  c_flipper_left_main:
    number: 00
    default_pulse_ms: 20
  c_flipper_left_hold:
    number: 01
    allow_enable: true
  c_flipper_right_main:
    number: 02
    default_pulse_ms: 20
  c_flipper_right_hold:
    number: 03
    allow_enable: true
```

Notice that we only added `default_pulse_ms:` entries to the two main coils, since the hold coils are never pulsed so it doesn't matter what their pulse times are. Now play your game and see how it feels. Then keep on adjusting the `default_pulse_ms:` values up or down until your flippers feel right. In the future we'll create a coil test tool that makes it easy to dial-in your settings without having to manually change the config file and re-run your game, but we don't have that yet. You might find that you have to adjust this `default_pulse_ms:` setting down the road too. If you have a blank playfield then you might think that your coils are fine where they are, but once you add some ramps you might realize it's too hard to make a ramp shot and you have to increase the power a bit. Later on when you have a real game, you can even expose these pulse settings to operators via the service menu.

2. Adjusting coil "hold" strength

If you're using single-wound flipper coils, you should also take a look at the `default_hold_power:` values. (Again, to be clear, you only have to do this if your flippers have a single winding. If you have dual-wound coils then the hold winding is designed to be held on for long periods of time so you can safely keep it on full strength solid and you can skip to the next step.)

We don't have any good guidance for what your `default_hold_power:` values should be. Really you can just start with a value of 0.125 or 0.25 and then keep increasing it (0.0 for 0% power to 1.0 for 100% power) until your flipper holds are strong enough not to break their hold when a ball hits them. Some hardware platform have additional options for fine-turning the hold power if this setting result in weird buzzing sounds or don't feel right. See the *coils:* section of each hardware platform's How To guide for details for your platform.

By the way there are a lot of other settings you can configure for your flippers. (As detailed in the *flippers:* section of the config file reference.) They're not too important now, but we wanted to at least look at the power settings to make sure you don't get too far into this tutorial with a risk of burning them up.

3. Check out the complete config.yaml file so far

If you want to see a complete `config.yaml` file up to this point, it's available in the "tutorials" folder of the mpf-examples package that you should have downloaded in Step 1 of this tutorial.

There are config files for each step, so the config for Step 4 should be at `/mpf-examples/tutorial/config/step_4`.

You can run this file directly by switching to that folder and then running the following command:

```
C:\mpf-examples\tutorial>mpf
```

Tutorial step 5: Add a display

In this step, we're going to add a graphical on-screen window which will help show what's happening in your machine as it runs. If you're planning to put an LCD display in the backbox of your machine, this is what we'll set up now. And if you want to use a physical DMD (whether it's an older-style mono DMD, or a newer full color LED-based DMD), you'll be able to use the screen window we set up in this step to show a software version of your DMD. If you are looking to set up a Segment or Alphanumeric display, follow this guide to set up them up: *Alpha-Numeric / Segment Displays*

Regardless of what type of display you want to use in your final machine, follow this step in the tutorial and then you can set up your final display later.

1. Run the media controller to see how it works

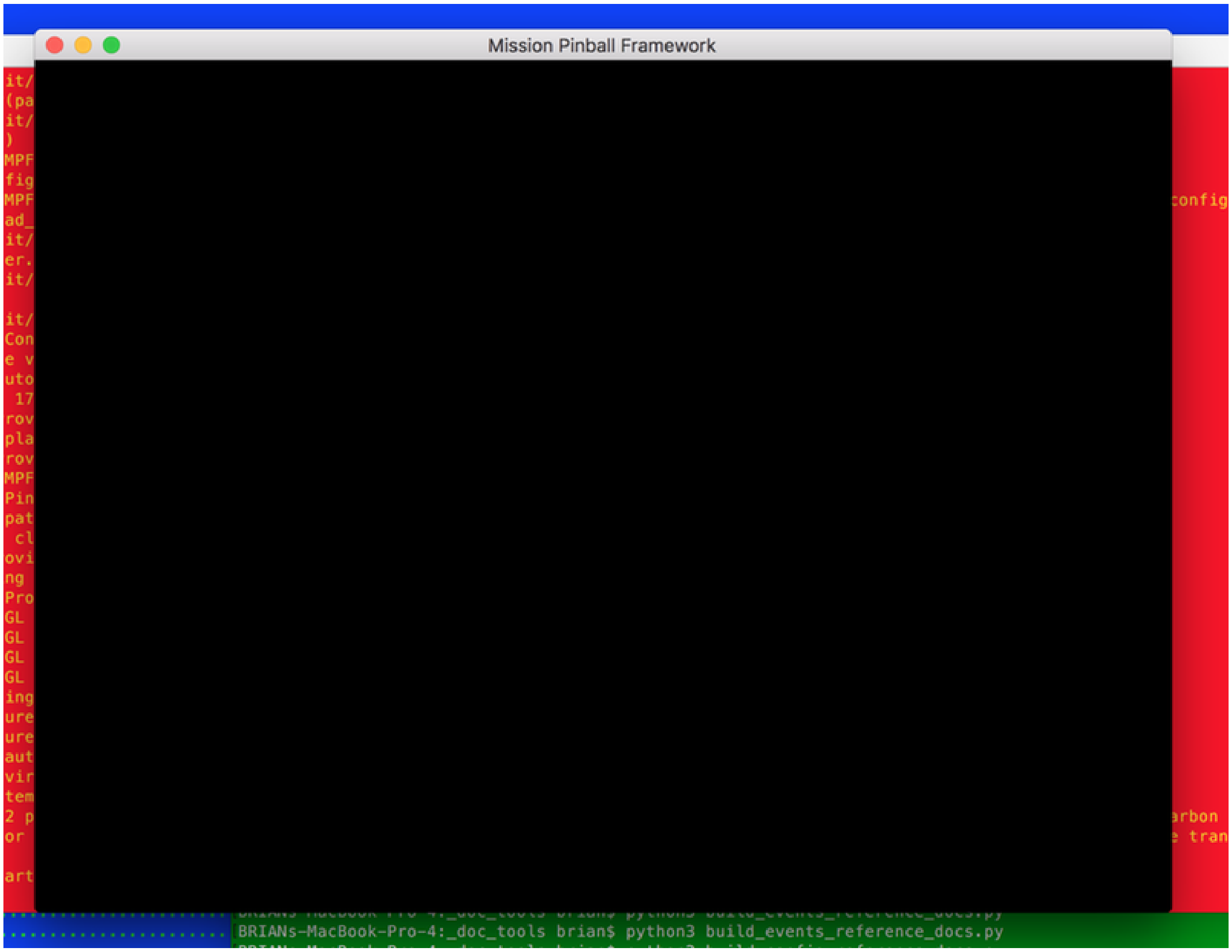
Remember from the MPF *MPF Overview* section (you read that, right?) that MPF is actually two separate pieces–the *game engine* and the *media controller*.

Up until this point in the tutorial, we've been running the MPF game engine only. In this step, since we're adding a display, we'll be working with the media controller.

So first, let's run the MPF media controller from your machine folder so you can see how it works. You do that with the `mpf mc` command, like this:

```
C:\pinball\your_machine>mpf mc
```

When you do this, you should see an 800x600 popup window that's completely black with the title "Mission Pinball Framework". Here's an example from Mac OS X:



You can close this window (and exit the MPF MC) by hitting the `Esc` key. (If this doesn't work, click your mouse in the popup window to give it focus and try again.)

You can also exit the MPF MC and close the popup window from the command line via `CTRL+C`.

2. Add a "display" to your config file

Now that you know how to run the MPF media controller (or "MPF-MC", as we often call it), let's configure your machine config so that window actually shows some content.

The MPF game engine and MPF-MC both read the same configuration files, so we'll be editing the same `config.yaml` file we've been working with all along.

The first step is to create a "display" in your MPF config, which is like an internal representation of a blank canvas that holds graphical content which can be shown on an LCD screen or a DMD. The MPF-MC can have multiple display canvases at the same time, and you can map different ones to different physical displays. (This means ultimately you can support multiple displays at the same time.)

The only setting for each display we need to worry about now is the height and width, both defined in terms of the number of pixels. So for now, create a single display called "window" set to 800x600 pixels. To do this, add the following to your `config.yaml` file:

```
displays:
  window:
    width: 800
    height: 600
```

Make sure that the word `displays:` has no spaces in front of it, since it's a top-level config item.

Note that in the example above, we used 2 spaces for the indentation instead of 4. That's fine, YAML doesn't care. (And you can even mix-and-match in the same file.) The only spacing thing that matters is items at the same level are indented the same number of spaces (like "width" and "height"). Also, no tabs.

The configuration above is creating a display called "window" which MPF will automatically map to the on screen popup window. There are more options here (especially when you get to using multiple displays) covered in the *Displays, DMDs, & Graphics* section of the documentation, but we don't need to worry about that.

Also, again, if your machine is going to use a physical DMD (whether mono or color), or if you want to have the "dot look" of an on-screen DMD on an LCD screen, for now just follow along the tutorial as is, and then you can read full display documentation afterwards to configure your displays. Everything we do in the tutorial will transfer over even if you ultimately use a different kind of display.

3. Add a slide & a text widget

Ok, so now we have a display called "window". If you run `mpf mc`, you will still see the black popup window (just like Step 2) since we haven't actually told the window to show anything. So in this step, we're going to add some content to the window display, starting with some simple text.

To do this, you need to understand some basic concepts about how the display system works in the MPF media controller.

Since the folks who originally started MPF spend a lot of time giving presentations, the display concepts and terminology are pulled from presentation software like Microsoft PowerPoint or Apple Keynote. So if you're familiar with those, you should be familiar with the display concepts in the MPF MC.

First is the concept of *slides*. Just like a PowerPoint presentation, an MPF display is essentially a window frame that shows slides. Many slides can exist, but only one is shown at a time, and that slide takes up the entire display. (Just like how a PowerPoint slide takes up the whole display when you're playing the slide show.)

In MPF-MC, when one slide switches to another, there can be an animated "transition", like fade, push in, move out, etc.

A slide is like a blank canvas that you put things on. The "things", in this case, are called *widgets*. MPF has different types of widgets, for example, text, images, videos, shapes, lines, etc. When you put a widget on a slide, you can specify all sorts of properties, like the size, position, alignment, colors, etc.

One slide can have lots of different widgets, and you can specify the order widgets are drawn to control which ones are "on top" of others. You can add and remove widgets from existing slides at any time, and you can also animate widget properties, meaning you can change the opacity (to make them flash), or you can animate their position, size, etc.

All of this will become more clear throughout the tutorial, so let's just jump right in.

In order to show some text, we first have to create a slide, add a text widget to that slide, and make that slide the active slide on the display.

So first let's create the slide. There are several ways to do this, so we're just going to show you one way here and then you can read the full documentation on *slide* later.

In MPF, all slides have names. You can define slides in the `slides:` section of the config. So let's create a slide called "welcome_slide", like this:

```
slides:
  welcome_slide:
```

Now let's add a `widgets:` section under that slide, then under that, we'll start creating some widgets.

```
slides:
  welcome_slide:
    widgets:
```

You can add as many widgets as you want to a slide. (And it's pretty common for slides to be made up of lots of widgets). For now let's add a text widget that reads "PINBALL!". Do this by adding the following to your config:

```
slides:
  welcome_slide:
    widgets:
      - type: text
        text: PINBALL!
```

There are a few things going on there.

First, notice that before the word `type:`, there's a dash (hyphen), followed by a space. This is how you specify a list of items in YAML. (Think of it kind of like the YAML version of a bullet list.) You need to do this when adding widgets to a slide since a single slide can have more than one widget, so the dash tells the YAML file (and MPF-MC) where the settings for one widget end and the next begin.

Second, the space AFTER the dash is important. WRONG: `-type: text` RIGHT: `- type: text`

The `type: text` line is telling MPF-MC that this entry is for a text widget. And the `text: PINBALL!` is setting the text for this widget to be "PINBALL!". (For now we're just hard-coding the text to be "PINBALL!", but in the future we'll look at how you can use dynamically-updating text (like for the player score) that updates automatically whenever it changes.

Now run `mpf mc` and what do you see?

A blank window again! :(

The reason the window is still empty is because even though we created a slide (called "welcome_slide") and we added a widget to that slide, we didn't actually configure MPF-MC to *show* that slide. So let's do that now.

4. Add a slide_player config

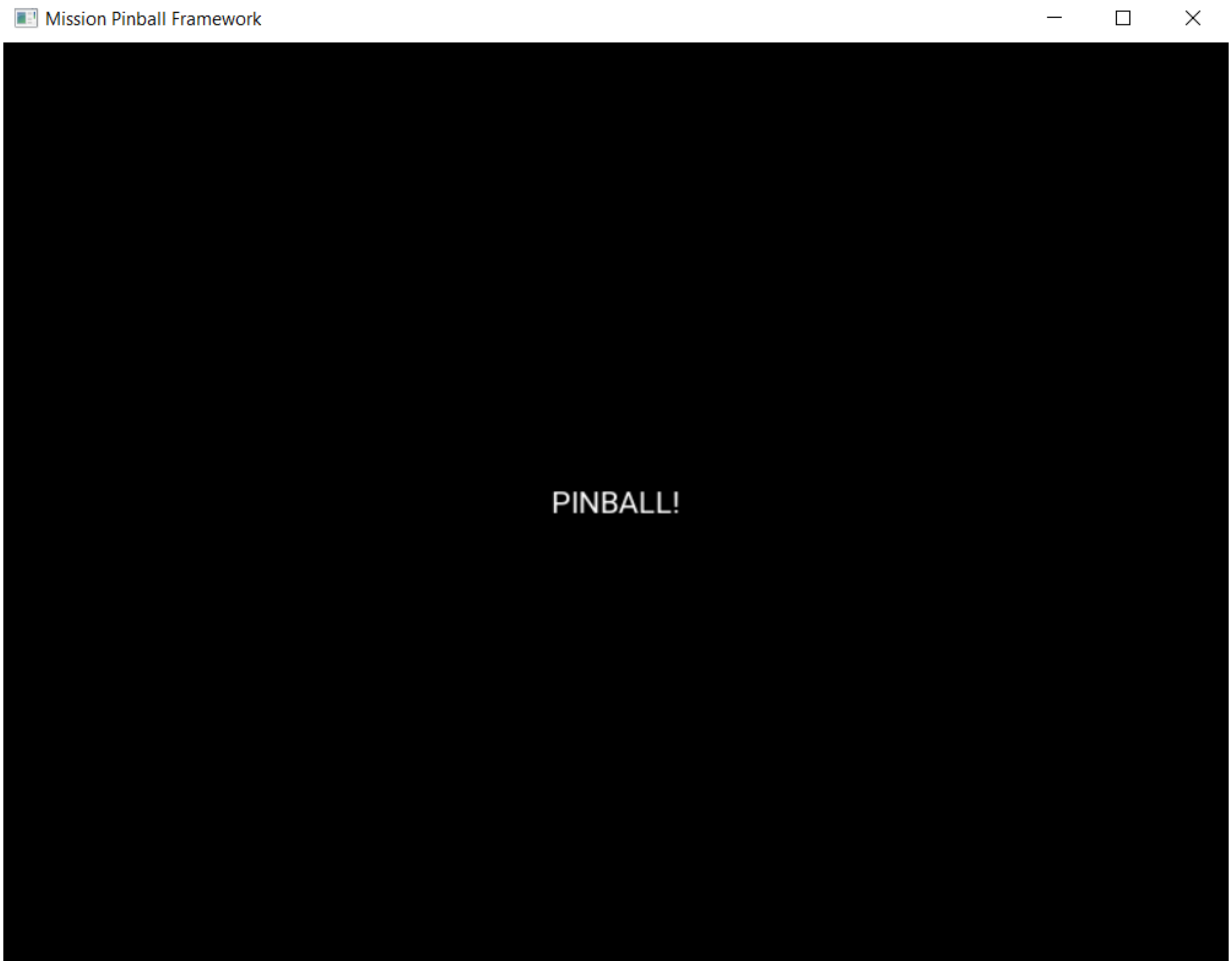
Next, create a new section in your config called `slide_player:`. The slide_player watches for certain events to occur, and when they do, it "plays" a slide.

To see this in action, add the following section to your machine config:

```
slide_player:
  init_done: welcome_slide
```

What this is doing is saying, "When the event called *init_done* happens, play the slide called *welcome_slide*." The *init_done* is an event that's posted by MPF-MC at the earliest possible point when it is ready after it initially starts up (literally it's saying "the MC is ready"). So what we're doing here is telling MPF-MC to show our welcome slide as soon as it can. (Check out the *events* documentation for details on what events are.)

To verify, run `mpf mc` again, and hopefully you see something like this:



Cool! We have text! Of course it's kind of small, and white, but it confirms that everything is working. Again, what's actually happening here is:

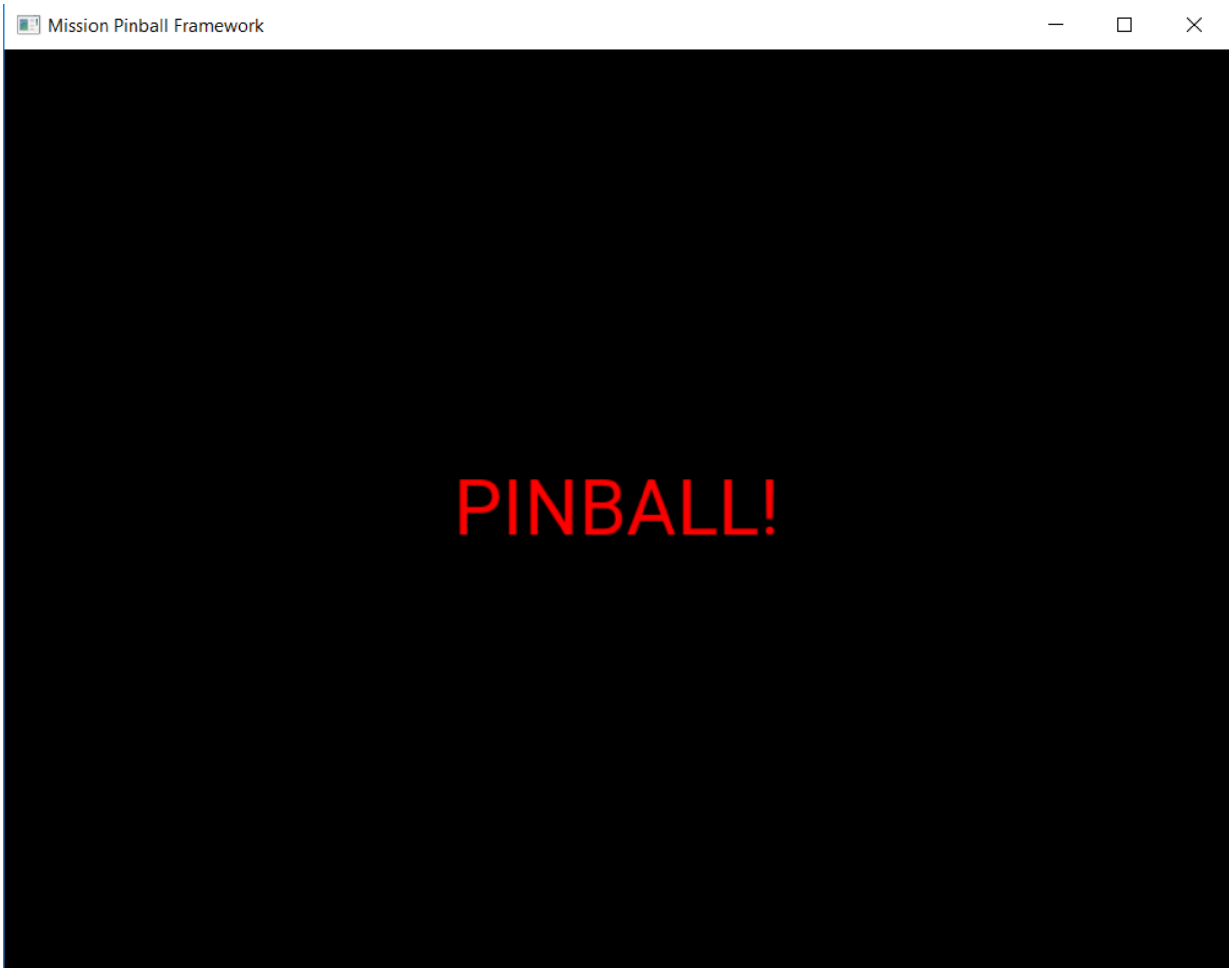
- You have a display called "window",
- which is showing a slide called "welcome_slide",
- because the slide_player was configured to show that slide when the "init_done" event happened, and
- that slide has a single widget,
- which is a text widget with its text set to "PINBALL!".

There are lots of settings for each widget type that you can use in your config file. Since this is a text widget, we can look at the *documentation for text widgets* to see what options we have.

For example, let's change the font size and the color, by adding `font_size:` and `color:` lines:

```
slides:
  welcome_slide:
    widgets:
      - type: text
        text: PINBALL!
        font_size: 50
        color: red
```

Now when you run `mpf mc` again, you should see this:



By default, the widget is centered in the slide, but you can play with different settings to position it wherever you want. (Check out *How to position widgets on slides* for details.)

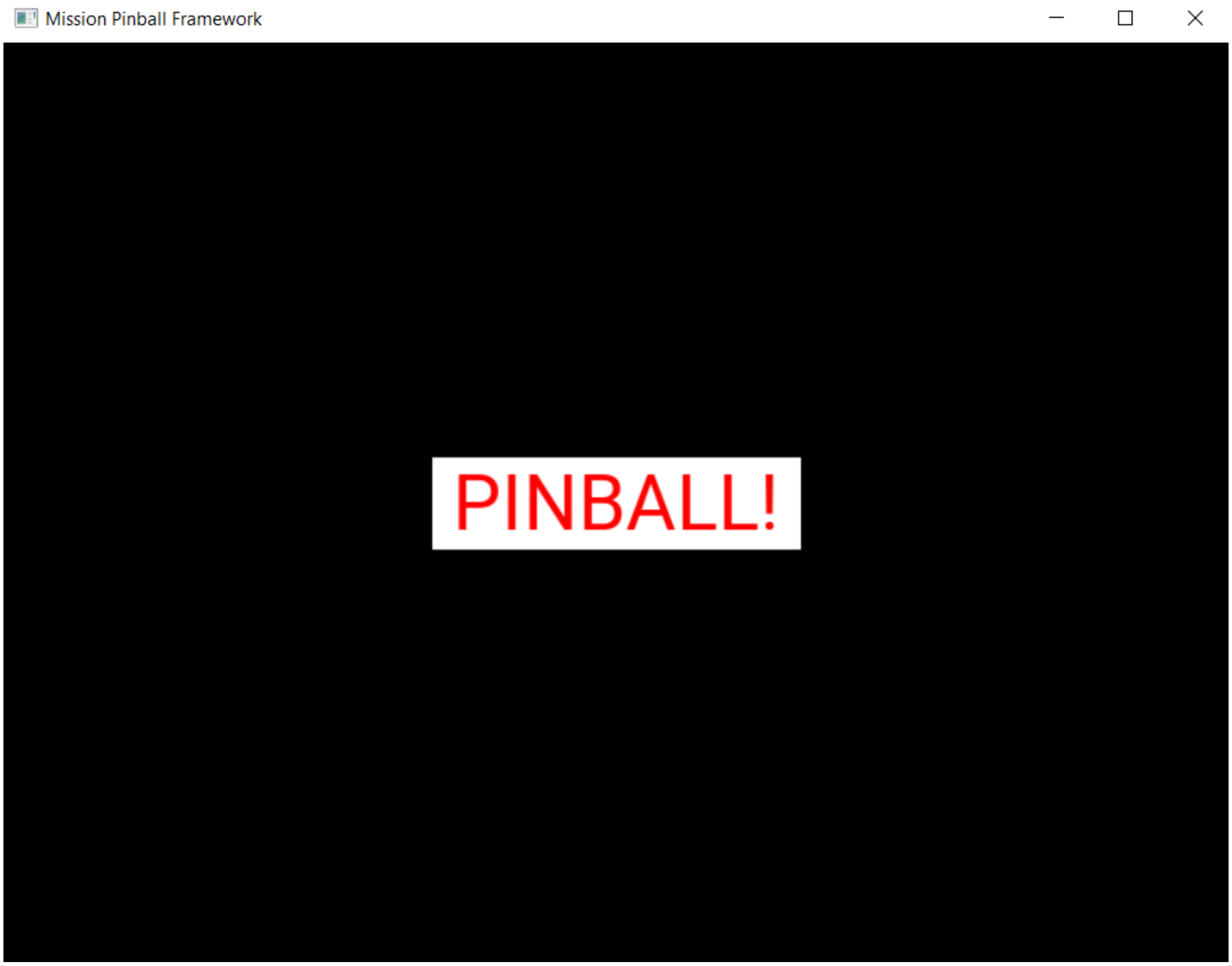
5. Add a second widget

We already mentioned that you can add as many widgets as you want to a slide and that there are lots of different kinds of widgets. Let's add a second widget to your welcome slide. This one will be a rectangle which appears behind the word "PINBALL!".

```
slides:
  welcome_slide:
    widgets:
      - type: text
        text: PINBALL!
        font_size: 50
        color: red
      - type: rectangle
        width: 240
        height: 60
```

Again, note that you use a dash followed by a space to denote the start of the second widget. This widget's type is "rectangle", with its height and width specified. Since we're not specifying any position, it will be centered (just like the text widget), and since we're not specifying a color, it will be white.

Now when you run `mpf mc`, you should see this:



Note that the word "PINBALL!" is "on top" of the white rectangle. That's because the order of the widgets on the display matches the order they're entered into the config file. So in this example, since the text widget comes first in the list of widgets for the welcome slide, the text widget is on top. If you switch the order and run `mpf mc` again, you'll just see the white rectangle with no text, since the rectangle would be "on top" and it would completely cover the PINBALL! text.

6. Run MPF-MC and the MPF game engine at the same time

Ok, so now you're able to run the media controller to get some widgets to show up. But so far, you were just running `mpf mc` which is running the media controller by itself, without the MPF game engine running.

So in this step, we're going to run them both at the same time.

The first thing you need to do is add another slide to your config for the MC to play, and this time we'll make that slide play on a different event.

So in your `slides:` section, add another slide called `attract_started`, like this:

```
slides:
  welcome_slide:
    widgets:
      - type: text
        text: PINBALL!
        font_size: 50
        color: red
      - type: rectangle
        width: 240
        height: 60
  attract_started:
    widgets:
      - text: ATTRACT MODE
        type: text
```

Note that `attract_started:` is indented the same number of spaces as `welcome_slide:`. Also note that in the attract_started slide, we switched the order of `text:` and `type:`. We did that here just to demonstrate that the order of settings in the config doesn't matter.

If you run this, nothing different will happen because all we did here in the slides section is define a slide. We need to use the `slide_player:` section to actually play the slide when some event happens.

So next, go to the `slide_player:` section of your config and add an entry for the event `mode_attract_started`. (This is the event that is posted whenever a mode starts, in the form of *mode_<mode_name>_started*.)

By the way, if you're wondering how we know what events to use, there's an *event reference* in the documentation which has a list of all the events in MPF as well as descriptions of when they're posted. You can use any of these as triggers for your slides via the `slide_player:`.

Anyway, add the `mode_attract_started` to your `slide_player:` like this:

```
slide_player:
  init_done: welcome_slide
  mode_attract_started: attract_started
```

Again, this is saying you want the slide called "attract_started" to play when the event called "mode_attract_started" happens.

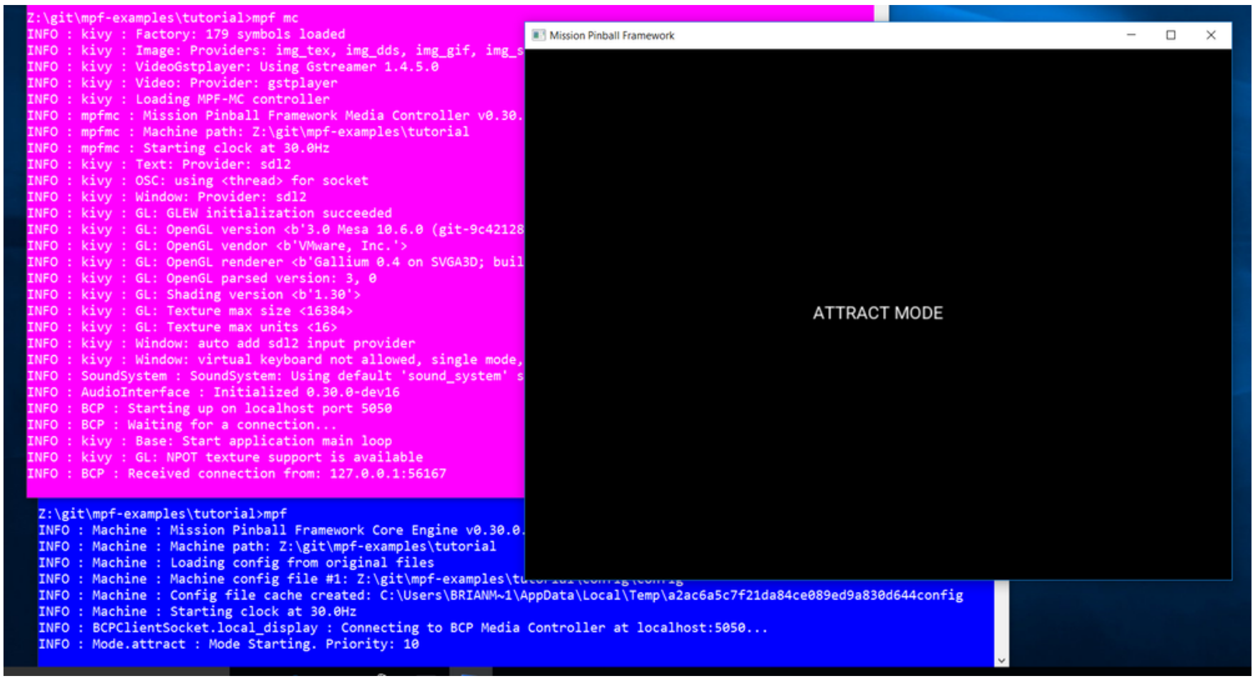
Now run `mpf mc` again. At this point you should see the welcome slide with the PINBALL! text. (You see the welcome slide because the MPF game engine isn't running, and the game engine is responsible for starting and stopping modes. So no game engine means no attract mode, and no attract mode means no *attract_mode_started* event, which means no *attract_started* slide.)

Now open a second terminal window and switch into your game folder and launch the MPF game engine. Remember from prior steps that we ran MPF with the `-b` option which told MPF to *not* try to connect to the MPF-MC. But now we have the MC running, so we want to run MPF without `-b` so it connects.

So this time, just run `mpf -t`, like this:

```
C:\pinball\your_machine>mpf -t
```

We added `-t` to disable the text ui on MPF because it might hide errors. When you run MPF, after some stuff scrolls by, you should see the *attract_started* slide replace the *welcome_slide*, like this:



So now MPF is running, it's talking to the MC, and you have the world's most boring attract mode!

To quit MPF, just make sure the graphical window has focus and hit the `Esc` key. That should cause both the MPF game engine and the MC to exit. (If they hang for some reason, you can click in the console window of the one that's hanging and press `CTRL+C` to kill it.)

Note that in the screen shot above, the colors of the command windows were changed. The magenta window is where `mpf mc` was run, and the blue window is where `mpf` was run.

Since the *attract_started* slide only has one widget, and since all we did with that widget is specify text (but not size, color, position, font, etc.), we just get the default text properties which are small, arial, and white.

7. Launching the MPF game engine and MPF MC at the same time

In the previous step, you used two separate console windows to launch `mpf mc` and `mpf` separately. (If you do this, by the way, you can launch either one first and it will wait for the other one.)

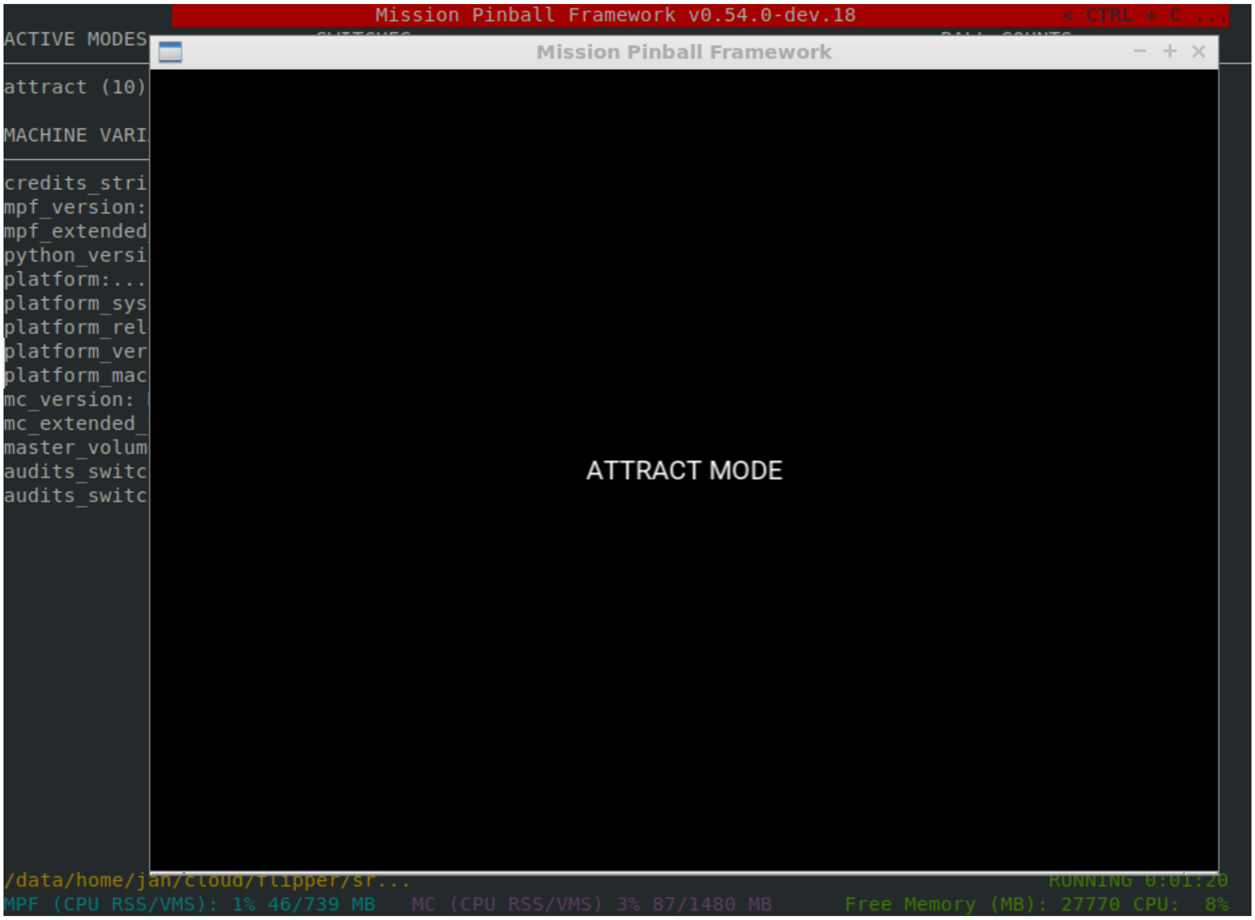
That's nice for learning purposes, but kind of annoying for everyday use. Fortunately there's a command called `mpf both` which launches both the game engine and the MC together.

Note: If you're using a Mac, you need to use MPF 0.32 or newer for `mpf both` to work.

Use it just like the others:

```
C:\pinball\your_machine>mpf both
```

When you do this, you should see the graphical window pop up (most likely showing the *welcome_slide* for a quick flash), then when the MPF game engine is up and running, you should see the graphical window flip over to the *attract_started* slide. Here's a screen shot:



This time we omitted `-t` and you will see the text ui again instead of the console log. You can also use `mpf both -t` if you prefer the log.

Check out the complete config.yaml file so far

If you want to see a complete `config.yaml` file up to this point, it's available in the "tutorials" folder of the mpf-examples package that you should have downloaded in Step 1 of this tutorial.

There are config files for each step, so the config for Step 5 should be at `/mpf-examples/tutorial/step_5`.

You can run this file directly by switching to that folder and then running the following command:

```
C:\mpf-examples\tutorial>mpf both
```

What if it doesn't work?

If you can't get it to work, there are a few things to look at.

If you get some kind of "KeyError" like `KeyError: 'welcome_slde'`, that means that it's looking for something it didn't find. Most likely this is the slide player looking for a slide that doesn't exist, so make sure the slide's entry in the `slides:` section matches the slide's name in the `slide_player:` section.

If the welcome slide works but you never see the attract slide, make sure you have the `mode_attract_started:` event name spelled properly. Also make sure you do *NOT* run MPF with the `-b` option since that tells it not to connect to the MC.

If you get YAML errors either copy the complete example below or read our *Debugging YAML Parse Errors* guide.

Most of the other errors should be pretty self-explanatory. If you get stuck, feel free to post to the mpf-users Google group.

Tutorial step 6: Add keyboard control

Once you get to this point, you should be able to run the MPF game engine as well as the media controller, and you should have a pop-up window which shows some text. You should have your flippers configured, and if you have a physical machine connected, you should be able to flip.

In this step, we're going to add some keyboard settings to your machine config which will let you map keyboard keys on your computer to switches in your pinball machine. This lets you "play" your game on your computer, which is useful for (1) cases where you don't have a physical machine nearby, and (2) scenarios where your pinball machine is all the way on the other side of the room and you don't feel like getting up every time you start MPF.

1. Create your key-to-switch mappings

The first step is to create your key-to-switch mappings in your config file. You do this by adding a `keyboard:` section, and then in there you add entries for each keyboard key and what type of action in MPF you want to map them to. (Switches, in this case.)

Here's an example where we map the left flipper button to the `Z` key and the right flipper button to the ? key:

```
keyboard:
  z:
    switch: s_left_flipper
  '?':
    switch: s_right_flipper
```

Note that the question mark is in quotes since it's a non-standard character, and if you don't put it in quotes, it will confuse the YAML parser.

Also it's weird that the key is the question mark, because if you push that key normally it types a slash. (The question mark is the shift option for that key.) So if you set a key mapping and it doesn't work, try the other character on the key.)

Again make sure that you have proper YAML formatting. The `z:` and `"?":` entries should indented the same number of spaces, and the "switch" words should be indented further. Also make sure you have a space to the right of the colon after `switch:`. At first you might think it's a bit tedious to have to write the word "switch" for each line. After all, why can't you just enter them as `z: s_left_flipper`? This is because the MPF keyboard interface can actually be used to control *a lot more than just keys*. The details of that are not important now, so for now just make sure your `keyboard:` section looks like the example above.

2. Test your new keyboard interface

At this point we're ready to test this out. Pretty simple. Save your config file and run your game again. (Seriously, we can't tell you how many times things don't work only to realize we didn't save our config after changing it!). So now run your game, starting both the media controller and the MPF core. Again you can either do this by running both commands manually in separate windows or by running `mpf both -t`.

Note that if you have a physical machine connected, *your physical flippers will not flip with the keyboard keys*.

Let's repeat this to be clear. If MPF is connected to physical hardware, pushing flipper button keys on your keyboard will not actually operate your physical switches. (We'll cover why not in Step 3 below.)

In order for the keys to work, the catch is that the graphical popup window (the one with the attract mode slide in it) has to be the active window for it to receive the keys. (It has to have "focus", in OS parlance.) Just like how your typing is only sent to the current active window on your desktop, the media controller's graphical window has to be active for your game to see your keystrokes and convert them to switches. So make sure this window is active (you can ALT+TAB to it or click on it).

Then try hitting the "Z" and "/" keys, and you should see them show up in your console window which is running the MPF game engine as MPF switch events, like this:

```
INFO : SwitchController : <<<<< switch: s_left_flipper, State:1 >>>>>
INFO : SwitchController : <<<<< switch: s_left_flipper, State:0 >>>>>
INFO : SwitchController : <<<<< switch: s_right_flipper, State:1 >>>>>
INFO : SwitchController : <<<<< switch: s_right_flipper, State:0 >>>>>
```

When you hit a key that you've configured on your keyboard, it's actually received by the media controller which in turn converts it to switch name and sends it to the MPF game engine. (This is because the MC controls the popup window, not MPF, and you need a window to track key states.)

Notice that there are actually state changes each time you hit and release a key. The "State: 1" means that switch has become active (i.e. when you press down the key), and the "State: 0" means that switch has just become inactive (when you release the key). You can experiment with this by holding down a key and seeing the log event for the associated switch becoming active, and then when you release it you'll see that switch becoming inactive. Go ahead and play around with this, and notice that you can push and hold the two keys in different orders and combinations.

3. Why can't you "flip" your physical machine with the keyboard?

If you're working with a physical machine with this tutorial, you might be surprised to see that your flippers don't fire when you hit the `Z` or `/` keys! Even more confounding is that you will still see the flipper switch events in your console log, and if you reach over and hit the physical buttons on your machine, the flippers will work. So what gives?!?

This happens because MPF uses "hardware rules" to program quick-response mechanisms (like flippers), meaning the flippers are activated by the control system rather than MPF software.

Read the *How MPF handles "quick response" mechs (flippers, slingshots, etc.)* guide for details.

4. Install the MPF Monitor (optional)

While pressing keyboard switches is great and fast it would be a lot of work to map all your switches to the keyboard (and remembering which key does what). Therefore you can later use the *MPF monitor* to lay them out visually and trigger them with your mouse (you can start using it right now if you want).

What if it doesn't work?

If you don't see your switch events in the console when you press your keys, there are a few things you can try to troubleshoot:

- Double-check to make sure you actually saved your updated config file. :)
- Make sure no modifier keys (shift, control, etc.) are being pressed at the same time. Since there are way more switches in a pinball machine than keys on a keyboard, MPF lets you add modified keys to your `keyboard:` map. This means that MPF will see `Z`, `SHIFT+Z`, `CRTL+Z`, `SHIFT+CTRL+Z`, etc. all as different switches.
- Remember that the media controller's pop-up window has to be in focus. Make sure it's the active window on your desktop and try hitting your keys again.
- Remember that your physical flippers will not flip if you hit the keyboard keys for your flipper buttons.
- Check if numlock is enabled. This seems to be common issue on Windows 10. Disable numlock in this case.
- Make sure you started `mpf both -t` and did not omit `-t` as this would hide the log and show the text ui instead.

Check out the complete config.yaml file so far

If you want to see a complete `config.yaml` file up to this point, it's in the `mpf-examples/tutorial/step_6` folder.

You can run this file directly by switching to that folder and then running the following command:

```
C:\mpf-examples\tutorial>mpf both -t
```

Tutorial step 7: Add your trough

At this point you have a flipping machine with a display, but you don't have a "working" pinball machine since you can't start or play games.

So the next two steps in this tutorial, we're going to get your first two *ball devices* set up—your trough and plunger lane. (A ball device is anything in MPF that holds a ball).

1. Read about ball devices

In MPF, a "ball device" is any physical mechanism in your machine that holds a ball.

You can read more about ball devices in the *Ball Devices* documentation, which we recommend that you do now to familiarize yourself with the concepts. (You don't have to understand everything about them for now, just skim through that link so you get the basics.)

2. Add your trough and/or drain

Now that you understand what a ball device is, lets add your first ball device, which is going to be trough (or drain) device which collects balls that drain from the playfield and stores them while they're not in play.

Since there are so many different types of ball drain and trough configurations, we can't write a single tutorial that walks you through all of them.

Instead, we have several tutorials. :)

So your next step is to visit the *Troughs / Ball Drains* documentation which lists all the options (with pictures), as well as links to step-by-step guides which walk you through the setup of the particular type of trough or ball drain you have in your machine.

3. Enable debugging so you can see cool stuff in the log

Once you have your trough or drain device (or devices, in some cases) set up, add one more setting to that device:

```
debug: true
```

This setting causes MPF to write detailed debugging information about this ball device to the log file. You have to run MPF with the `-v` (verbose) option to see this.

This will come in handy in the future as you're trying to debug things, and it's nice because you can just turn on debugging for the things you're troubleshooting at that moment which helps keep the debug log from filling up with too much gunk.

For example, if you have a modern style trough with a jam switch, you'd add the debug setting like this:

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6, s_trough_jam
    eject_coil: c_trough_eject
```

(continues on next page)

(continued from previous page)

```
    tags: trough, home, drain
    jam_switch: s_trough_jam
    eject_coil_jam_pulse: 15ms
    debug: true
```

4. Don't test yet

Since the trough or drain device works hand-in-hand with the plunger lane, and since we haven't set up a plunger lane yet, it's not worth testing your config at this point. We'll get the plunger lane set up in the next step.

Check out the complete config.yaml file so far

If you're following along with the example tutorial configurations, at this point there could be some significant divergence between the examples and your machine since the examples are based on a Demolition Man machine with a modern opto-based trough.

We still have the examples which you can try, and they'll work fine because they use the "virtual" platform which doesn't connect to real hardware. So you can run them and follow along, but just be aware that they might not match your own files exactly.

The complete machine config is in the `mpf-examples/tutorial/step_7` folder.

You can run this file directly by switching to that folder and then running the following command:

```
C:\mpf-examples\tutorial>mpf both
```

Tutorial step 8: Add your plunger lane

In this step we're going to create your *plunger lane* (or *shooter lane* or *ball launcher* or *catapult* or whatever you want to call it).

This is the device that holds the ball after it's been ejected from the trough or drain where it sits waiting for the player to put it into play.

It's important to understand that a ball device is *anything that holds a ball*, even if that's just a divot in the wood with no switch where the ball sits waiting for the player to pull back on a spring plunger.

MPF's ball tracking only works if MPF knows where all the balls are at all times, which is why it needs to "know" about the plunger lane, and you let MPF know about a plunger lane by configuring it as another ball device.

1. Add your plunger/catapult/launcher/etc.

Like the trough, there are several different plunger designs. Some are purely mechanical, some launch the ball with a button which fires a coil, and some have both options. Also, some plunger lanes have a switch which the ball sits on while it's waiting to be plunged, and others don't.

Visit the *Plungers & Ball Launch Devices* documentation for pictures that show each option and step-by-step guides which walk you through configuring each type for MPF.

2. Revisit your trough/drain device and add it as source_device to your playfield

Even though this is mentioned in the how-to guides, once you have your plunger device set up, be sure to go back to your trough or ball drain device and add the new plunger lane as an eject target, like this:

```
eject_targets: bd_plunger
```

Of course you'd add the name that you gave your plunger device, which could be something like "bd_catapult" or whatever you called it.

Also, if you have a two-stage drain (like a System 11 machine), you'd add this to the second device (the one that feeds the plunger).

Tell the playfield to use the plunger for new balls:

```
playfields:
  playfield:
    tags: default
    default_source_device: bd_plunger
```

3. Check out the complete config.yaml file so far

Again, our example config will probably diverge from yours since you might have different types of drain and plunger devices, but we do have a complete machine conform for Demolition Man for this step which you can view in the `mpf-examples/tutorial/step_8` folder.

You can run this file directly by switching to that folder and then running the following command:

```
C:\mpf-examples\tutorial>mpf both
```

4. Fire up your game and test

Unfortunately there are a few more things we need to configure before you can play a full game, but if you want to test what you have so far, you can launch MPF and drop a ball into your trough and you should see some cool things in your log file. To do this, launch the MPF game engine with the `-v` command line options so it shows the verbose information in the log file, like this:

```
C:\pinball\your_machine>mpf -vbt
```

You don't have to launch the media controller this time since we're just looking at the console output of the MPF game engine, which is why we added the `b` command line option too. (The `b` option tells the MPF game engine not to use the BCP protocol and not to try to connect to the MC.) You also have to add `t` to disable the text ui to see the verbose log. Otherwise, you would only see the verbose output in the logfile in the `logs` directory of your machine.

Note: For more information about command line options take a look at *MPF commands* and *mpf game (command-line utility)*.

Once your game is running, drop a ball into your trough and you should see a whole bunch of trough switches changing between active (State: 1) and inactive (State: 0).

If you don't have a physical machine, you can run MPF with the `-v` option and see a bunch of stuff in the log too by hitting the keyboard keys for the trough switches which will add and remove balls.

Now quit MPF and open the MPF log file (which is in your machine's `/logs` folder). Grab the latest file with "mpf" in the name (if you ran `mpf both` then you'll have separate log files from MPF and the MC).

Search (or filter) the log for the name of your trough or drain device, and you should see all sorts of interesting things. Here's a small snippet:

```
2016-11-18 03:54:06,103 : DEBUG : ball_device.bd_trough : Counting balls by checking switches
2016-11-18 03:54:06,103 : DEBUG : ball_device.bd_trough : Confirmed active switch: s_trough1
2016-11-18 03:54:06,103 : DEBUG : ball_device.bd_trough : Confirmed active switch: s_trough2
2016-11-18 03:54:06,103 : DEBUG : ball_device.bd_trough : Confirmed active switch: s_trough3
2016-11-18 03:54:06,103 : DEBUG : ball_device.bd_trough : Confirmed active switch: s_trough4
2016-11-18 03:54:06,103 : DEBUG : ball_device.bd_trough : Confirmed active switch: s_trough5
2016-11-18 03:54:06,103 : DEBUG : ball_device.bd_trough : Confirmed inactive switch: s_trough_jam
2016-11-18 03:54:06,103 : DEBUG : ball_device.bd_trough : Counted 5 balls
2016-11-18 03:54:06,103 : DEBUG : ball_device.bd_trough : Switching to state idle
```

What if it doesn't work?

If you've gotten this far and your trough, drain, and/or plunger isn't working right, there are a few things you can try:

If your log file shows a number of balls contained in one of your devices doesn't match how many balls you actually have, that could be:

- You didn't add all the ball switches to the *ball_switches:* section of the device's config.
- Your trough uses opto switches but you didn't add *type: NC* to each switch's configuration.
- A a switch isn't adjusted properly so the ball is not actually activating it. (Seriously, we can't tell you how many times that's happened! We've also found that on some machines, if you only have one ball in the trough that the single ball isn't heavy enough to roll over the top of the eject coil shaft. In that case we just add a few more balls to the machine and it seems to take care of it.) Either way, if you have a ball in the trough, the switch entry in your log should show that the switch is active (*State:1*), like this:

```
2014-10-27 20:05:29,891 : SwitchController : <<<<< switch: trough1, State:1 >>>>>
```

If you see State:1 immediately followed by another entry with State:0, that means the ball isn't activating the switch even though it might be in the trough.

If you get a YAML error, a "KeyError", or some other weird MPF error, make sure that all the switch and coil names you added to your ball device configs exactly match the switch and coil names in the `switches:` and `coils:` sections of the machine config.

Also make sure that all your names are allowable names, meaning they are only letters, numbers, and the underscore, and that none of your names start with a number.

Finally, make sure your YAML file is formatted properly, with spaces (not tabs) and that you have no space to the left of your colons and that you do have a space to the right of your colons. See our *Debugging YAML Parse Errors* guide if you got YAML errors. At this point your trough is ready to go! Next we have to configure your plunger lane.

Tutorial step 9. Add the start button

Obviously in order to play an actual game, you have to be able to start a game, and that requires a start button. So let's add that now.

1. Add a switch for your Start button

First, add the switch for your start button to the `switches:` section of your config file. Again this should be easy by now. In this tutorial we'll just call this button `s_start` and add it like this:

```
switches:
  s_start:
    number: 11
```

2. Add a "start" tag to your Start button

Just like the special-purpose tags we used when configuring the ball devices, MPF uses some *special purpose tags for switches*, too. One of them is `start`, as MPF watches for switches tagged with "start" to start games and add players to running games.

Sometimes people ask "Why do you use a tag for this? Why not just look for a switch named "start?" Again, we want MPF to be as flexible as possible, and we feel that game builders should be able to name their switches whatever they want. (Some want to preface with `s_`, others might not, etc.) So we use a "start" tag behind the scenes to make whatever switch you want act as the start button. So now your start switch in your `switches:` section should look like this:

```
switches:
  s_start:
    number: 11
    tags: start
```

3. Add keyboard entries for your start switch

If you're keeping your keyboard shortcuts up to date, you can create a keyboard entry for your start switch. This is especially helpful if you're building a custom machine from scratch and you don't have a physical start button wired up yet. In that case just enter some dummy value for the `number:` of your start switch. Then when you run a physical machine (without the -x command line option), you can start the game with your computer keyboard but actually play it on physical hardware. For your start button keyboard key, how about using the `S` key? To do so, add an entry like this to the `keyboard:` section of your config file:

```
keyboard:
  s:
    switch: s_start
```

4. Add at least one playfield switch

Another thing you need to do is to configure at least one playfield switch. Why? Because when a ball is launched from your plunger onto the playfield, MPF "confirms" that the ball actually made it onto the playfield when a playfield switch is activated. How do you configure a switch as a playfield switch? You use tags, by adding a `playfield_active` tag to a switch.

At this point you might be wondering, "Wait, I thought the eject_timeouts for the plunger was used to let MPF know when a ball really made it out of the plunger?" That's true, and technically at this point you don't need a playfield switch. However, this will speed up your ejects in a real machine and you'll eventually tag all your playfield switches with `playfield_active`, so we're just getting starting on this now. To do this, create a new entry in your `switches:` section for one of your playfield switches, for example:

```
switches:
  s_right_inlane:
    number: 12
    tags: playfield_active
```

Note: The tags playfield_active and above the start tag are *special purpose tags for switches*.

While you're at it, create a keyboard key mapping for this switch in the `keyboard:` section of your config, like this:

```
keyboard:
  q:
    switch: s_right_inlane
```

If you want you can go ahead and add entries for all your playfield switches, though that will take awhile. For now just make sure you have at least one, and make sure the ball hits that switch after it launches from the plunger before it drains. (There are lots of options for what you can do if a ball drains before it hits a switch, but we're not going to go into those now.)

If you do decide to add all your playfield switches now, you'll want to add the *playfield_active* tag to all the switches that might be hit by a ball being loose on the playfield. (So lane switches, ramp switches, rollovers, standups etc.) You do *not* want to tag ball device switches with `playfield_active` since if a ball is in a ball device, then it's not loose on the playfield.

At this point we're really, really close! There are a few more quick things we want to do, then run some checks. But then we're ready to play a real game!

Check out the complete config.yaml file so far

If you want to see a complete `config.yaml` file up to this point, it's in the `mpf-examples/tutorial/step_9` folder.

You can run this file directly by switching to that folder and then running the following command:

```
C:\mpf-examples\tutorial>mpf both
```

Tutorial step 10: Run a real game

Holy Moly! It's actually time to run your first real game with MPF. When we say a "real" game, we're talking about with multiple players and balls machine flow from attract to game mode and back to attract once the game is over.

1. Make one quick addition to your display configuration

We know that at this point, you just want to run your game. The problem is if we run it now, the display will continue to show "ATTRACT MODE" throughout the entire game since we haven't configured it for anything else. So let's make a quick addition to the `slide_player:` section of your config so it will show the player and ball number when a game is in progress. (Later in this tutorial we'll revisit this and explain what's actually going on. For now just make this change.) In your config file, add a `ball_started:` entry with the following information. Your complete `slide_player:` section should now look like this:

```
slide_player:
  init_done: welcome_slide
  mode_attract_started: attract_started
  ball_started:
    widgets:
      type: text
      text: PLAYER (number) BALL (ball)
```

2. Add initial active switches and bind trough switches to your keyboard

If you are not using physical hardware you need some way to control the ball inside your trough. We will first make sure that the trough switches will be active (as if there was a ball sitting on them) when your virtual machine starts up. Additionally, we add keyboard bindings for ball switches to the numbers 1 to 5 and the plunger switch to p.

```
virtual_platform_start_active_switches:
  - s_trough1
  - s_trough2
  - s_trough3

keyboard:
  1:
    switch: s_trough1
    toggle: true
  2:
    switch: s_trough2
    toggle: true
  3:
    switch: s_trough3
    toggle: true
  4:
    switch: s_trough4
    toggle: true
  5:
    switch: s_trough5
    toggle: true
  p:
    switch: s_plunger
    toggle: true
```

This way you can drain balls by activating trough switches.

3. Change your flipper config so they don't automatically enable on machine boot

Almost there! The other quick change we need to make is to remove the `enable_events:` from the flipper configuration that we added back in the *Get Flipping!* step.

This is because by default, MPF will automatically enable your flippers when a ball starts and disable them when a ball ends. But since we added a configuration setting to your flippers that set them to automatically enable themselves immediately when MPF loaded, that setting overwrote the default setting which enables your flippers when a ball starts. So as your config file is now, the flippers enable when MPF boots, then they disable when the first ball ends, and that's it. They won't enable again for Ball 2.

To make this change, simply remove the `enable_events: machine_reset_phase_3` line from each of your two flipper sections of your config file. So now your ``flippers:` section should look like this: (It might not be 100% identical since you might have single-wound flipper coils and/or EOS switches.)

```
flippers:
  left_flipper:
    main_coil: c_flipper_left_main
    hold_coil: c_flipper_left_hold
    activation_switch: s_left_flipper
  right_flipper:
    main_coil: c_flipper_right_main
    hold_coil: c_flipper_right_hold
    activation_switch: s_right_flipper
```

4. Running your game with physical hardware

If you have a physical machine attached, go ahead and run your game without the `-x` or `-X` command line options. (If you don't have a physical machine and you want to simulate a game using the keyboard keys, skip to Step 4 below.)

```
C:\pinball\your_machine>mpf both -X
```

Make sure you have at least one ball in the trough and then run your game. The display should display "ATTRACT MODE." Hit the start button. A ball should be kicked out of the trough and into the plunger lane, and the display should change to "PLAYER 1 BALL 1." If you have a coil- fired plunger, you should be able to hit the launch button and the coil should fire. If you have a manual plunger, you should be able to plunge and flip. If you hit the start button a second time during Ball 1, a second player should be added. (The display won't show this since we haven't configured it to show a message, but you can see this in the logs and when the ball drains then it should go to Player 2 Ball 1 instead of Player 1 Ball 2.)

A few caveats to this early bare-bones game:

- Since you haven't configured any scoring yet, this game will be boring and nothing will score. But hey, you're playing!
- If your flippers, trough eject, or plunger coil is too weak or too strong, you can adjust them in the coil's `default_pulse_ms:` setting in the config file.
- If you start MPF with a ball in the plunger lane and you have a coil-fired plunger, MPF will immediately fire the plunger to kick out the ball. This is by design since you don't have a "home" tag in your plunger ball device's configuration, which means that MPF will automatically eject the ball to get all the balls into ball devices tagged with "home."

- If you shoot a ball into a playfield lock or any other ball device, it will get stuck there since you haven't configured that device. (In this case you need to add configuration entries for those ball devices so MPF can know about them. Then it will automatically kick out any balls that enter. We'll get to that later.)
- By default MPF is configured to allow a maximum of 4 players per game, with 3 balls per game. You can change this in the *game:* section of the machine config.

5. "Playing" a game without a physical machine attached

If you've been adding keyboard switch map entries to your config file as you've been going through this tutorial, you can actually "play" a complete game on your computer keyboard. Here's how you do it:

1. Launch the MPF game engine and the MC. Note that in order for this to work, we want to use the "smart virtual" platform. This will be the default, but make sure you do not have `platform: virtual` in your config. (If you do have a platform entry in your config, make sure it's `platform: smart_virtual`.) If you have a different platform setting for your physical hardware, you can still run without the hardware connected by using the `-X` (uppercase *X*) command line option to specify the smart virtual platform interface.
2. Push the "S" key to start a game. At this point MPF will eject a ball from the trough to the plunger
3. If you have a coil-fired plunger, push the "L" key (or whatever key you mapped to your launch button) to launch the ball.
4. If you do not have a coil-fired plunger, push the "P" key (or whatever key you mapped to your plunger lane switch) to un-toggle that switch which simulates the ball leaving the plunger lane. Note: The toggle option in the *keyboard:* section is useful for testing your game from your computer when you're not around your physical machine.
5. Now you can "flip" with the "Z" and "?" keys.
6. After you get bored of this, push the "1" key to activate a trough ball switch. At this point MPF will think a ball drained and you should see the display switch to Ball 2 and the trough switch should open and the plunger lane switch should close as the "smart virtual" platform ejects a ball from the trough to the plunger.
7. Repeat until you're bored.
8. After Ball 3 is over the display will change back to the "ATTRACT MODE" text and you can push "S" again to start another game.
9. Congrats! You just played your first virtual pinball game. Yeah, it's boring, but you did it!

6. What if your game won't start?

If your game doesn't start or doesn't work, hopefully we've given you enough information in this tutorial to work out what the problem is. That said, here's a list of things that could go wrong:

- If you see a config error try running `mpf both -t -v -V -X` to disable the text ui and add verbose logging.
- No ball in the trough.

- Virtual games need balls too; add the `virtual_platform_start_active_switches` section of the complete config file. (Alternatively, if you are using the smart_virtual platform with `-X` press `1` and `2` to add balls to the trough via keyboard)
- Ball in the trough, but not activating the switch.
- Trough switches are optos but you didn't add `type: NC` to your switch configurations. (Mechanical trough switches do not need a `type:` setting.)
- Trough is trying to eject, but the trough coil's `default_pulse_ms:` setting is too weak and the ball can't get out.
- Incorrect switch or coil numbers which don't match up to your actual hardware inputs and outputs.
- Some other setting isn't configured properly, which could lead to who-knows-what error? (Maybe compare your config file to the complete config from mpf-examples?)

If you're still having problems, feel free to post to the mpf-users Google group.

Check out the complete config.yaml file so far

If you want to see a complete `config.yaml` file up to this point, it's in the `mpf-examples/tutorial/step_10` folder.

You can run this file directly by switching to that folder and then running the following command:

```
C:\mpf-examples\tutorial>mpf both -X
```

Remember though that unless you're following this tutorial with an actual *Demolition Man*, you'll have some differences in your config file.

Tutorial step 11: Add the rest of your coils and switches

Okay, so at this point you have a working game. The biggest problem you might run into is that if you shoot your ball into a playfield device like a VUK or popper, the ball will get stuck. Why? Because you haven't yet added the switches to your config file while let MPF know that a ball is there, and you haven't added the coils which MPF needs to fire to eject a ball. So MPF literally has no idea that those switches and coils even exist, which means it has no ability to detect a ball entering a device and to eject it. So when we're building a config for a new game, at this point we go through our config and add all the remaining switches and coils to and `switches:` and `coils:` sections of the config file.

1. Add the rest of your switches

This step is pretty simple. If you building a config for an existing machine, we usually use the operators manual as our starting point and just move down the list and add all the switches as they're listed in there. We don't worry about tags at this point *except for* `playfield_active` tag. We add this tag to any switch the ball can hit when it's active and rolling around on the playfield. (So this is going to be your lanes, slingshots, pop bumpers, ramp entry & exit switches, rollovers, stand up targets, and anything else the ball can hit when it's in motion. The tricky thing is that you do not add a `playfield_active` tag to switches in other ball devices (drop_targets, kickbacks, troughs or the shooter lane). For example, if you have a hole in the playfield that the ball rolls into which requires a coil pulse to kick it out of – that is not a playfield switch (since when the ball is in that hole, it's not actively rolling around the playfield). We'll actually set that switch up as a part of a ball device in a later step.

2. Add the rest of your coils

Next add entries for the rest of your coils, again using the operators manual as a guide if you're building a config for an existing machine. You don't have to worry about pulse times at this point—just get the coils added.

Check out the complete config.yaml file so far

If you want to see a complete `config.yaml` file up to this point, it's in the `mpf-examples/tutorial/step_11` folder.

You can run this file directly by switching to that folder and then running the following command:

```
C:\mpf-examples\tutorial>mpf both -X
```

Note that starting with this step, the actual coil, switch, and ball_device names don't 100% match with what we have in the tutorial. This shows you that there are lots of different options when it comes to naming things.

Tutorial step 12: Add the rest of your ball devices

Now that you've added all your switches and coils, you'll probably notice that the ball is *still* getting stuck in devices on the playfield when it enters them. This is because MPF doesn't know that certain switches and coils are associated with ball devices, so MPF doesn't know that it should fire a coil when a certain switch becomes active. So the next step is to create configuration entries for the rest of your ball devices.

The good news is that once you do this, a ball entering a device will automatically be ejected, so when you're done with this step, your ball shouldn't get stuck anywhere.

To do this, take a look at all the ball devices around your playfield and then create entries for each one in the *ball_devices:* section of your config file. Depending on your machine, you might have 5 or 6 of these. (Ball devices are anything where the ball could go where it's held and not actively rolling around on the playfield.) At a bare minimum, you need to add `ball_switches:`, `eject_coil:`, and `eject_timeouts:` settings for each ball device you add. The `eject_timeouts:` entry is critical, because if a ball ejects to the playfield but then doesn't hit a switch right away, this is the how long MPF will wait before assuming the ball made it out of the device successfully. (Again, set this timeout to be the longest amount of time that could pass with a ball failing to eject and falling back in.) Simple playfield kickouts might be fine with 500ms or 750ms, and VUKs might be around 2 or 3 seconds.

After you add all your ball devices, you should be able to play a game without the ball getting stuck anywhere! (And if you start MPF with balls already stuck in devices, MPF will automatically eject the balls when it boots because these additional devices do not have "home" listed as one of their tags.) Here's the `ball_devices:` section from a *Demolition Man* config file:

```
ball_devices:
  bd_trough:
    tags: trough, home, drain
```

(continues on next page)

(continued from previous page)

```
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough_jam
    eject_coil: c_trough_eject
    entrance_count_delay: 300ms
    jam_switch: s_trough_jam
    eject_targets: bd_plunger
    debug: true

  bd_plunger:
    ball_switches: s_plunger_lane
    entrance_count_delay: 300ms
    eject_timeouts: 3s
    eject_coil: c_plunger_eject
    player_controlled_eject_event: sw_launch

  bd_retina_hole:
    ball_switches: s_eject
    eject_coil: c_retina_eject
    eject_timeouts: 1s

  bd_lower_vuk:
    ball_switches: s_bottom_popper
    eject_coil: c_bottom_popper
    eject_timeouts: 2s

  bd_upper_vuk:
    ball_switches: s_top_popper
    eject_coil: c_top_popper
    eject_timeouts: 2s

  bd_elevator:
    ball_switches: s_elevator_hold
    mechanical_eject: true
    eject_timeouts: 500ms

playfields:
  playfield:
    tags: default
    default_source_device: bd_plunger
```

Remember that if you need to adjust the eject coil pulse time, you do that in the coil's property in the `coils:` section of your config file, not in the ball device configuration.

Check out the complete config.yaml file so far

If you want to see a complete `config.yaml` file up to this point, it's in the `mpf-examples/tutorial/step_12` folder.

You can run this file directly by switching to that folder and then running the following command:

```
C:\mpf-examples\tutorial>mpf both
```

Tutorial step 13: Add slingshots, pop bumpers, and other "autofire" devices

While we're setting up the basic playfield devices, let's configure the "autofire" devices like slingshots and pop bumpers. (An "autofire device" is anything where you have one switch and one coil and the switch being hit automatically causes the coil to fire.) This makes the game more fun since it's kind of sad to see a ball hit a slingshot and nothing happen. You add these autofire devices in the `autofire_coils:` section of your machine configuration. It's pretty simple. Just create an entry for the name you'd like to give that device, and then add sub-entries for the `switch:` and `coil:` for that device. For example, here's the `autofire_coils:` configuration for *Demolition Man*, which has two standard slingshots, and upper slingshot near the pop bumpers, and two pop bumpers (which we happen to refer to as "jets" in this config):

```
autofire_coils:
  left_slingshot:
    coil: c_left_slingshot
    switch: s_left_slingshot
  right_slingshot:
    coil: c_right_slingshot
    switch: s_right_slingshot
  upper_slingshot:
    coil: c_top_slingshot
    switch: s_top_slingshot
  left_jet:
    coil: c_left_jet_bumper
    switch: s_left_jet
  right_jet:
    coil: c_right_jet_bumper
    switch: s_right_jet
```

Autofire devices in MPF are somewhat intelligent. They will only be activated while a ball is in play during a game, which means they automatically deactivate themselves during attract mode and if the player tilts. (You can override these default settings as well as configure additional MPF events that will cause them to activate or deactivate. See the *autofire_coils:* section of the configuration file reference for details, though you don't have to do that now.)

Remember if you want to adjust the strength of these coils, you can do that in the coil's `default_pulse_ms:` setting in the `coils:` section of your config.

Check out the complete config.yaml file so far

If you want to see a complete `config.yaml` file up to this point, it's in the `mpf-examples/tutorial/step_13` folder.

You can run this file directly by switching to that folder and then running the following command:

```
C:\mpf-examples\tutorial>mpf both
```

Tutorial step 14: Add your first game mode

By this point in the tutorial you should have a "playable" game, though it's pretty boring because there's no scoring, no modes, and the display just shows *PLAYER X BALL X* the whole time.

So in this step the real fun will begin as we configure our first game mode! So far all of the configuration you've been doing has been machine-wide configuration which was stored in the `/config` folder in your game's machine folder.

All of the configuration options you added to the `config.yaml` applied machine-wide.

In this step, we're going to add a `/modes` folder to your machine folder. Then we'll add a subfolder for each game mode, and in each we'll create a YAML config file that controls what happens in that specific mode.

What's cool about MPF's modes system is that all of the configuration you do for a mode is only active when that mode is active. In fact from here on out, almost everything you configure will be at the *mode*-level rather than the *machine-wide* level. As we go deeper into the tutorial and the How To guides, you'll start to get a feel for what types of things should be in the machine-wide configuration versus the types of things that should be in mode-specific configurations. Pretty much all the hardware (coils, switches, lights, leds, ball devices, platform, DMD, etc.) are configured as machine-wide settings, and then game logic-type things (scoring, shots, sound effects, animations, light shows, etc.) are configured as mode-specific settings.

MPF can have as many modes running at once as you want. In fact you'll probably use this to your advantage, breaking up your game into lots of little modes to make the programming easier. (Many of these modes will not be "in your face" modes that the player is aware of. Things like skill shot, combo timers, super jet counters, etc., will all be configured as modes even though the player wouldn't think of them as modes.)

1. Read the documentation about modes

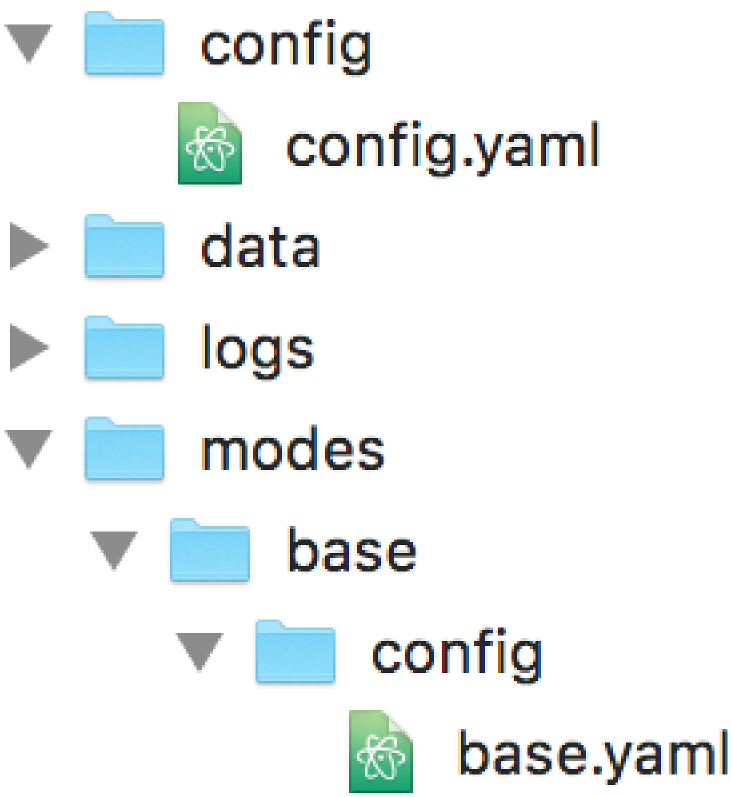
The first step to setting up a game mode is to understand how game modes work in MPF. So read *that documentation now* to get an overview, and then come back here for the step-by-step walk-through of doing your first mode.

2. Set up the folders & files for your "base" mode

The first mode we're going to create is a mode called "base." (Don't call it "game" because MPF has a built in mode called "game" that you don't want to overwrite.) This "base" mode will be running at all times while a game is in play and can be thought of as the "default" game mode. We'll set up default shots, scoring, configure the display to show the score, etc. Everything in the base game mode will be available if no other higher-priority modes are running. To create the base game mode:

1. Create a folder called `modes` in your machine's folder.
2. Create a subfolder for your `modes` folder called `base`. (You will ultimately create one subfolder for each mode you have, and the name of the folder controls the name of the mode.)
3. Inside your `base` folder, create a folder called `config`. (This folder will hold your mode-specific config files.)
4. Inside your `config` folder, create a file called `base.yaml`. (This is the default config file for your base mode. We use the naming scheme `<mode_name>.yaml` instead of `config.yaml` for these to make it easier to keep track of which files are which if you open a bunch of them at once in your editor.)

At this point your machine's folder & file structure should look like this:



3. Add your base game mode's settings to its config file

The settings that control a mode are configured in the mode's own configuration file. We do this because it allows modes to be completely self-contained. In other words, as long as you have a mode's folder and all its content, then you have everything you need for that mode.

So let's configure some settings for the base mode in the base mode's config file. To do this, open your new mode's `base.yaml` in your code editor. Add the config_version, then create a top-level configuration section called `mode:`. On the next line, indent four spaces and add the entry `start_events: ball_starting`. On the following line, also indent four spaces and type `priority: 100`. Your `base.yaml` file should now look like this:

```
##! mode: my_mode
#config_version=5
mode:
  start_events: ball_started
  priority: 100
```

There are lots more settings besides `start_events` and `priority` which you can set for a mode. See *mode:* for details.

The two settings we added here should be pretty obvious. The `start_events: ball_starting` means that this mode will automatically start when the MPF event *ball_starting* is posted. (In other words, this mode will start whenever a ball starts.) You can also enter a list of *stop_events* to control how the mode ends, though if you don't enter one here then the mode will automatically stop when the ball ends, so you don't have to specify a stop event now.

The `priority: 100` means that everything this mode does will have a base priority of 100. We'll create future modes at higher priorities so they can take over the display, control lights, filter and block scoring, etc. (You read the *documentation about modes*, right?)

Also, when you create your own modes, keep them between 100 and 1,000,000. MPF has some built-in modes above and below those values that should stay at the top and bottom of the priority stack.

4. Add your mode to your machine-wide config file

Now that you have a mode set up, you need to go back to your machine- wide configuration file to add this new mode to the list of modes that your game will use. At first you might think this is a bit confusing. After all, you just created a folder and a config file for your new mode, so why do you have to specify that mode in another location too?

The reason is we don't want to automatically include a mode in a game just because that mode has a folder in the *modes* folder. (After all, what if you're testing something out, or if you have multiple versions of a mode you're playing with? It would be dangerous if MPF just automatically loaded every mode it found.)

So instead we built MPF so that you have to add all the modes you want to be available in a game to a list in the machine-wide config file. To do this, go back to your machine-wide `config.yaml` file (in `<your_machine>/config/config.yaml`) and add a top-level section called `modes:`. (Like all the sections in your config file, you can put this section anywhere you want in your file. Maybe up towards the top so it's easy to find later?) Then on the next line, type two spaces, then a dash, then another space, then type `base`. So now that section of your `config.yaml` should look like this:

```
modes:
  - base
```

Note that it's very important that you put dashes in front of each mode in this list? Why? Because with dashes, MPF will be able to combine settings together in this list from different config files.

For modes that's important, because MPF has several built-in modes it uses for its own things. (For example, "attract" and "game" are both modes, and we'll be creating future ones that you might want to use too for tilt, volume control, game statistics, high score entry, credits, etc.)

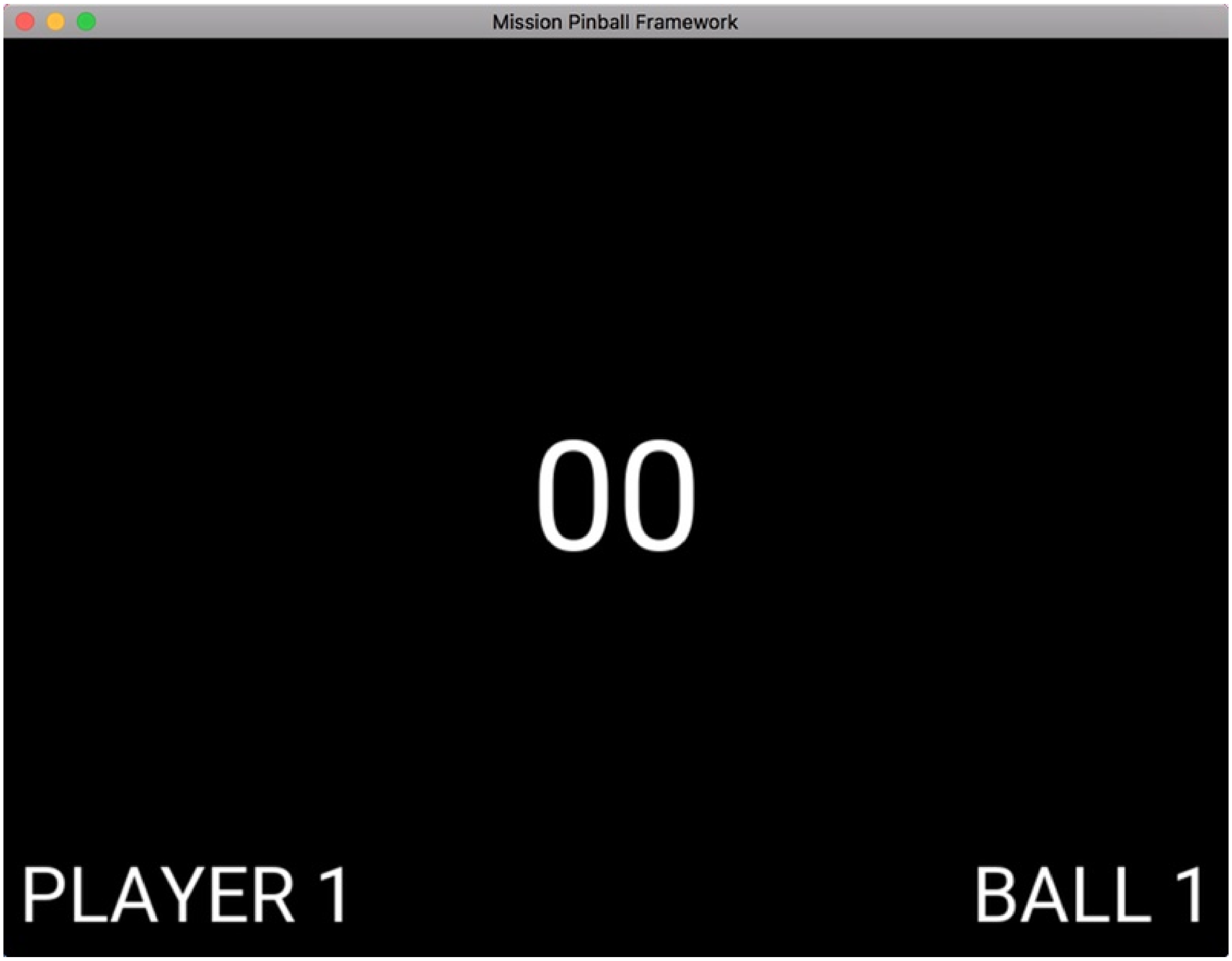
5. Run your game to verify your new mode works

Be sure to save the changes to `base.yaml` and `config.yaml`, and then run your game again. For this test, you do not need to use verbose logging since mode information is reported in the basic level of logging. Once MPF is running, start a game and you should see something like on the console and/or the log file when you run `mpf both -t`:

```
INFO : Mode.attract : Mode Starting. Priority: 10
INFO : SwitchController : <<<<< switch: s_start, State:1 >>>>>
INFO : SwitchController : <<<<< switch: s_start, State:0 >>>>>
INFO : Mode.game : Mode Starting. Priority: 20
INFO : Mode.game : Player added successfully. Total players: 1
INFO : Mode.base : Mode Starting. Priority: 100
INFO : SwitchController : <<<<< switch: s_trough_1, State:0 >>>>>
INFO : SwitchController : <<<<< switch: s_shooter_lane, State:1 >>>>>
INFO : SwitchController : <<<<< switch: s_shooter_lane, State:0 >>>>>
```

6. Make your base mode do something useful

We already mentioned that there are lots of different things you could add to your base mode. For now, let's configure the display so that it shows the player's score, as well as which player is up and what ball it is, like this:



To do this, go back to your base mode's config file (`<your_machine>/modes/base/config/base.yaml`) and add a section called `slide_player:`. Then add the following subsections so your complete `base.yaml` looks like this:

```
##! mode: base
#config_version=5
mode:
  start_events: ball_starting
  priority: 100

slide_player:
  mode_base_started:
    widgets:
      - type: text
        text: (score)
        number_grouping: true
        min_digits: 2
```

(continues on next page)

(continued from previous page)

```
        font_size: 100
      - type: text
        text: PLAYER (number)
        y: 10
        x: 10
        font_size: 50
        anchor_x: left
        anchor_y: bottom
      - type: text
        text: BALL (ball)
        y: 10
        x: right-10
        anchor_x: right
        anchor_y: bottom
        font_size: 50
```

We briefly touched on the `slide_player:` functionality earlier in this tutorial and how you can configure it to show certain slides when various MPF events happen.

Every time a mode starts in MPF, an event called *mode_(name)_started* is posted. So in this case, we set our slide player entry to play when it sees the event `mode_base_started` which means it will play that slide as soon as the base mode starts. (And since you configured your base mode to start based on the *ball_starting* event, this means this slide will be created and shown whenever a new ball is started.)

You may be wondering why we don't set that slide to play on the *ball_starting* event? The key to remember with game modes is that all the settings in your mode-specific config file are only active when the mode itself is active. In the case of our base mode, the *ball_starting* event is what actually causes the mode to start. When *ball_starting* is posted, the base mode starts and loads its configuration. At that point that *ball_starting* event has already happened, so if you set a slide to play within that mode then it will never play because it doesn't start watching for that event until after it happened. (Hopefully that makes sense?)

Anyway, if you look at the `slide_player:` settings, you'll see that the slide that is shown when the event `mode_base_started` is posted contains three text widgets. One that shows the score, one that shows the player and one that shows the current ball number. Note that the `text:` entries for those have have some words in parentheses.

Words in parenthesis signs are variables that are replaced in real time when they're updated. In this case these are "player variables" because they are values that belong to the current player. More on using dynamic text (that is, text that automatically updated itself as underlying values change), is *here*.

Also note that there are some additional positioning settings, like `x:`, `y:`, `anchor_x:`, and `anchor_y:`. You can read about these in our *How to position widgets on slides* guide.

Finally, note that the text widget showing the score has settings for `number_grouping:` and `min_digits:`. You can read about what those do in the *documentation for the text display widget*.

7. Remove the old slide_player: ball_started entry

Now that you have this cool score display from your new base mode, you can go into your machine-wide `config.yaml` and remove the slide_player: entry for ball_started:. So now the slide_player: in your machine-wide `config.yaml` should just look like this:

```
slide_player:
  init_done: welcome_slide
  mode_attract_started: attract_started
```

What if it didn't work?

- Make sure you actually start a game. Remember that this new base mode is only active when a ball starts from a game that's in progress, so you won't see the mode until a game starts. (If you're not able to start a game, check the troubleshooting tips in the previous step.)
- If you get some kind of crash or error, specifically any errors that mention anything about "config" or "path," double-check that you put all the files in the proper locations back in Step 2. (A common mistake is to put *base.yaml* in the */modes/base* folder rather than the */modes/base/config* folder.)

Check out the complete config.yaml file so far

If you want to see a complete `config.yaml` file up to this point, it's in the `mpf-examples/tutorial/step_14` folder.

You can run this file directly by switching to that folder and then running the following command:

```
C:\mpf-examples\tutorial_step_14>mpf both
```

Tutorial step 15: Add scoring

By now you have a "playable" game with a base game mode, and you've got a score showing on the display, but it's still pretty boring since nothing is actually configured to register a score yet. So in this step we're going to add some scoring.

1. Understand in scoring works in MPF

MPF includes a core module called the *Variable Player* which is responsible for adding (or subtracting) points from a player's score. Actually, that's not a completely accurate description. We should really say that the variable player is responsible for adding or subtracting value from any player variable. (A player variable is just a key/value pair that is stored on a per-player basis.) The *score* is the most obvious player variable. But MPF also uses player variables to track what ball the player is on, how many extra balls the player has, etc. You can create player variables to track anything you want. Ramps made, combos made, number of modes completed, aliens destroyed, etc.

The variable player is responsible for adding and subtracting value from any player variable based on events that happen in MPF. You configure which events add or subtract value to which player variables in the `variable_player:` section of a mode's configuration file.

2. Add a *variable_player:* section to your base.yaml mode config file

The first step is simply to add a `variable_player:` section to your base mode's *base.yaml* config file. So in this case, that will be `<your_machine>/modes/base/config/base.yaml`. Add a new top level configuration item called *variable_player:*, like this:

```
variable_player:
```

3. Add point values for events

Then inside the `variable_player:` section, you create sub-entries for MPF events that you map back to a list of player variables whose value you want to change. By default, whenever a switch is hit in MPF, it posts an event *<switch_name>_active* . (A second event called *<switch_name>_inactive* is also posted when the switch opens back up.) To give the player points when a switch is hit, add sub-entries to the `variable_player:` section of your config file, with some switch name followed by "_active", like this:

```
##! mode: base
variable_player:
  s_right_inlane_active:
    score: 100
  s_left_flipper_active:
    score: 1000
```

Now save your config, start a game (`S`), hit the `L` key to launch a ball, then hit the `Q` key to trigger the right inlane switch . You should immediately see a score of 100 points. Then if you hit the `Z` key for the left flipper, you'll see the player's score increase by 1000 points. You can hit it as many times as you want to see the score increase:

Remember from the previous step that the `slide_player:` section of the config contains a text widget with a value of `(score)` in parentheses, and any values in parentheses are updated automatically when the underlying player variable changes. So that's how the display is updating automatically here.

By the way, there's a *reference list of many built-in events* in the documentation, so you can browse through that to get an idea of the various types of events that exist which you can use to trigger display slides or score events.

Note that `variable_player:` events in a mode's config file are only actually active when that mode is active. So the section we're adding in this step is in the base mode's config, which we've set to start any time a ball starts. But if the base mode ever wasn't running, then the `s_right_inlane_active` and `s_left_flipper_active` events wouldn't trigger a score.

When you create more modes in the future, you can actually configure that a score event in a higher-priority mode "blocks" the variable_player/scoring event in a lower-priority mode. So you could have a pop bumper that is worth 100 points in a base mode, but then you could also make it worth 5,000 points in a super jets mode while blocking the 100 point score from the base mode since if the scoring from both modes was active, you'd get two scoring events–the 100 from the base mode and the 5,000 from the super jets mode. (More on that later.)

Later on you can also configure *shots* which can control lights and manage sequences of switches and lots of other cool things, so that's how you can track the ball moving left-to-right or right-to-left around a loop, and from there you'll be able to configure different scoring events for each direction. (Again, we'll get to this later. For now you can just wire up scoring to a switch to see it working.)

4. Play with more player variables

As we said, you can add or subtract value from any player variable via the `variable_player:` section–even player variables that you make up.

For example, try changing your scoring section to this:

```
# we will initially set the value to 0 when the machine starts up
player_vars:
  potato:
    initial_value: 0

##! mode: base
# in your base mode (modes/base/config/base.yaml)
variable_player:
  s_right_inlane_active:
    score: 100
  s_left_flipper_active:
    score: 1000
    potato: 1
  s_right_flipper_active:
    potato: -2
```

We use the word "potato" here to illustrate that player variables can be anything. So now when the left flipper is active, the player variable called "score" will increase by 1000, and the player variable called "potato" will increase by one. (If you make a reference to a player variable that hasn't been defined before, it will automatically be created with a value of 0.)

Also notice that when the right flipper is hit, the player variable called "potato" will have a value of 2 subtracted from it.

Player variables exist and are tracked even if they're not displayed anywhere. So if you run your game now and start flipping, the potato value will change. Again, player variables are stored on a per-player basis, so if you start adding additional players to the game, they'll each have their own copies of their own player variables. Also the player variables are destroyed when the game ends. (It is possible to save certain variables from game-to-game, but we'll discuss those later, as those are not player variables.)

So now that we're tracking this potato variable, let's add it to the display. To do this, let's add another widget to the slide that is show when the base mode starts. (So we're going to be editing `<your_machine>/modes/config/base.yaml` again. Add the potato text entry, like this:

```
##! mode: base
# in your base mode (modes/base/config/base.yaml)
slide_player:
  mode_base_started:
    widgets:
      - type: text
        text: (score)
        number_grouping: true
        min_digits: 2
        font_size: 100
      - type: text
        text: PLAYER (number)
        y: 10
        x: 10
```

(continues on next page)

(continued from previous page)

```
        font_size: 50
        anchor_x: left
        anchor_y: bottom
      - type: text
        text: BALL (ball)
        y: 10
        x: right-10
        anchor_x: right
        anchor_y: bottom
        font_size: 50
      - type: text
        text: 'POTATO VALUE: (potato)'
        y: 40%
```

Notice that we put `text: 'POTATO VALUE: (potato)'` in quotes. That's because we actually want to show the colon as part of the text that's displayed on the screen. However colons are important in YAML files. So if we made our entry like this: `text: POTATO VALUE: (potato)`, then we would get a YAML processing error because the YAML processor would freak out. "OH MY THERE ARE TWO COLONS?? WHAT'S THIS MEAN??? <crash>"

So we use quotes to tell it that the second colon is just part of our string.

Now you can run your game (via `mpf both`), S to start a game, L to launch a ball, then use the Z and / keys to left and right flip which will adjust the potato value accordingly.

Notice that when you first start a game, the onscreen text says `POTATO VALUE: (potato)`. That's because when this slide is first displayed, there is no player variable called "potato"–it's not created until you hit a flipper button–so the text widget doesn't know what to do with "potato", so it just prints it as is. Later we'll learn how to properly initialize variables, but the main thing for now is to see how the scoring and slide player works.

Check out the complete config.yaml file so far

If you want to see a complete `config.yaml` file up to this point, it's in the `mpf-examples/tutorial/step_15` folder with the name `config.yaml`. You can run it be switching to that folder and running `mpf both`:

```
C:\mpf-examples\tutorial_step_15>mpf both
```

Tutorial step 16: Create an attract mode display show

Now that we have a running game and some basic scoring, let's continue to make the display more useful by creating a slide show that plays during the attract mode and cycles through a few different slides. ("GAME OVER", "PRESS START", . . . that sort of thing.)

1. Create an attract mode folder structure

So far it looks like your game only has one mode. (The *base* mode you created a few steps ago.) But MPF actually has a few built-in modes that it uses to do its thing. For example, there's a mode called "attract" which runs the attract mode (including watching for the start button press to start a game), and there's a mode called "game" which actually runs your games. (You may have noticed these modes in your logs. *Attract* runs at priority 10 and *game* runs at priority 20.)

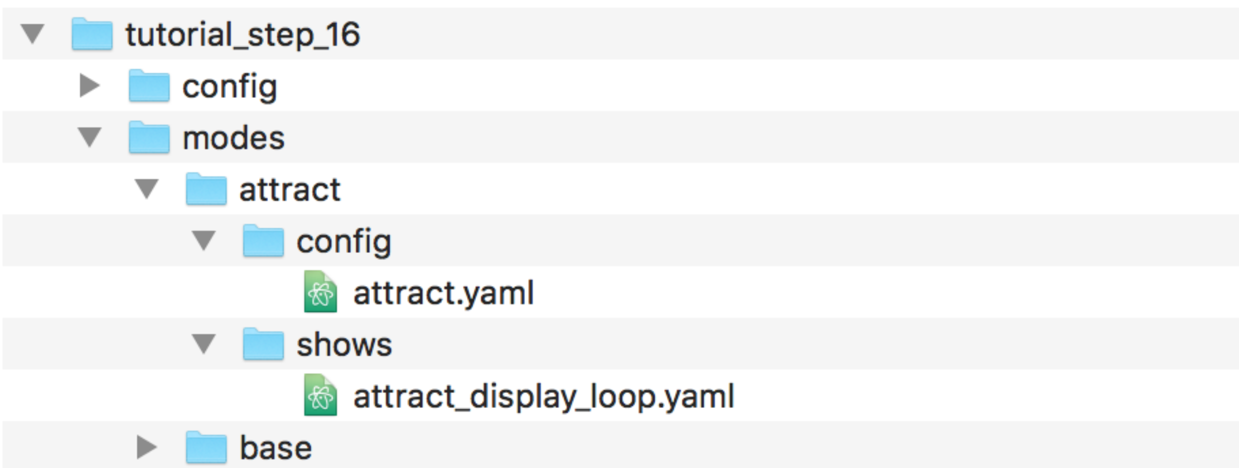
Even though the attract mode is built-in, you can still create an attract mode folder and an attract mode config which enable you to extend the attract mode for your own use. So let's do that now.

Go into your machine's `/modes` folder (which should only have your base folder in it) and create a new folder called `attract`. Now you should see two folders in your machine's `/modes` folder, the base folder and the attract folder.

Now create a `/config` folder in your attract folder, and then create a new config file called `attract.yaml`. So the attract folder is pretty much just like the base folder, with the file `attract.yaml` used to control the settings that will be used when the attract mode is active.

Finally, create a folder called `/shows` in your new attract mode folder, and inside that folder, create a new file called `attract_display_loop.yaml`.

Your new machine folder structure should look like this:



2. Edit your show yaml file

MPF has the ability to run "shows" which are coordinated series of lights, sounds, slides, flashers, images, videos, etc. These show files also use the `.yaml` file format, though they're different than the yaml config files. You can name the show whatever you want. In this case we called it `attract_display_loop.yaml` since that pretty much describes what it does.

Note that we put this show file in a folder called "shows" in our attract mode folder. Technically you can play any show from any mode (and you could add a machine-wide `/shows` folder if you want), but we prefer to add the shows used by a mode inside that mode's `/shows` folder since it keeps everything from one mode together.

Here's a complete sample `attract_display_loop.yaml` file you can use as a starting point:

```
#show_version=5
##! show: attract_display_loop

- duration: 3s
  slides:
    awesome_slide:
      widgets:
```

(continues on next page)

(continued from previous page)

```
      - type: text
        text: YOU ARE AWESOME
        font_size: 50
      transition:
        type: push
        duration: 1s
        direction: left

- duration: 3s
  slides:
    press_start:
      widgets:
      - type: Text
        text: PRESS START
        animations:
          pre_show_slide:
          - property: opacity
            value: 0
            duration: .5s
          - property: opacity
            value: 1
            duration: .5s
            repeat: false
      - type: Text
        text: FREE PLAY
        color: green
        y: 10
        anchor_y: bottom
      transition:
        type: move_in
        duration: 1s
        direction: right

- duration: 3
  slides:
    mission_pinball:
      widgets:
      - type: Text
        text: MISSION PINBALL
        color: red
      transition:
        type: move_in
        duration: 1s
        direction: top

- duration: 3
  slides:
    last_game_score_slide:
      widgets:
      - type: text
        text: LAST GAME
        font_size: 50
        y: 60%
      - type: text
```

(continues on next page)

(continued from previous page)

```
        text: (machine|player1_score)
        number_grouping: true
        min_digits: 2
        font_size: 50
        y: 40%

##! mode: attract
```

First, notice the first line is `#show_version=5`. This is similar to the config_version in config files, except since this is a show file, it's "show_version".

Next, notice that the show file is broken into steps, each beginning with a dash and then a `duration:` entry. The `duration:` entry controls how long each step is. The default unit for this value is seconds, so `duration: 3` is valid, though you can enter standard time strings like `duration: 3s` or `duration: 300ms`, etc.

By the way, when you play back a show, you can set the playback speed. So even though all the steps are 3 seconds long in our example show, when you play the show, you could (for example), set the playback speed to 2.0, and each step would be 1.5 seconds instead of 3 (since it's playing 2x as fast).

There's a whole section of documentation on *shows*, so review that at some point for all sorts of details about show files, formats, etc.

In addition to the `duration:` setting in each step, also notice that each step has a `slides:` setting. The format and content of the `slides:` section of a show is identical to the `slide_player:` section in a config file. (In the future you'll see this applies to other "players"; for example, `light_player:` in a config file is the same as `lights:` in a show, `sound_player:` in a config file is the same as `sounds:` in a show, etc.)

Then in the `slides:` section of each step, we've added a slide name. These slides are named `awesome_slide`, `press_start`, `mission_pinball`, and `last_game_score_slide` in the example above. The slide names don't really matter, but since none of these slides have been defined yet, we add a `widgets:` section to each one and define them here. (The slides are only created once, the first time they're displayed. After that they are kept in memory so they can be used over and over. They're only removed from memory when the attract mode stops.)

Also notice that we added `transition:` settings which control how one slide transitions to the next. Without transitions, the new slide appears instantly. But with transitions, we can make one slide move in from the side, or cross fade, etc.

The last slide deserves special mention - it displays the score of the previous game. Player variables such as score are only valid during a game and lose their value once the game ends. To allow access to the score of a previous game, MPF saves this player variable to a machine variable which can be accessed outside the running game. A discussion of this and other machine variables is found *here*.

3. Configure your show to play automatically

Now that you've created your show, we need to make it so it plays. In this case we want this show to play whenever the attract mode is running. To do this, go back to the config file for the attract mode (`<your_machine>/modes/attract/config/attract.yaml`) and add the following:

```
#config_version=5
##! show: attract_display_loop
```

(continues on next page)

(continued from previous page)

```
##! mode: attract

show_player:
  mode_attract_started: attract_display_loop
```

Note that we don't need a `mode:` section here because those settings are already configured in the default attract mode settings folder contained inside of MPF. So instead all we need to do is add a `show_player:` entry. Like the `slide_player:` we've used in the past, the `show_player:` section contains sub-sections for MPF events, and when that event is posted the shows underneath it are started.

In this case we're going to start the show when the *mode_attract_started* event is posted.

You can also use the `show_player:` section of a config to set events that stop shows, but shows that are started from modes automatically stop when that mode stops. (The beauty of mode-based configs!) So in this case, the `attract_display_loop` will automatically stop when the attract mode stops (which it does when a game starts).

4. Remove the attract mode stuff from your machine config

One last thing you should do here while you're at it is go back into the machine-wide config <your_machine>/config/config.yaml and remove the attract_started slide from the `slides:` section, and the `mode_attract_started` entry from your `slide_player:` section.

OLD machine-wide config (partial):

```
# old
slides:
  welcome_slide:
    widgets:
      - type: text
        text: PINBALL!
        font_size: 50
        color: red
      - type: rectangle
        width: 240
        height: 60
  attract_started:
    widgets:
      - text: ATTRACT MODE
        type: text

slide_player:
  init_done: welcome_slide
  mode_attract_started: attract_started
```

NEW machine-wide config:

```
slides:
  welcome_slide:
    widgets:
      - type: text
        text: PINBALL!
        font_size: 50
```

(continues on next page)

(continued from previous page)

```
      color: red
    - type: rectangle
      width: 240
      height: 60

slide_player:
  init_done: welcome_slide
```

The reason we remove this is because it's not necessary now that we have our new attract mode display show running.

Plus, even if you don't remove this entry, the original "ATTRACT MODE" text from the machine-wide config won't show up anymore. Why? Because the attract mode runs at Priority 10, and the machine-wide config is Priority 0. So the display show from the attract mode config will show on top of the slide from the machine-wide config, so we may as well remove the machine-wide one.

Now when you run your game via `mpf both`, you should see the attract mode display show. Then when you press Start (or the `S` key), everything else should proceed as it did before.

If you play through a complete game (3 balls), then when the game is over, you should see the attract mode display show start up again.

Check out the complete config.yaml file so far

If you want to see a complete `config.yaml` file up to this point, it's in the `mpf-examples/tutorial/step_16` folder. You can run it be switching to that folder and running `mpf both`:

```
C:\mpf-examples\tutorial_step_16>mpf both
```

Tutorial step 17: Add lights (or LEDs)

Now that you're able to run a complete (albeit boring) game, let's get your lights or LEDs configured and make it so they play a show while your machine is in attract mode.

If you're following this tutorial with virtual hardware, it's still worth doing this step because use can use *The MPF Monitor* to see your lights and LEDs in realtime against a picture of your playfield.

1. Understand "lights"

In MPF, "lights" refers to bulbs that are plugged into a lamp matrix, or to direct-connected LEDs (which are usually RGB). So lights might be either LEDs or lamps in a matrix. See *"Lights" versus "LEDs" (Some LEDs are lights?!?)* for details.

2. Add your lights/LEDs to your machine config file

Once you figure out whether you have lights or LEDs, you need to add the relevant section to your machine configuration file. There's probably not much to explain here. Adding lights is pretty similar to adding switches and coils.

See the relevant documentation for each for instructions how to enter them:

- *Lights*

In the following we assume that `l_light1` and `l_light2` exist. If you do not have lights with that name make sure to adjust all examples accordingly or you will run into issues.

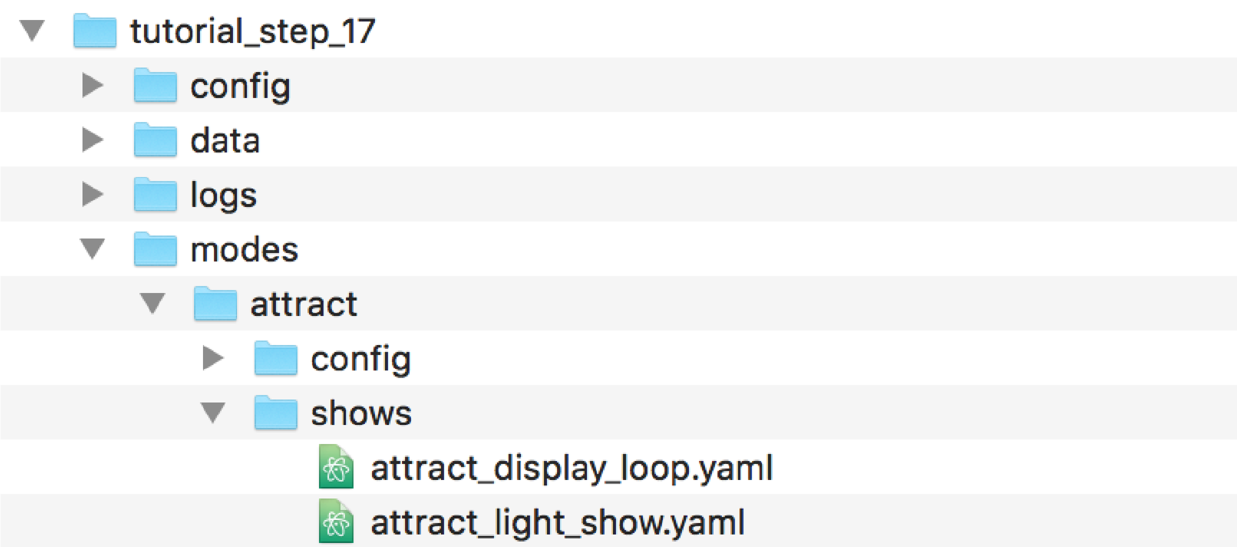
If you would like to see a fully working example you can take a look at the two example configurations in *LEDs*.

3. Create an attract mode light/LED show

Once you add your lights, you need a simple way to test them to make sure they're working. We typically throw together a quick attract mode light show so we can see some blinking lights as soon as MPF boots up.

The easiest way to create a complex series of light actions is with MPF's *show* functionality. This is the exact same type of show that we use for the display loop, except this time we configure lights for each step instead of slides.

So the first thing to do is to create another show file in your attract mode shows folders. Let's call this one `attract_light_show.yaml`. Your attract mode shows should now look like this:



Note that we started both of these file names with the word "attract". That is certainly not required and you can name them whatever you want. We find it's a bit easier to add the mode name so we can know which files are which when we have a bunch of files open in the editor at the same time.

4. Add some entries to your show

There are all sorts of things you can do with a light show file that you'll become familiar with as you get deeper into your game configuration. For now we're just going to create a simple show that cycles through three lights. We'll call them `l_light1`, `l_light2`, and `l_light3`, though there's a good chance that you don't have lights with those names in your machine so you'll have to change them to names that actually exist for you. If you have matrix lights, add entries to your `attract_light_show.yaml` file so that it looks something like like this:

```
##! show: attract_light_show
#show_version=5
- duration: 1
  lights:
    l_light2: 0
    l_light1: ff
- duration: 1
  lights:
    l_light1: 0
    l_light2: ff
- duration: 1
  lights:
    l_light2: 0
    l_light3: ff
- duration: 1
  lights:
    l_light3: 0
    l_light2: ff
```

Matrix lights don't have color setting since their color is determined by the color of the bulb and/or the color of the insert. So the `0` and `ff` values here just represent "off" (0) and "on" (255). If you look at the four steps in this show, you'll see the first step turns off *l_light2* and turns on *l_light1*, the next one turns *l_light2* and turns off *l_light1*, etc. In other words, if this show runs in a loop you'll get a never ending 1-2-3-2-1-2-3-2-1-2-3-2. . . pattern. If you have RGB LEDs, then you can have some more fun and actually specify different colors for each light at each step. For example, if you just wanted to have a show that cycled three RGB LEDs through the colors of the rainbow, you could create a show like this:

```
##! show: attract_light_show
#show_version=5
- duration: 1
  lights:
    l_led1: red
    l_led2: red
    l_led3: ff0000
- duration: 1
  lights:
    l_led1: ff6600
    l_led2: ff6600
    l_led3: ff6600
- duration: 1
  lights:
    l_led1: ffcc00
    l_led2: ffcc00
    l_led3: ffcc00
- duration: 1
  lights:
    l_led1: lime
    l_led2: 00ff00
    l_led3: 00ff00
- duration: 1
  lights:
    l_led1: blue
    l_led2: 0000ff
```

(continues on next page)

(continued from previous page)

```
    l_led3: 0000ff
- duration: 1
  lights:
    l_led1: ff00aa
    l_led2: ff00aa
    l_led3: ff00aa
```

Obviously this is just the very beginning of what you can do. You can create shows that are hundreds of steps involving dozens of lights. (Notice that if you don't specify a change for a particular light for a step then that light just stays at whatever it was before. In other words, you only have to enter the new values for the lights that change each step-—you don't have to enter all the lights from scratch every step.)

Again, notice that for the color of the LEDs, you can specify a color either in the form of a string name or a 6-digit hex color codes. If you go with names, you can use any of these colors.

5. Configure your show to play

This new show file is just like your existing display show, except this one contains settings for lights. So to get it to play, add it to the `show_player:` section of your attract mode config file, set to play on the *mode_attract_started* event just like the display show.

The only catch here is that the YAML file cannot have the same setting entered twice. (If you did this, the second one would overwrite the first one which would be really confusing. In fact if MPF sees that, MPF will exit and print a warning about the duplicate so you can fix it.)

MPF offers a way around this though, in that you can add a `.1` to the end of the event name, like this:

```
##! mode: test_mode
#config_version=5
show_player:
  mode_attract_started: attract_display_loop
  mode_attract_started.1: attract_light_show
```

Adding the `.1` doesn't really affect anything in terms of how this works, it just makes it so this is valid YAML and both entries get set. (And you can have more than one, `.2`, etc. In fact you can have any number, they don't have to be in order or anything.

You also might be wondering why we don't just make a single attract show and put the slides and lights in the same show?

Certainly that's possible, but we like to keep things separate, as this will let you start and stop them on their own, and it will make it easier to tweak things (like the playback speed) of one thing without breaking other things.

Save your files, and run your game. You should see your light show and the display show start playing once the attract mode starts up.

If you're using the virtual interface without a real pinball machine, this is probably a good time to use the *MPF Monitor* to see that the light show is actually working. (Expand the "light" or "LED" section in the devices window to see your lights and watch the colors cycle.

6. Speed things up

While it's cool that the show is working, it's kind of lame because it runs so slow with 1 second between steps. So let's speed it up.

You could go into your show and adjust the `duration:` of each step, but that's kind of a pain since you have to change every single step, and it makes it annoying when you're playing with different values.

Instead, we like to tweak the playback speed of the show which is something we can do in the `show_player:` entry. (In fact, we almost always use the duration values in shows as a sort of "relative" duration of one step to another, and then set the actual speed at play time.

So if we want each step to be 1/4th of a second, we need to play the show at 4x the speed. Simple, just add a `speed: 4` to the show_player entry.

```
#config_version=5
show_player:
  mode_attract_started: attract_display_loop
  mode_attract_started.1: attract_light_show
    speed: 4

# don't try this, it won't work
```

If you try to run MPF with the config above, MPF will halt with the following error (scroll to the right to see it all):

```
ValueError: YAML error found in file /mpf-examples/tutorial_step_17/modes/attract/config/attract.yaml.␣
↪Line 6, Position 10
```

What gives?

The problem is that entries in YAML files can be *either* setting names and values *or* section names with subsections, but not both. So in the example above, it sees `mode_attract_started.1: attract_light_show` as a setting name and value, but then it also sees `speed: 4` indented under it. The YAML processor doesn't know what to do?

To fix this, we need to make a slight change to our YAML file, like this:

```
##! mode: test_mode
#config_version=5
show_player:
  mode_attract_started: attract_display_loop
  mode_attract_started.1:
    attract_light_show:
      speed: 4
```

What we've done is moved the show name (`attract_light_show`) under the event name (`mode_attract_started.1`), and then we added the speed setting under there.

If you wanted to, you could consolidate the duplicate `mode_attract_started` entries like so:

```
##! mode: test_mode
#config_version=5
show_player:
  mode_attract_started:
    attract_display_loop:
```

(continues on next page)

(continued from previous page)

```
    speed: 1
  attract_light_show:
    speed: 4
```

Either option is fine, and you'll probably end up with both techniques scattered throughout your configs.

7. Configure more light shows to all run at once

The simple light show with two or three lights is a good first step, but it's hardly what could be considered a "real" attract mode light show. Unfortunately if you look at a real pinball machine, you might be overwhelmed by all the crazy light action. But if you really look closely, you'll realize that the super-complex looking light shows on real pinball machines are just lots of little shows all running at the same time.

For example, look at how we can break down the attract mode light show of *Demolition Man*:

https://youtu.be/_h_rhHExmX4

So if we were creating the attract mode light show like this for MPF, we would actually create lots of little shows each with just a few lights in them. Then we'd end up with a list of show files, like this:

- flipper_red_flashing.yaml
- purple_mode_sweep.yaml
- inlane_alternating.yaml
- random_flashing.yaml
- car_chase_sweep.yaml
- ramp_orbit_sweep.yaml
- right_orbit_sweep.yaml
- claw_sweep.yaml
- mtl_sweep.yaml
- center_ramp_sweep.yaml
- standups_sweep.yaml

Again, we'd make every step of every show have a duration of 1. Then in our `show_player:` configuration, we'd configure the list of shows to play when the attract mode starts instead of just one. For example:

```
show_player:
  mode_attract_started:
    attract_display_loop:
      speed: 1
    flipper_red_flashing:
      speed: 2
    purple_mode_sweep:
      speed: 4
    inlane_alternating:
      speed: 3
```

(continues on next page)

(continued from previous page)

```
      random_flashing:
        speed: 2
      car_chase_sweep:
        speed: 3
      ramp_orbit_sweep:
        speed: 5
...(truncated. you get the idea)
```

(If you were really duplicating the *Demolition Man* attract mode light show, you'd also want to implement a play list which plays sets of shows in timed sequences since the real machine does one thing with the lights for a few seconds, then another, etc.

Check out the complete config.yaml file so far

If you want to see a complete `config.yaml` file up to this point, it's in the `mpf-examples/tutorial/step_17` folder with the name `config.yaml`. You can run it be switching to that folder and running `mpf both`:

```
C:\mpf-examples\tutorial_step_17>mpf both
```

Tutorial step 18: Add your first shot

At this point you have a machine you can turn on, lights flash, the display works plays, you can hit start, you have a base mode with some simple scoring, and you can play complete games. Not bad! In this step we're going to introduce you to a key MPF concept called "shots", which is an important concept in MPF and something that you'll use *a lot* when you're putting together your game logic.

1. What's a shot?

First, take a look at the *introduction to shots* documentation to understand what a shot is.

2. Create your first shot

To define your a shot, you add a `shots:` entry to a config file in a mode, and then under there, you set the switch, timing, and other details that make up that shot.

You'd typically define your shots per mode, since the behavior differs depending on the mode. If you want a shot to be available in every mode you can also put them in the base mode which is usually active all the time.

Note: Before 0.30 you could define shots per machine-wide. This caused very complex configs and is no longer supported. You can put shots into your base mode if you want them to be active all the time during a game.

Let's start by creating our first shot in the base mode's config file (base.yaml).

```
##! mode: base
shots:
  my_first_shot:
    switch: s_right_inlane  # pick a switch that's valid in your machine
```

Depending on your machine, you might not actually have a switch called "s_right_inlane", so feel free to pick a different switch name. For now just keep it simple—a standup or a lane switch or something.

Also, to make following the tutorial easier, go ahead and call this shot "my_first_shot" even if you're using a different switch name. You can change the name of the shot to something more meaningful later.

Next, find the `variable_player:` section that you added in Step 15, and change the first entry from `s_right_inlane_active:` to `my_first_shot_hit`, like this:

```
##! mode: base
variable_player:
  my_first_shot_hit:  # this was s_right_inlane_active
    score: 100
  s_flipper_lower_left_active:
    score: 1000
    potato: 1
  s_flipper_lower_right_active:
    potato: -2
```

Do you understand what this is doing?

Remember that the variable_player section will add (or remove) value from a player variable when certain events happen. So the OLD entry from Step 15 would increase the score by 100 points when the event "s_right_inlane_active" happened, and the NEW entry changes that so the 100 points are added when the event "my_first_shot_hit" happens.

This illustrates something to know about shots: Whenever a shot is "hit", then an event is posted with the name of the shot plus "_hit" added onto it.

So in this case, the shot "my_first_shot" will post then event "my_first_shot_hit" whenever that shot is made.

If you save your changed config file and run MPF again, start a game with the `S` key, then hit the right inlane switch with the `Q` key, you should see the player's score increase by 100 points.

So it kind of looks like nothing really changed, except now we're using a real shot instead of scoring based on the switch entry.

At this point you might think that this is overly complicated. After all, everything worked fine before without having to mess with shots and all, so why bother?

Again, this is just a simple example to get you started. The real power of shots comes in as you define more complex shots, as you get into shot profiles (doing different things depending on the state of the shot), and enabling, disabling, blocking, and overriding shots based on different modes.

3. Change the shot profile

Every shot in MPF has a "shot profile" applied to it. (Since we didn't specify a profile in the shot we just created, it uses a default profile called, wait for it. . . "default".)

A shot profile is a list of steps (or states) for a shot. For example, the default profile (which is built-in to MPF) has two states:

1. unlit
2. lit

When a new game starts, the shots in MPF start at the first step of the profile. In other words, the shot called "my_first_shot" starts in the "unlit" state. Then, when the shot is hit, the profile is advanced to the next step. (So when "my_first_shot" is hit, that shot advances from the "unlit" to the "lit" state.)

You can apply the same profile to multiple shots (and the state of each shot is tracked separately), so if you have "my_first_shot" and "my_second_shot", they both start "unlit", but if you hit "my_second_shot", then it advances to "lit" but "my_first_shot" stays in the "unlit" state.

Shot profiles have all sorts of settings (which we'll get to in a bit), including options for what happens when the shot is hit when it's in the final state—does it just stay there or does it go back to the first state? (The built in "default" shot profile will stay in the lit state even if it's repeatedly hit.)

Also, tracking which state a shot is at is done on a per-player basis, so if Player 1 advances a shot from "unlit" to "lit", then when Player 2 starts, that shot will be back in the "unlit" state.

One of the cool things about shot profiles is you can tie them to shows, and then when you define your shots, you can specify how those shows are played. In other words, you can associate a light or LED with your shot, and then that light will be off when the shot is "unlit" and then turn on when the shot is lit.

Let's do that now.

3a. Associate a light/led with your shot

To do this, go back to the mode config where you defined the shot (base.yaml) and change the `shots:` section.

If you have LEDs in your machine, change it to this:

```
##! mode: base
shots:
  my_first_shot:
    switch: s_right_inlane
    show_tokens:
      led: led_1 # pick an LED that's valid in your machine
```

If you have a lamp matrix, change it to this:

```
##! mode: base
shots:
  my_first_shot:
    switch: s_right_inlane
    show_tokens:
      light: l_light_quick_freeze # pick a light that's valid in your machine
```

In either case, be sure to pick an LED or light name that is a valid light in your machine.

For now don't worry about what "show_tokens" is or what's happening. (We'll get to that.)

Save your config, then re-run MPF and start a game. The light or LED you picked should be off.

Now hit the switch for the shot. You should see the 100 point score increase, and you should also see the light or LED turn on. (If it's an RGB LED, it will turn on white. We can change that later.)

If you hit the switch again, you'll still get 100 points each time (since the "my_first_shot_hit" is happening each time), but the light won't turn off since the shot is staying in the "lit" state since the default shot profile isn't configured to go back to the first step when it gets to the last step.

3b. Create a custom shot profile

Next, let's create a custom shot profile that has more than the "lit" and "unlit" steps.

To do this, we'll add a section to the mode's config file (base.yaml) called `shot_profiles:`. Create that section now, and define a shot profile called "my_first_profile" with the following settings:

```
##! mode: base
shot_profiles:
  my_first_profile:
    states:
      - name: unlit  # step 1
        show: off
      - name: flashing  # step 2
        show: flash
      - name: lit  # step 3
        show: on
    loop: true
```

Take a look at this shot profile to see what's happening.

First, notice that in the `my_first_profile:` section, there's a subsection called "states". This is a list of all the states (steps) that shots will use when this profile is applied. (Note the dashes to separate each step.)

The states/steps are listed in the order they'll cycle through as the shot is hit.

Each step has a `name:` setting which is the name of the step (or, more accurately, the name of the state that shot is in when a shot with that profile applied to it is at the step).

Also notice that each step has a `show:` setting. This is the name of the MPF show (just like display show we created in Step 16 or the light show we created in Step 18). These shows need to be valid shows within MPF. In this case we're using shows named "off", "flash", and "on", as those are valid names for three shows that are built-in to MPF.

What's basically happening here is that when a shot with this profile applied is at the first step of the profile, the state name will be called "unlit" and the show called "off" will be played. Then when the shot is hit, it will advance to the next step, which is called "flashing" in this case. The show called "unlit" will be stopped, and then the show called "flash" will be played. If the shot is hit again, it will advance to the "lit" state, the "flash" show will stop, and the show called "on" will be started.

This shot profile also includes a `loop: true` setting that means when a shot is hit that's in the last step of the profile, it will loop back to the first step. (So hitting the shot when it's lit means the shot will loop back to "unlit".)

3c. Apply the new profile to the shot

Simply creating a shot profile doesn't mean that any shots use it. It just means that profile is available to be used, much like how creating a show is separate from playing the show.

So next we need to tell our shot that it should use the new profile we just created by adding a `profile:` setting.

```
##! mode: base
shots:
  my_first_shot:
    switch: s_right_inlane
    show_tokens:
      led: l_led1 # or use light: here, depending on your machine
    profile: my_first_profile
```

Save your config and re-run MPF. Once you start a game, the light or LED from your shot should be off. Hit the switch for the shot, and the light or LED should starting flashing. (It will be slow—1 second on, 1 second off.) Hit it again, and it should go on solid. Hit it again and the shot will go back to the "unlit" state. Hit it again and the light or LED should flash. Etc.

Note that you must actually start a game for this to work. Shots are only active when games are in progress, and the state is tracked per-player which means that players must exist, etc.

If you play a multi-player game, you should see that the state of that shot is maintained and restored separately for each player.

3d. Apply custom scoring based on state

Remember that the `scoring:` section of the base mode config scores 100 points each time that shot is hit. So as you're hitting the switch over and over to cycle through the states, each time you do that the player gets 100 points.

That scoring entry is based on the `my_first_shot_hit`, which is generated every time that shot is hit since shots make events in the form `<shot_name>_hit`.

However, each time a shot is hit, there's two ADDITIONAL events posted which are `<shot_name>_<profile>_hit` and `<shot_name>_<profile>_<state>_hit`.

For example, when you start a new game with the shot and shot profile we've been working with, when you hit the switch for that shot, three shot-related events will be generated:

- my_first_shot_hit (shot + "hit")
- my_first_shot_my_first_profile_hit (shot + profile + "hit")
- my_first_shot_my_first_profile_unlit_hit (shot + profile + state + "hit")

When you hit that same shot a second time, the following three events will be generated: The first two are the same since they're based on shot name and profile name, but the last one is different because the shot's state is different.

- my_first_shot_hit (shot + "hit")
- my_first_shot_my_first_profile_hit (shot + profile + "hit")
- my_first_shot_my_first_profile_flashing_hit (shot + profile + state + "hit")

Hitting that shot again will generate the following three events:

- my_first_shot_hit (shot + "hit")
- my_first_shot_my_first_profile_hit (shot + profile + "hit")
- my_first_shot_my_first_profile_lit_hit (shot + profile + state + "hit")

And so on. . .

Now let's look at how we can give the player a different number of points when they hit that shot depending on what state the shot's in.

Here's the existing variable_player section from the base mode config:

```
##! mode: base
variable_player:
  my_first_shot_hit:
    score: 100
  s_flipper_lower_left_active:
    score: 1000
    potato: 1
  s_flipper_lower_right_active:
    potato: -2
```

Again, the player gets 100 points each time that shot is made regardless of what state it's in since the scoring event is the generic shot hit event which does not include details of what state the shot is in.

Now let's change the variable_player section to this:

```
##! mode: base
variable_player:
  my_first_shot_my_first_profile_unlit_hit:
    score: 100
  my_first_shot_my_first_profile_flashing_hit:
    score: 1000
  s_flipper_lower_left_active:
    score: 1000
    potato: 1
  s_flipper_lower_right_active:
    potato: -2
```

We changed the name of the event for the first variable_player entry from "my_first_shot_hit" to "my_first_shot_my_first_profile_unlit_hit". This means those 100 points will only be added if that shot is hit while it has the "my_first_profile" applied AND while that profile is in the state "unlit".

The next entry, for 1000 points, will only be called when that shot is hit with "my_first_profile" applied while it's in the state "flashing".

Save your config and run your game. If you hit the switch for the shot, you should get 100 points and the light should start flashing. Hit it again, and you should get 1000 points and the light should turn on steady. Hit it a third time, and you should get no points, but the light will also turn off since the profile is set to loop and it will go back to the first (unlit) state.

In other words, hitting the `Q` key (or the actual switch if you have a real machine) should result in the following sequence of total score (one for each hit): 100, 1100, 1100, 1200, 2200, 2200, 2300, 3300, 3300. . .

4. Add a second mode and score the shot from there

One of the most powerful features of shot profiles is that shots can have multiple profiles defined at the same time (with each active mode having the ability to apply its own profile).

To illustrate this, we're going to create a new mode, called "mode2". So go ahead and create a `mode2` folder in your `modes` folder, then add the `config` folder into that folder, and then create the `mode2.yaml` mode configuration file for that mode.

Open up the `mode2.yaml` file and add the following lines. (We'll explain them step-by-step next.)

```
##! mode: mode2
#config_version=5
# mode2 config file

mode:
  start_events: mode2_start
  stop_events: mode2_stop
  priority: 200

widgets:
  mode2_start_banner:
    type: text
    text: MODE 2 STARTED
    font_size: 50
    color: lime
    y: 80%
    expire: 1s

widget_player:
  mode_mode2_started: mode2_start_banner

variable_player:
  my_first_shot_hit:
    score: 1
```

Remember that you also have to go back into your machine-wide config file to add the new `- mode2` entry to your `modes:` section. While we're in there, let's also add `keyboard:` entries for some events we can use to stop and start the mode.

Here are changes you'll make to the machine-wide config file:

```
# from the machine-wide config.yaml file

modes:
  - base
  - mode2

...

keyboard:  # existing keyboard entries not shown.
  n:
    event: mode2_start
  m:
    event: mode2_stop
```

Now save your files and run your machine. Then press the following keys:

- `S` - starts the game
- `Q` - hits your shot, score jumps to 100
- `Q` - hits your shot, score jumps to 1100

- N - starts mode2. You should see a 1-second green message showing this
- Q - hits your shot, score jumps to 1101
- Q - hits your shot, score jumps to 1202

You can press M to stop mode2 (though there is no on-screen message) and then continue to hit Q and notice the score jumps through the [+100, +1000, 0] cycle over and over.

You can press N again to start mode2 and notice that every time you press Q, you the score increases +1 (in addition to the [+100, +1000, 0] from the base mode.

Press M to stop mode2 again and notice that the +1 scoring stops.

So what's happening here?

First, notice that in the mode2.yaml file, we configured the following variable_player entry:

```
##! mode: mode2
variable_player:
  my_first_shot_hit:
    score: 1
```

Notice that that variable_player entry is just based on "my_first_shot" being hit. It does not contain any of the profile or state information in it, which means that it will always score the +1 regardless of the state of that shot.

Of course even while mode2 is running, the base mode is also running. That means that when both modes are running, mode2 is always scoring +1 per hit, and the base mode is cycling through the [+100, +1000, 0] scoring depending on what state the shot is in.

When you stop mode2 (with the M key), that removes the scoring from mode2, but since the base mode is still running, you still get the scoring from there.

5. Configure a new shot profile in mode2

In the previous step, we added a new mode and accessed the shot from within that mode, but that new mode still used the same shot profile as the base mode.

However, it's also possible to create a brand-new shot profile in a mode that will be applied to the shot when that mode is active.

This is useful if you want to "override" a shot profile from a lower mode based on a higher priority mode. For example, maybe you have a stand-up target in your base mode that you're using for some basic scoring. But then in a jackpot mode, you want that target to flash a light instead of just the regular on/off behavior from the base mode. You would do this by applying a different shot profile in the jackpot mode.

To illustrate this, open up your mode2.yaml file and:

1. Updated the `variable_player:` section from the example below
2. Add the `shots:` section from below
3. Add the `shot_profiles:` section from below

```
##! mode: mode2
# snippet from mode2.yaml
variable_player:
```

(continues on next page)

(continued from previous page)

```
  my_first_shot_mode2_flashing_hit:
    score: 10000
  my_first_shot_mode2_lit_hit:
    score: 100

shots:
  my_first_shot_mode2:
    switch: s_right_inlane
    profile: mode2

shot_profiles:
  mode2:
    states:
      - name: flashing
        show: flash
        speed: 5
      - name: lit
        show: on
    loop: false
    block: true
```

Save your files and run your game again, pressing the following keys:

- `S` - starts the game
- `Q` - hits your shot, score jumps to 100,
- `Q` - hits your shot, score jumps to 1100
- `N` - starts mode2. You should see a 1-second green message showing this
- `Q` - hits your shot, score jumps to 11,100
- `Q` - hits your shot, score jumps to 11,200
- `Q` - hits your shot, score jumps to 11,300
- `M` - stops mode2
- `Q` - hits your shot, no score change
- `Q` - hits your shot, score jumps to 11,400
- `Q` - hits your shot, score jumps to 12,400

Let's deconstruct the changes to the `mode2.yaml` config file too see what's going on.

First, notice that we added a `shots:` section and then added "my_first_shot" to it, like this:

```
##! mode: mode2
shots:
  my_first_shot:
    profile: mode2
```

However, unlike the "my_first_shot" entry in the base mode config, in the mode2 config we did NOT redefine the `switch:` or `show_tokens:` entries. Instead, we just added the `profile:` setting and told it to use a profile called `mode2`.

So what this means is that we're not creating a new shot or changing the configuration of the shot, rather, we're just saying that when mode2 is active, we want to apply a different shot profile to the shot. (Remember that settings from mode configuration files are only active when that mode is active.)

Next, take a look at the `shot_profiles:` section:

```
##! mode: mode2
shot_profiles:
  mode2:
    states:
      - name: flashing
        show: flash
        speed: 5
      - name: lit
        show: on
    loop: false
    block: true
```

In this case, we defined a profile called `mode2` which has two states: "flashing" and "lit". (These state names could be whatever you want, "incomplete" and "complete" or whatever.) Note also that we added `speed: 5` to the flashing step. That setting will be applied to the "flash" show when it's played, and you can use any of the *show_player:* settings there. In this case that will play the show at 5x speed, so we'll see a very fast flashing.

Also note that we added `block: true` to this profile. That means that when this profile is active, any shot profiles from lower priority modes will be disabled. Since mode2 runs at priority 200, the profile "my_first_profile" which we assigned in the base mode config (base.yaml) will be blocked.

And, since the variable_player events in the base mode are based on the shot being hit with the "my_first_profile" applied, this is why when mode2 is running, we don't get the variable_player events from the base mode. Those events are not posted because my_first_profile is not active because the higher priority profile attached to the shot in mode2 is blocking it.

If you were to remove the `block: true` from the mode2 profile in the mode2 config, then when you hit the shot while mode2 was active then you would get the scoring from both the base mode and mode2 mode applied.

(not done writing yet. . .)

Next steps to write

- Show tokens
- Shot groups
- advancing shots
- shot reset events

Check out the complete config.yaml file so far

If you want to see a complete `config.yaml` file up to this point, it's in the `mpf-examples/tutorial/step_18` folder. You can run it be switching to that folder and running `mpf both`:

```
C:\mpf-examples\tutorial_step_18>mpf both
```

Even if you have real hardware, it's probably worth running the MPF Monitor which will show you the events as they're posted that correspond to the shot being hit and it changing profiles.

Tutorial step 19: Testing your machine

Before you continue with your machine, we want to take a moment to let you know about MPF's automated testing features.

One of the cool things about MPF is that you can write "tests" which actually launch and run MPF and your machine config and then check to make sure everything is alright. These tests can hit switches and check to make sure that coils fired, or that lights are the right color, or that a certain mode is running, or that certain text is on the display, etc.

What's great about these tests is that they're easy to write, so you can write them bit-by-bit as you're creating your MPF config files. Eventually you'll have tests that cover hundreds of little things, and you can run them every time you change something in your config. Then down the road when your config is very advanced, you might be changing something in one area that accidentally breaks something else. (Maybe a mode doesn't stop properly so an unrelated playfield light is the wrong color.) Without tests, you might only find the bug after hours of play, but with the tests, you'll know immediately that something isn't right.

The only "catch" with the tests is that they're written in Python, so you have to learn a little Python to be able to use them. If you don't want to worry about tests right now because you're just learning MPF or just getting started, that's fine. No problem! But we wanted to make sure that you knew that these automated tests were available.

We have a tutorial which explains how to write tests on our developer site which follows this general tutorial (that you're reading now) 1-to-1. In other words, Step 2 in the MPF tutorial created an empty config file and got MPF up and running with the attract mode active, and Step 2 in the test writing tutorial shows how to write a test that verifies everything is ok.

In fact we have tests for every step in the tutorial in the MPF Examples repository. (That's what's in the "tests" folder in each step's machine folder.) You can even run the tests yourself (even if you don't know Python or don't know how to write tests) to verify that the config files you typed in are entered correctly.

More information about writing unit tests for your machine, as well as the test writing tutorial, is available here: http://developer.missionpinball.org/en/dev/testing/writing_machine_tests.html.

Overview video about testing your machine:

https://youtu.be/DPX-1FbBaYM

Tutorial step 20: Next steps

So you got a basic running machine. Where to go next?

- Setup the *MPF monitor* for faster development and light show testing
- Try the *service cli* to debug all kinds of stuff
- Configure your remaining *mechs*.
- Add more *game logic* (more modes)

Video about how to structure your modes:

https://youtu.be/JLgeGBc03bM

Video about developing your game without hardware:

https://youtu.be/7XmIIhzEREk

CHAPTER 6

MPF compatible control systems / hardware

MPF controls a pinball machine by interfacing to a modern pinball control system. (See the *MPF Overview* for details.) MPF itself is hardware-independent, meaning that MPF (and the configs and code you build) runs on a *normal/embedded PC* and can work with lots of different kinds of control systems and hardware devices.

Not only does this give you a choice of what type of pinball control hardware you want to use, it also means that you have the flexibility to change your hardware at any time without having to change any game code. You could even release a game code update that works on multiple platforms—all with the same code!

Here's a demo video of us switching out a P-ROC controller for a FAST controller in 3 minutes and running the same game code on both:

https://youtu.be/_Zw_cHw2CXY

It's possible to mix-and-match multiple types of hardware in a single MPF machine config. For example, you could combine the SmartMatrix RGB DMD with a FAST Core controller, or a FadeCandy LED controller with a P-ROC, etc. (You can even mix-and-match platforms within the same type of device, meaning you could have some LEDs attached to a FAST Pinball controller and others attached to a FadeCandy. See the *Mixing-and-Matching hardware platforms* guide for details.)

MPF currently supports the following hardware control systems. We are always adding more, so if there's a hardware device that you'd like to use that we don't support, let us know. (Or better yet, write your own interface to it and submit a pull request to the MPF codebase!)

Also see our guide on *voltages found in a pinball machine*.

List of supported control systems & hardware

Here's a list of all the different types of control systems and hardware that MPF currently supports. If there's a type of hardware you'd like us to support that you don't see on this list, please post a message to the MPF Users Google Group and we'll go from there.

Primary control systems

You'll need to pick one of these three as the main interface between MPF and your pinball machine.

- ***Open Pinball Project (OPP) controllers***
  - Gen 2 OPP hardware, with many combinations of wing boards for drivers, switches, switch matrix, LEDs & incandescent lights
  - *CobraPin Pinball Controller*
- ***Arduino Pinball Controller (APC)***
  - *New in MPF 0.53*
  - System 3 to System 11c
  - Segment displays
  - External sounds
  - Switches, rules and coils
  - Lights and enable triggers
- ***LISY***
  - *New in MPF 0.50*
  - Gottlieb System 1 (LISY1)
  - Gottlieb System 80 (LISY80)
  - Bally and Stern Games manufactured from 1977 to 1985 (LISY35) *New in MPF 0.53*
  - Segment displays
  - External sounds
  - Switches, rules and coils
  - Lights and enable triggers
- ***Multimorphic***
  - P-ROC with PDB driver boards (PD-16, PD-8x8, PD-LED)
  - P-ROC in all supported existing machines (Williams, Stern, etc.)
  - P3-ROC with PDB driver boards (PD-16, SW-16, PD-LED)
  - Plasma & LED mono DMDs (P-ROC)
  - Accelerometer-based tilt (P3-ROC)
  - I2C slave boards (see below for which I2C boards are supported) (P3-ROC)
  - Alphanumeric displays via aux port (P-Roc)
- ***FAST Pinball***
  - Core Controller, Nano Controller, WPC Controller
  - 0804, 1616, 3208 I/O Boards
  - Servo controller daughter board
  - Power Filter Driver Board coin-door interconnect

- Plasma & LED mono DMDs (Core & WPC controllers)
- FAST RGB LED-based DMD

- ***Stern SPIKE / SPIKE 2 machines***
  - *New in MPF 0.33*
  - A computer running MPF can directly connect to a SPIKE machine with a simple "USB to serial" converter which you plug into the SPIKE main board.
- ***Penny K Pinball PKONE Platform***
  - Nano Controller
  - PKONE Extension (switches, coils, rules, servos)
  - PKONE Lightshow (simple LEDs, WS281x RGB/RGBW LEDs)
- ***Virtual (software-only) controllers***
  - MPF includes virtual hardware interfaces you can use to run MPF when it's not connected to physical hardware. (This is good for working on your game when you're not around your machine, or if you don't have real hardware yet.)
  - You can also integrate MPF with a *Virtual Pinball (VPX)* table to play your game with simulated hardware.
  - The *MPF Monitor* is a graphical tool you can also use to visually interact with MPF which is especially useful if you're not using MPF with physical hardware.

Additional supported hardware

The following hardware devices can be combined with primary control sytstems to provide additional functionality.

- ***Snux System 11 driver board***
  - Supported in combination with the P-ROC or FAST WPC controller
  - Supported for System 11, 11A, 11B, 11C
  - Should work in Data East machines too, though it's never been tried
- ***I2C Servo Controllers***
  - Servos connected to I2C-based servo controllers
- ***Fadecandy RGB LED controllers***
  - 512 RGB LEDs per Fadecandy
  - Can connect multiple Fadecandys to support more LEDs
- ***Pololu Maestro servo controllers***
  - Supports up to 24 servos per board
- ***SmartMatrix RGB LED display controller***
  - Supports a "real" color DMD made up of RGB LED matrix
- ***RGB.DMD RGB LED display controller***
  - Supports a "real" color DMD made up of RGB LED matrix

- ***MyPinballs Segment Display Controller***
  - *New in MPF 0.50*
  - Alphanumeric segment displays
  - Also supports TNA Numeric Score Displays
- ***Light Segment Displays***
  - Control segment displays via light outputs or driver on another platform
  - BCD segment displays
  - 7-segment displays
  - Serial driven displays
  - RGB segment displays
  - Other formats and custom built displays
- ***Trinamics StepRocker***
  - *New in MPF 0.50*
  - StepRocker stepper controller
- ***Raspberry Pi***
  - *New in MPF 0.50*
  - Local (MPF on the RPi) or remote via ethernet
  - All inputs and outputs
  - I2C and SPI
- ***PIN2DMD RGB DMD***
  - *New in MPF 0.54*
  - 128x32 or 192x64 RGB LED DMD
  - Connected via USB
- ***Native I2C on Linux***
  - *New in MPF 0.50*
  - I2C devices on any nativ I2C bus
- ***MMA8451-based accelerometers***
  - *New in MPF 0.50*
  - Connected to I2C
- ***Pololu Tic***
  - *New in MPF 0.52*
  - Stepper controller connected to USB
- ***Open Sound Control (OSC)***
  - Control lights via OSC (i.e. your DMX controller)
  - Receive incoming switch changes (i.e. from your MIDI keyboard)

- Receive incoming events (i.e. from your MIDI keyboard)
- Send events to OSC (to generate sounds or trigger actions)

There is a *hardware roadmap* for other hardware which we want to support in the future.

Video on how platforms work internally and how to implement them:

https://youtu.be/PvQVoUzL8Cc

Configuration Guides

We have configuration guides which show you how to setup and use different types of pinball mechanisms with the various control systems and hardware that MPF supports:

How to configure Open Pinball Project (OPP) hardware for MPF

This how to guide explains how to set up your MPF configuration files to interface with an Open Pinball Project (OPP) pinball controller.

This page is about the software side of things. Hardware and electrical engineering stuff is documented at the OPP section in the pinballmakers.com Wiki.

Overview video about OPP:

https://youtu.be/WU98MRDeYeQ

Connecting OPP to your computer

Connect the OPP board to your computer via USB. Make sure that your OPP chains do not get too long since the serial throughput is limited per chain. You can connect multiple chains.

Verify Connected Boards via mpf hardware scan

You can run *mpf hardware scan* to see all connected node boards:

```
$ mpf hardware scan

Connected CPUs:
 - Port: com1 at 115200 baud
 -> Board: 0x20 Firmware: 0x10100
 -> Board: 0x21 Firmware: 0x10100

Incand cards:
 - CPU: com1 Board: 0x20 Card: 0 Numbers: [16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
↪31]

Input cards:
 - CPU: com1 Board: 0x20 Card: 0 Numbers: [0, 1, 2, 3, 8, 9, 10, 11, 12, 13, 14, 15]
 - CPU: com1 Board: 0x21 Card: 1 Numbers: [0, 1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
↪18, 19, 20, 21, 22, 23, 24, 25, 26, 27]
```

(continues on next page)

(continued from previous page)

```
Solenoid cards:
 - CPU: com1 Board: 0x20 Card: 0 Numbers: [0, 1, 2, 3]
 - CPU: com1 Board: 0x21 Card: 1 Numbers: [12, 13, 14, 15]

LEDs:
 - CPU: com1 Board: 0x21 Card: 1
```

If your boards do not show up checkout our *OPP troubleshooting guide*.

On Linux: Blacklist cytherm module

If you are using OPP hardware on linux you should blacklist the cypress thermometer because it conflicts with OPP.

In `/etc/modprobe.d/blacklist.conf` add:

```
blacklist cytherm
```

If blacklist.conf does not exist, just create a new empty file as root. Afterwards, reboot your PC.

On Linux: Add udev rules to ensure persistent device names

If you have more than one ttyACM connected to your PC (e.g. multiple OPP chains or other USB-serial adapters) you can assign a name to your ports based on the USB port they are connected to.

First identify the port of your OPP hardware. Usually it should be `/dev/ttyACM0` or `/dev/ttyACM1`.

Then run `udevadm info` on your port:

```
udevadm info /dev/ttyACM0
```

This will show you the `DEVPATH`. Now replace the last part `ttyACMX` with an asterisk and add an udev rules like this in `/etc/udev/rules.d/opp.rules`:

```
SUBSYSTEM=="tty", ACTION=="add", DEVPATH=="/devices/pci0000:00/0000:00:14.0/usb1/1-4/1-4:1.1/*",␣
↪SYMLINK+="ttyOPP1", GROUP="adm", MODE="0660"
```

After a reboot you should get a `/dev/ttyOPP1` device if you connect an OPP device to that specific USB port. You can use that port in your config.

On Ubuntu: Stop ModemManager

ModemManager tries to initialise all `/dev/ttyACMxx` devices as modem. That might cause delays after attaching OPP hardware and might also leave the hardware in a weird state with garbage on the bus. If you do not use any modems just disable and stop ModemManager:

```
sudo systemctl disable ModemManager
sudo systemctl stop ModemManager
```

What if it did not work?

Have a look at our *OPP troubleshooting guide*.

Configuring your machine for OPP

Related Config File Sections
hardware:
opp:

1. Configure the Hardware platform for OPP

To use MPF with OPP, you need to configure your platform as *opp*, like this:

```
hardware:
  platform: opp
```

2. Configure the OPP-specific hardware settings

When you use OPP hardware with MPF, you also need to add an `opp:` section to your machine-wide config which contains some OPP- specific hardware settings. MPF's default config file (*mpfconfig.yaml*) contains enough default settings to get you up and running. The only thing you absolutely have to configure is your ports.

Understanding OPP hardware ports

Even though OPP controllers are USB devices, they use "virtual" COM ports to communicate with the host computer running MPF. On your computer, if you look at your list of ports and then plug-in your OPP controller, you will see a new port appear. The exact names and numbers of these ports will vary depending on your computer and what else you've plugged in in the past.

Note: USB to serial converters add latency when communicating between the host computer, and the target device. It probably will not matter, but if given the choice between a "real" serial port, and a USB-serial port converter, the "real" serial port will have less latency. The real serial port must use 5V signal levels when talking to OPP hardware.

Adding the port to your config file

If you're using an OPP controller, you need to add the serial port to your MPF config. So if you plug in the OPP controller and see a port such as *COM7* appear, you'd set your config like this:

```
opp:
  ports: COM7
```

Full details of the port options as well as the other options available here are in the `opp:` section of the configuration file reference. Note that if you're using Windows and you have COM port numbers greater than 9, you may have to enter the port names like this: `\\.\COM10` `\\.\COM11` `\\.\COM12`, etc.

(It's a Windows thing. Google it for details.) That said, it seems that Windows 10 can just use the port names like normal: `com10`, `com11`, `com12`, so try that first and then try the alternate format if it doesn't work. On Linux, the port usually is `/dev/ttyACM0` or `/dev/ttyACM1`. On Mac, look for some `/dev/cu.modemXXXX` device.

What if it did not work?

Have a look at our *OPP troubleshooting guide*.

OPP Switches

Related Config File Sections
switches:

For switches, you can use most of the settings as outlined in the `switches:` section of the config file reference. There are only a few things that are OPP-specific:

Number:

OPP switches are numbered sequentially depending on which wing board is the switch input. Wing position 0 contains switch numbers 0 to 7. Wing position 1 contains switch numbers 8 to 15. Wing position 2 contains switch numbers 16 to 23. Wing position 3 contains switch numbers 24 to 31. The switch is numbered using the position of the OPP card (starting at 0), then a '-', and finally the switch number on the card.

Enter them as a combination of board-switch, like `0-12`.

```
switches:
  some_switch:
    number: 0-15
```

The above example configures a switch input as the first OPP card, and the second wing board, last input. On the microprocessor card, the input is marked as 1.7 (wing port 1, position 7).

Switch inputs for solenoids follow the same number convention. Since only four inputs are available for each wing card, it uses the first four switch numbers. Solenoid wing 0 uses switch numbers 0 to 3. Solenoid wing 1 uses switch numbers 8 to 11. Solenoid wing 2 uses switch numbers 16 to 19. Solenoid wing 3 uses switch numbers 24 to 27.

Switch inputs for a switch matrix are number slightly differently. To configure an 8x8 switch matrix wing 2 is configured as the matrix input and wing 3 is configured as a matrix output. The OPP hardware strobes the eight outputs while reading from the eight inputs. This allows 64 inputs to be read using only 16 wires. The matrix switch inputs are numbered from 32 to 95. Switches 32 - 39 are column 0, switches 40 - 47 are column 1, switch 48 - 55 are column 2, switches 56 - 63 are column 3, switches 64 - 71 are column 4, switches 72 to 79 are column 5, switches 80 to 87 are column 6, and switches 88 to 95 are column 7.

Fully working Example

Lets bring above informaton together and learn by example. Though the following example is a fully working minimal set for the Cobra controller, it is as well helpful to understand the concpet more if you use a different set of hardware. For this example to work physically, you only need to power up the micro controllers, no need for any other power supply on the Cobra board. You need to connect a switch to the switch inputs. See as well in *CobraPin Pinball Controller powered by OPP* how to connect a switch. In this example I am using `0-0-16` if you use a different switch input, then you need to change the config file. This `config.yaml` is the only configuration file you need in your project. The config file is fully valid for the Cobra board connected to a Linux PC running mpf. If you have a Cobra board but run Windows or macOS you have to change the `ports`. If you run a completely different hardware you have to adapt the `hardware` section.

```
#config_version=5

  hardware:
    platform: opp
    driverboards: gen2

  opp:
    ports: /dev/ttyACM0, /dev/ttyACM1 # change this if you are not using Linux

  switches:
    my_test_switch:
      debug: true
      number: 0-0-16 # change this if you have connected the switch to a different input
      tags: switch_tag1, switch_tag2
      events_when_activated: active_event1, active_event2
      events_when_deactivated: inactive_event1
```

The important part for this example is to understand the events which are being posted. First of all please obey that we have set `debug:true`, this is necessary to see the events in the mpf monitor. Events are only visible in the mpf monitor when they are either consumed or if the `debug:true` flag is set. Since we don't consume the event in our example we need to set debug to true. Before you start this mpf project with `mpf both` please start `mpf monitor` and activate the window in the monitor to view the events. Now you can press and release the switch and monitor the events being posted. When pressing the switch you should be able to see the following events:

- `my_test_switch_active` based on the switch <switch_name>_active
- `sw_switch_tag1` based on the tags sw_<tag_name>
- `sw_switch_tag1_active` based on the tags sw_<tag_name>_active
- Same as the last two, just for the second tag `switch_tag2`
- `active_event1` based on the configuration `events_when_activated`
- `active_event2` based on the configuration `events_when_activated`

Once you release the switch again some events are being fired:

- `my_test_switch_inactive` based on the switch <switch_name>_active
- `sw_switch_tag1_inactive` based on the tags sw_<tag_name>_active
- `sw_switch_tag2_inactive` based on the tags sw_<tag_name>_active

- `inactive_event1` based on the configuration `events_when_activated`

Please obey the difference in activating and releasing a switch in terms of what events are being fired. When activating a switch the event `sw_<tag_name>` is being fired, there is no corresponding event when a switch goes inactive. See as well the *Events* reference.

What if it did not work?

Have a look at our *OPP troubleshooting guide*.

OPP coils / drivers

Related Config File Sections
coils:

There are a few things to know about controlling drivers and coils with OPP hardware.

> **Warning:** Please ensure that you have established *common ground between logic and coil power* before turning on high voltage on your coils (especially on homebrew machines). Ignoring this might lock on your coils, overheat them, burn down your house or kill you. We are serious, floating grounds are dangerous. If you are not an electrical engineer read the *guide about voltages and power*.
>
> In a nutshell: You need to connect your logic ground (5V/12V) and your high voltage ground (48V or 80V). A *power entry or power filter board* is a convenient solution to solve this (and more) issues.
>
> Always turn all PSUs off when connecting power or you might fry all boards at once. This is generally a good idea but even more important when connecting more than one power supply to a board.
>
> IF YOU DID NOT UNDERSTAND WHAT THIS WARNING MEANS STOP NOW AND TRY TO UNDERSTAND IT. OTHERWISE YOUR HARDWARE WILL LIKELY BURST INTO FLAMES AND YOU NEED TO WAIT A FEW DAYS FOR A REPLACEMENT OR EVEN WORSE IT MIGHT KILL YOU. IGNORING THIS IS THE MOST COMMON CAUSE FOR BROKEN DRIVER BOARDS.

Number

OPP coils are numbered sequentially depending on which wing board is the coil output. Wing position 0 contains coil numbers 0 to 3. Wing position 1 contains coil numbers 4 to 7. Wing position 2 contains coil numbers 8 to 11. Wing position 3 contains coil numbers 12 to 15. The coil is numbered using the position of the OPP card (starting at 0), then a '-', and finally the coil number on the card.

```
coils:
  some_coil:
    number: 0-12
```

The above example configures a coil output as the first OPP card, and the third wing board, first output. On the microprocessor card, the output is marked as 3.4 (wing port 3, position 4).

Pulse time

The OPP hardware also has the ability to specify the "pulse time". Pulse time is the coil's initial kick time. For example, consider the following configuration:

```
coils:
  some_coil:
    number: 0-12
    default_pulse_ms: 30
```

When MPF sends this coil a pulse command, the coil will be fired for 30ms.

Hold Power

If you want to hold a driver on at less than full power, MPF does this by using *default_hold_power* parameter which works for all platforms. It can range from 0.0 to 1.0 and defines the time share the coil is on (0%-100%).

The period is fixed at 16ms for OPP. To set the hold power to 25%, set default_hold_power to .25 and OPP will use 4ms/16ms = 25%.

```
coils:
  some_coil:
    number: 0-3
    default_pulse_ms: 32
    default_hold_power: 0.5
```

This will configure OPP card 0, solenoid wing 0, last solenoid to have an initial pulse of 32 ms, and then be held on at 50% power.

Pulse Power

Note: This feature is only available on OPP firmware version 2.3.0.5 and above.

If you want to pulse a driver on at less than full power, MPF does this by using *default_pulse_power* parameter. This can be used along with default_pulse_ms to fully tune the response of a coil. This can be especially useful when controlling lower voltage coils from a 48V source.

It can range from 0.0 to 1.0 and defines the time share the coil is on (0%-100%). Any value under 0.03125 will be forced to output 3.125%.

```
coils:
  some_coil:
    number: 0-3
    default_pulse_ms: 32
    default_pulse_power: 0.8125
    default_hold_power: 0.125
```

This will configure OPP card 0, solenoid wing 0, last solenoid to have an initial pulse of 32 ms at 81.25% power, and then be held on at 12.5% power.

Recycle Factor

OPP allows you to fine tune the *recycle time of your coils*. If you add `recycle: True` to your coil you can set `recycle_factor` in the `platform_settings` secton of your coil to set the recycle time. The time will be `default_pulse_ms * recycle_factor`. For instance, if you set a pulse time of `10ms` and a recycle_factor of two the coil will cool down for at least `20ms`. This is an example:

```
coils:
  some_coil:
    number: 0-3
    default_pulse_ms: 10
    default_recycle: true
    platform_settings:
      recycle_factor: 2
```

What if it did not work?

Have a look at our *OPP troubleshooting guide*.

Related How To guides

- *Coil Resistance and Hardware Details*
- *Wiring Dual Wound Coils*
- *Dual-Wound versus Single-Wound coils*
- *Adjust coil hold power*
- *Adjust coil strength (pulse times)*
- *Recycle / "Cool Down" Time*
- *Details About Flippers*
- *How to configure single-wound flippers*
- *How to configure dual-wound flippers*
- *Flipper end-of-stroke (EOS) switches*

OPP Lights

Related Config File Sections
lights:

If you're using an OPP incandescent wing card, the lights are numbered the same as the input switches. OPP bulbs are numbered sequentially depending on which wing board controls the output. Wing position 0 contains bulbs 0 to 7. Wing position 1 contains bulbs 8 to 15. Wing position 2 contains bulbs 16 to 23. Wing position 3 contains bulbs 24 to 31. The bulb is numbered using the position of the OPP card (starting at 0), then a '-', and finally the bulb number on the card.

```
lights:
  some_light:
    number: 1-16
    subtype: matrix
```

The above example configures a bulb on the second OPP card, and the third wing board, first bulb On the microprocessor card, the input is marked as 2.0 (wing port 2, position 0).

What if it did not work?

Have a look at our *OPP troubleshooting guide*.

OPP LEDs

Related Config File Sections
lights:

OPP hardware can directly drive LED strips. This features is currently being developed. Documentation will be added as the feature becomes more mature.

LEDs work similar to matrix lights (chain 0, board 1, LED 1):

```
lights:
  some_led:
    number: 0-1-1
    subtype: led
    type: rgb
```

Note, counting starts always with 0, so LED 1 in aboves example is the 2nd LED of the strip.

Overview video about serial LEDs:

https://youtu.be/Q9BG9T7Kj4A

Channel and Number Syntax

In MPF *lights* abstract a light source which emits arbitrary colors. However, this is not true for all real lights. Some support only white (GIs), others only a single-color (i.e. red inserts) and others support full RGB. For that reason MPF knows *light numbers and channel numbers*. Internally, a light consists of one or multiple channels. For instance, a single-color GI will contain a single white channel. While a RGB light will control a red, green and a blue channel. A white light behind a red insert should be a single red channel (because it cannot emit other colors through the red insert). You can configure those channels using the `channels` setting or use `start_channel` and `type` to define the channels. See *Lights* for details.

However, in most cases a platform supports one type of lights (per `subtype`) this would be overly verbose and we added the `number` setting for configuring lights in the common platform way. For instance a platform for GIs will configure single channel white lights or a serial LED controller will configure RGB lights with three channels.

OPP assumes RGB lights by default. For everything else (i.e. RGBW) you have to use channels.

Light Numbers

OPP numbers use the format: `serial_chain-card_num-index`

`chain_serial` is only relevant if you got multiple chains connected via USB. See *Connecting OPP to your computer* for details about chains. If you only got one chain you can omit this part and your format becomes `card_num-index`.

`card_num`` is the index of the board on the chain. As the first board is always at addr `0x20` you can calculate the addr using `0x20 + card_num`. If you only got one board you can omit the board and your format becomes just `index`.

For instance, `0-0-0` for the first RGB LED on chain `0` on card `0x20`. In this case you can also use `0-0` or `0` (channel `0-2`). `0-0-1` or `0-1` or `1` is the second LED on the chain (channels `3-5`).

`3-2-6` is the 6th LED on board 2 (addr `0x22`) of chain `3` (channels `18-20`).

Channels

OPP channels use the format: `serial_chain-card_num-internal_index`

This is mostly the same as numbers above except that `internal_index = 3 * index`. This is because serial LEDs are traditionally RGB (or GRB) LEDs with exactly three channels. However, this is not true for RGBW or similar LEDs which do not work with this style of numbering. Luckily, you can chain them instead and have MPF calculate the internal channels for you:

```
lights:
  led_0:
    start_channel: 0-0-0
    subtype: led
    type: rgb    # will use red: 0-0-0, green: 0-0-1, blue: 0-0-2
  led_1:
    previous: led_0
    subtype: led
    type: rgbw   # will use red: 0-0-3, green: 0-0-4, blue: 0-0-5, white: 0-0-6
  led_2:
    previous: led_1
    subtype: led
    type: rgbw   # will use red: 0-0-7, green: 0-0-8, blue: 0-0-9, white: 0-0-10
```

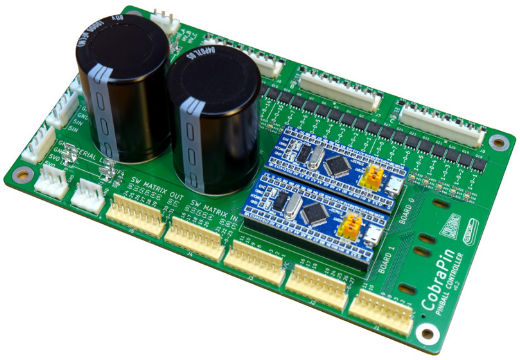
See *WS2811 and WS2812 LEDs in Pinball* for details.

What if it did not work?

Have a look at our *OPP troubleshooting guide*.

CobraPin Pinball Controller powered by OPP

This page is under development. . . don’t believe a word you read.



Features:

- 24 solenoid outputs broken into 3 banks
- 38 direct inputs <OR> 22 direct inputs + 8x8 switch matrix
- Neopixel support for 512 RGB pixels (RGBW also possible but may be limited to ~460 pixels)
- 24-50V power filter. Board also provides the common ground for the supplies.
- Fuses for solenoid banks and Neopixels

The size of the board is about 197 x 115 mm. Mounting holes are available in the corners of the board, spaced 184 mm and 103 mm respectively. Mounting holes are good for M4 screws.

Overview video about Cobrapin:

https://youtu.be/UQNMaLv3woo

Overview video about OPP:

https://youtu.be/WU98MRDeYeQ

Video about cobrapin extension board:

https://youtu.be/lfxKcaiZyMs

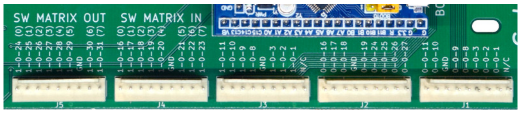
Power Input and Filter

J9: Solenoid power input (24-50V).

J10: Neopixel 5V input.

The filter provides consistent power to solenoids while also protecting the power supply from sudden current surges that may otherwise cause a fault. The connectors for the power supply on the board are JST VH style connectors.

Switch Inputs



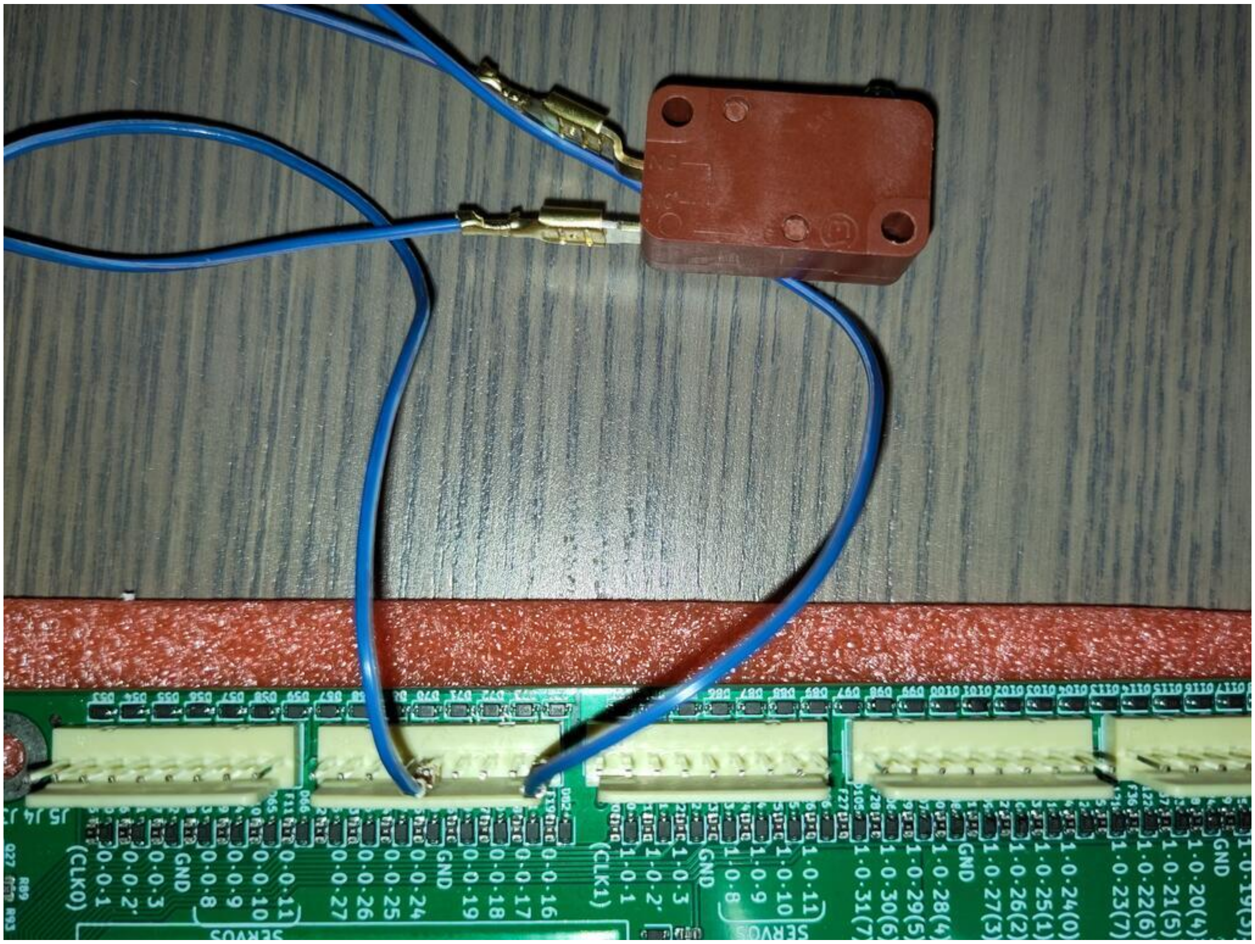
J1, J2, J3: Direct input switches.

J4, J5: Remaining direct input switches or switch matrix input/output.

If you do have either 38 direct inputs or 22 direct inputs + a 8x8 switch matrix depends on your Cobra board. You specify this as an option when you order your board. The switch inputs are labeled in silkscreen with the MPF compatible numbers. The two pins labeled "N/C" are not connected to anything.

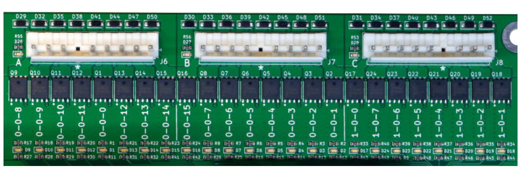
The connectors for the switches on the board are KF2510 style connectors. Each connector also includes a logic ground pin. Use this for the direct input return. If you measure the voltage between GND and a switch (in below picture 0-0-16) you should measure 3.3V.

For that to measure only the micro controllers need to be powered up, no need to apply any other voltage on the Cobra board. To perform a simple test connect any kind of switch to one of the inputs and setup a little mpf test configuration.



Do not apply any voltage to the switches, most likely that will destroy your CPU. For further details and fully working Cobra board configuration example please check *OPP Switches*. For autofire devices please see for additional remarks in the solenoid section below.

Solenoid Outputs

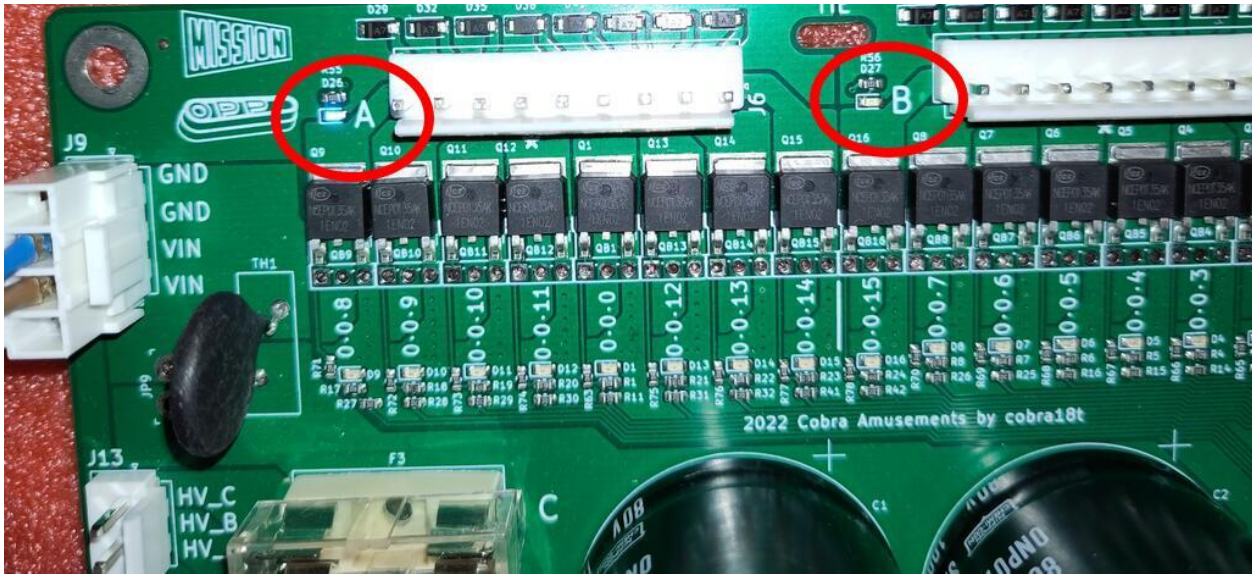


J6, J7, J8: Solenoid outputs.

The 24 solenoids are broken up into 3 banks of 8 outputs. The connectors for the solenoids on the board are JST VH style connectors. There is a ninth pin on the connector that can be used as a key. Each solenoid has a diode to help protect the transistor. You may still use coils with axial diodes installed, but you MUST ensure that you connect them with the correct polarity.

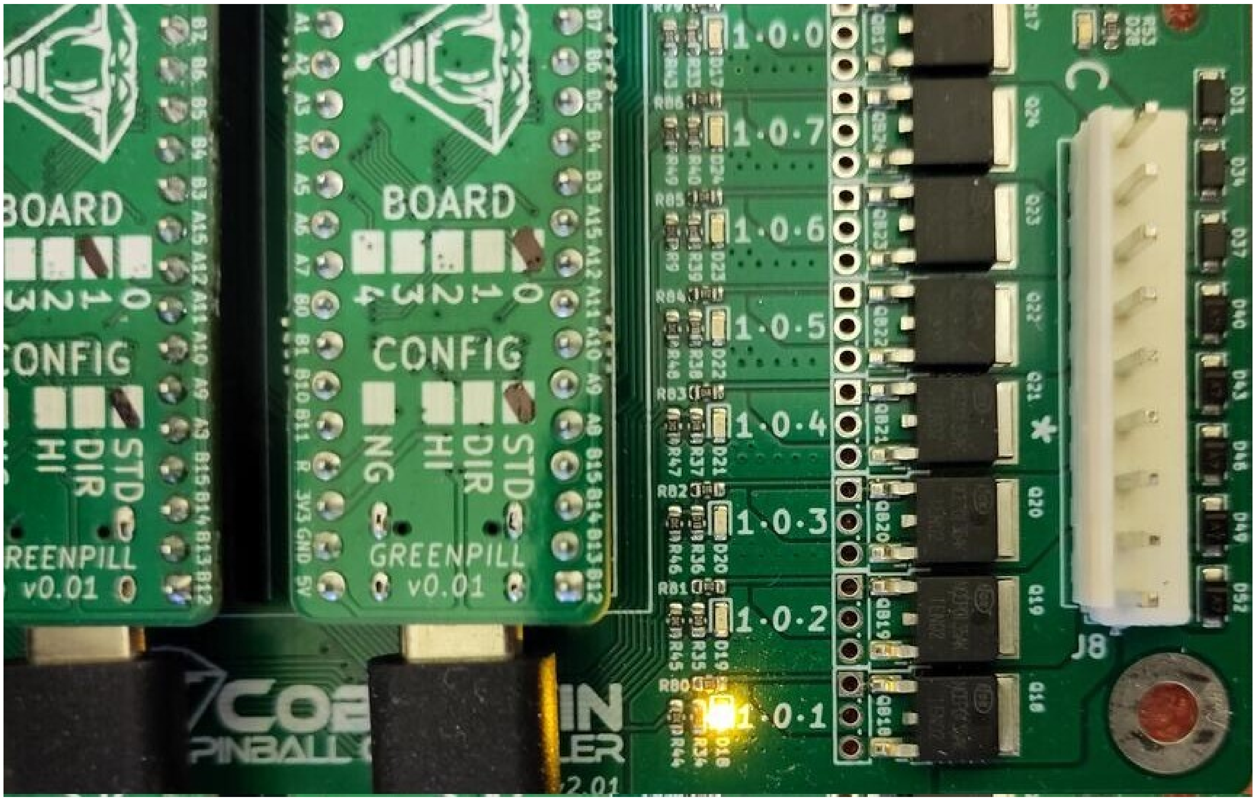
The solenoid outputs are labeled in silkscreen with the MPF compatible numbers. *OPP coils / drivers*. You need to obey that some of these outputs are controlled by the first micro controller and some by the second. The first digit of the solenoid number shows by which micro controller it is addressed, e.g. `1-0-7` is controlled by micro controller 1. This is important later for the autofire devices (flippers, slingshots, bumpers), because they are hardware controlled and the switch and the coil must use the same controller. In other words a coil on `0-x-y` of an autofire device must be controlled by switch with number `0-a-b`.

Each bank has an LED next to it to indicate if that bank has power. Check these if you are concerned you have blown a fuse.



In above picture you see that the LED for bank A is alight but not for bank B. In order to have the LED alight you only need to have connected your high power supply, no need for the 5V power supply or to have micro controllers booted up. Please be aware that once you remove the power supply the LED will still glow for a while until the the capacitors have discharged.

Each solenoid has an associated LED to indicate it is being driven by the processor. It is highly recommended to test a new setup without high voltage power or without the coils plugged in. Using these LEDs, you can verify that each output is being driven correctly, in the picture below coil 1-0-1 is being driven at this very moment.



To run the above test, there is no need for a high voltage power supply neither for any coil. Only the mirco controllers need to be powered up. The `config.yaml` below is the only configuration file you need in your project. The config file is fully valid for the Cobra board connected to a Linux PC running MPF. If you have a Cobra board but run Windows or macOS you have to change the `ports`.

```
#config_version=5

hardware:
   platform: opp
   driverboards: gen2

opp:
   ports: /dev/ttyACM0, /dev/ttyACM1 # change if your Cobra board uses different ports

coils:
    c_my_coil:
       number: 1-0-1
       pulse_events: s_my_switch_active

switches:
   s_my_switch:
      number: 0-0-16
```

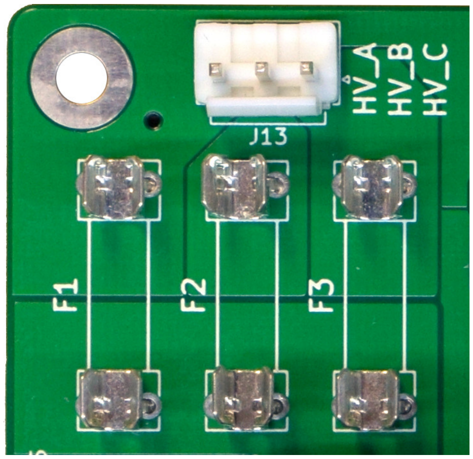
Some remarks on above `config.yaml`

- Obey that we don't have an autofire device in this example, and thus the coil and the switch can be connected to the different micro controllers.
- In the coil section `pulse_events` is being used, don't mix it up with `enable_events` which would not only pulse the coil but have it on permanently.

- When a switch is being activated automatically an event *(switch_name)_active* is being fired. The above example makes use of this fact.

To have a fully working example for setting up autofire coils see the *Autofire Coils* section of the documentation.

Solenoid Power Output and Fuses



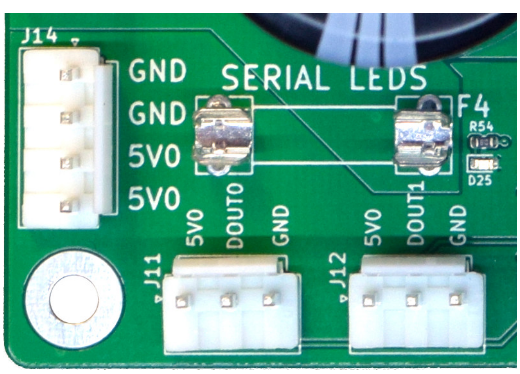
J13: Solenoid power outputs.

F1, F2, F3: Solenoid power bank fuses.

The fuses are 5x20mm. Each fuse provides power to a bank of 8 solenoids.

Note: Solenoids in bank A should only be powered by the HV_A pin, bank B should only be powered by HV_B, bank C should only be powered by HV_C. Failure to do so may confuse future troubleshooting and could eventually blow out a transistor.

Neopixel Support



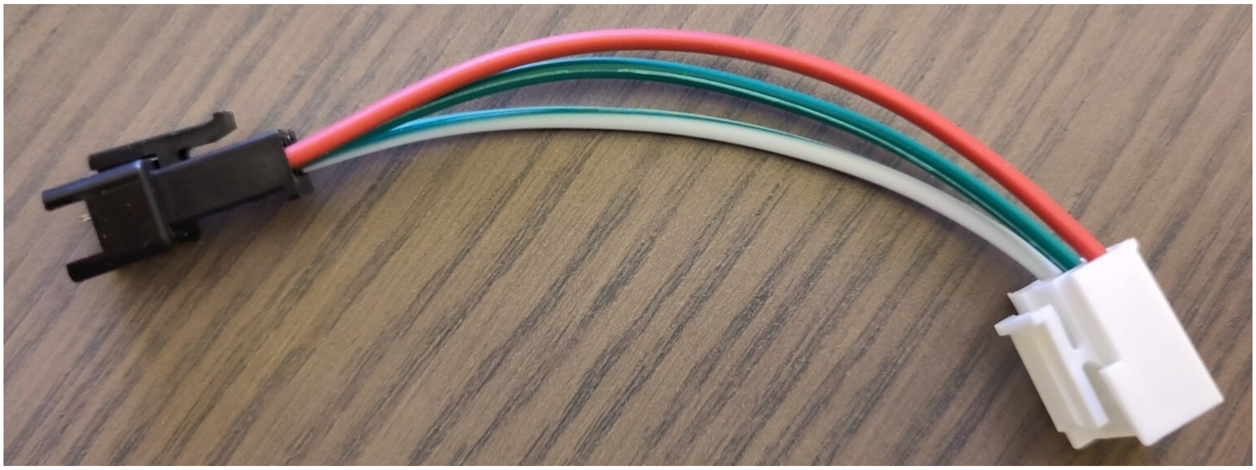
J10: Power input for Neopixels, most likely 5V, but if you use 12V Neopixels you need to provide 12V power here. Power input is used for both Neopixel chains.

J11, J12: Neopixel outputs

F4: 5V fuse for neopixels

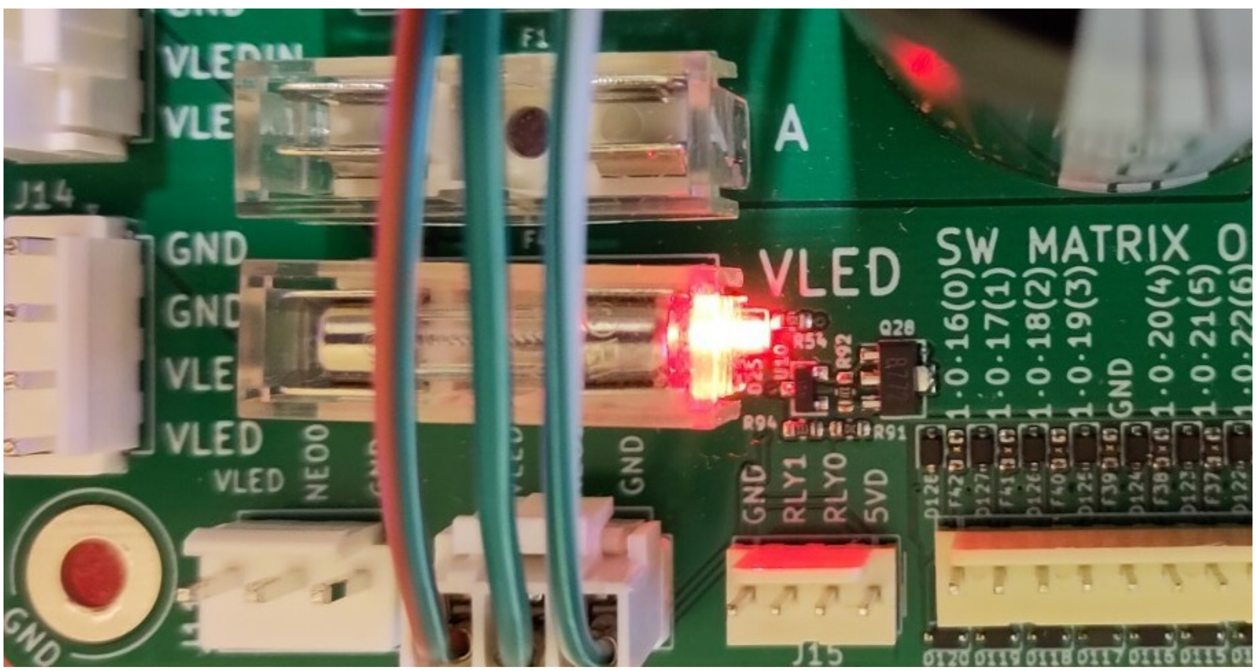
J14: Fused 5V output

The connectors J10, J11, J12 and J14 are JST connectors VH style. There are lots of Neopixels which come with a JST connector SM style. You might want to craft a little converter cable in such a case.

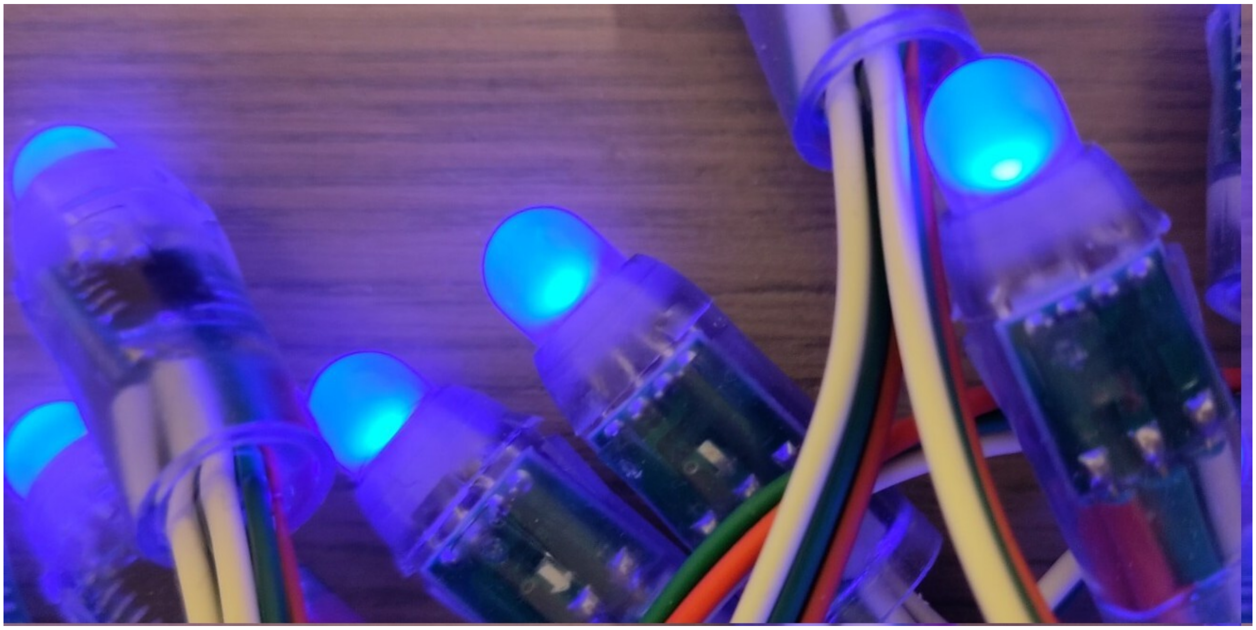


There are two neopixel chains that support 256 RGB pixels each for a total of 512. RGBW pixels are also possible, but the number may be limited to 230 pixels per chain for a total of 460.

The J14 fused output can be used to provide additional power taps in a neopixel chain. Each pin is rated for 7A continuous. The fuse holder is rated for 10A. The red D25 LED can be used to confirm you have a good fuse and are providing power for neopixels. For the LED to light up there is no need to run any MPF configuration, you don't even have to power up the micro controllers.



When you order the micro controllers you have various options, one option to choose from is Regular vs NoGlow. If you order the Regular version then after power is provided for the Neopixel and the micro controllers are powered up (still no need to run any MPF on them), the LEDs of your strip will glow blue, which is a good first test.



In order to addess the LEDs in MPF you need to know their address

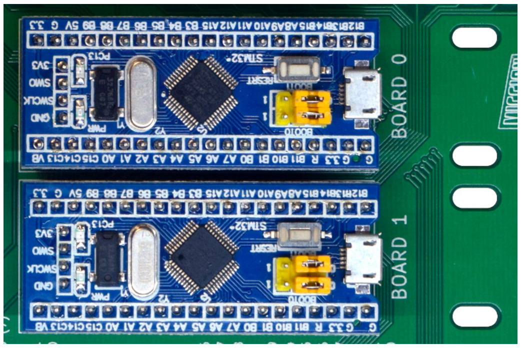
J11: NEO 0 Neopixel output, all these lights have MPF numbers with the format 0-0-##. The first LED in the chain is 0-0-0.

J12: NEO 1 Neopixel output, all these lights have MPF numbers with the format 1-0-##. The first LED in the chain is 1-0-0.

Details on how to configure LEDs in your mpf project can be found here *OPP LEDs*.

Two fully working example for the Cobra board can be found in the generic LED section *LEDs* where as well the more general concept is explained.

Microcontrollers

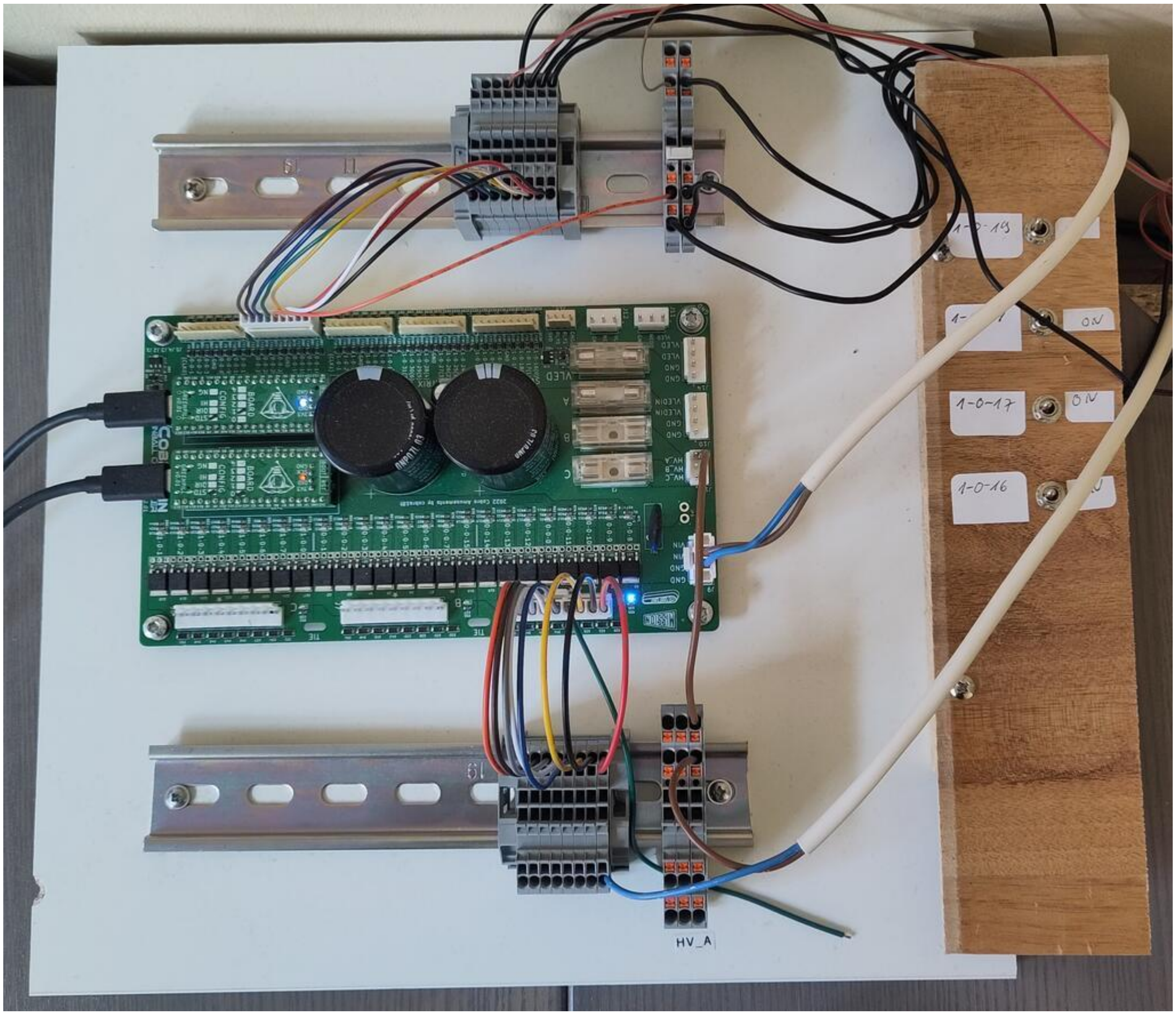


The brains of the CobraPin are two STM32 microcontroller boards programmed with OPP firmware. They are connected to the host computer via micro USB connectors.

Note: It is important to have your config file refer to the silkscreen board numbers (0 and 1) in the correct order, otherwise the labels on the solenoids, switches, etc. will refer to incorrect pin numbers.

The microcontrollers are removable so you can replace them if they fail for whatever reason. They are widely available and often referred to as "STM32 Blue Pill" boards. The right angle header that is normally used as a programming port is replaced with a vertical header so that those pins can be used on the CobraPin board.

Test Rig



For an easy start you might want to setup a test rig similar to the one shown above. The advantage is that you can use push in clamps to change connected hardware easily without the need to crimp lots of cables. For the connection from the board to the push in clamps you can use pre-fabricated headers with wires, then there is no need to crimp anything.

BOM (Bill of material)

Item	Amount	Description
board	1	A wooden board about 30x30cm
Cobra board	1	
DIN rail	2	each rail about 20cm or longer
Spacer	8	Spacer (plastic) to mount the board, one spacer above and one below the board. Diameter M4.
Screws	4	About 4,0x20mm (depending on the length of your spacer)
Screws	4	About 3,0x15mm to mount the rails, length can vary depending on board thickness
KF2510 wires, 9 pins	5	KF2510 plugs with wire, 9 pins
KF2510 wires, 4 pins	1	KF2510 plugs with wire, 4 pins
JST VH wires, 9 pins	4	JST VH plugs with wires, 9 pins
JST VH wires, 4 pins	3	JST VH plugs with wires, 4 pins
JST VH wires, 3 pins	3	JST VH plugs with wires, 3 pins
push in connector 1:1	>10	1:1 wire connection, for each switch and each coil you need one, buy plenty.
push in connector 1:n	5	1:n wire connection, for ground of the switches and power for the soils, amount varies.

You can use as well bridges to connect multiple cage clamps together, that might be handy for ground connection. See the two cage clamps at the top right, they have a little bridge (this light grey/white box) to have all inputs internally connected.

Example Config

```
#config_version=5

#CobraPin Example Config

hardware:
  platform: opp
  driverboards: gen2



opp:
  #Use the USB ports defined by your OS for the two STM32 boards
  ports: /dev/ttyACM0, /dev/ttyACM1
  #USING SERIAL NUMBERS INSTEAD OF CHAINS
  #  Board 0 has serial number 0, Board 1 has serial number 1.
  #  This is convenient if your OS tends to reassign the serial port.
  #  MPF will automatically address the correct board even if the ports
  #     are swapped.

  #For multiple CobraPin boards in a game, you will either have to give
  #  the STM32 boards on the second CobraPin board new serial numbers
```

(continues on next page)

(continued from previous page)

```
  #  (10 and 11 are suggested for the 2nd board since 2 is used by the
  #  CobraPin Xpansion Board)
  #  <OR> Use the chains section to assign a port to a board number.
  #  Mixing these up could cause blown FETs, coils, and fuses. Proceed
  #  with caution. Test without coil power and use the yellow coil LEDs
  #  for feedback.
  #chains:
     #0: /dev/ttyACM0
     #1: /dev/ttyACM1


psus:
  default:
    #Gives the capacitors extra time to recharge after firing a coil
    #  and eases the load on the power supply. Doesn't affect autofire
    #  devices like flippers, pops, slings.
    release_wait_ms: 50


#One giant config file can get difficult to manage. You can put any of
#  these config sections in its own yaml file and link to it with the
#  config section here
config:
  #- switches_config.yaml
  #- lights_config.yaml
  #- coils_config.yaml
  # ...


switches:

  #DIRECT SWITCHES
  #switch numbers are labelled in silkscreen on the board
  s_left_flipper:
    number: 0-0-27
    tags: left_flipper
  s_right_flipper:
    number: 0-0-26
    tags: right_flipper
  s_startButton:
    number: 0-0-25
    tags: start


  #MATRIX SWITCHES
  #valid numbers are 1-0-32 through 1-0-95
  s_lowerDrop1:
    number: 1-0-32

  # ...

  s_topRollunder:
    number: 1-0-95
    ignore_window_ms: 250ms    #tune to assist in debouncing
```

(continues on next page)

(continued from previous page)

```
lights:

  #SERIAL LEDS (neopixels)
  #NEO0 output supports 256 LEDs numbered 0-0-0 to 0-0-255
  l_shootAgain:
    number: 0-0-0
    subtype: led
    type: grb   #Most WS2812-based LEDs are grb color order.
                #This line not required for rgb ordered LEDs like the
                #  WS2811 LEDs shown below

  # ...

  #NEO1 output supports 256 LEDs numbered 1-0-0 to 1-0-255
  l_gi_1:
    number: 1-0-0
    subtype: led
    tags: gi    #you can group similar LEDs with user defined tags
  l_gi_2:
    number: 1-0-255
    subtype: led
    tags: gi


coils:
  #coil numbers are labelled in silkscreen on the board

  #There are multiple ways to configure flippers, use the one that
  #  matches your hardware
  c_flipper_left:
    number: 0-0-8
    allow_enable: true
    default_hold_power: 1.0
    default_pulse_ms: 50
  c_flipper_right:
    number: 0-0-4
    allow_enable: true
    default_hold_power: 1.0
    default_pulse_ms: 50
  c_ballRelease:
    number: 1-0-1
    default_hold_power: 0.15
    default_pulse_ms: 30


flippers:
  #Add your flipper config


autofire_coils:
  #Add your autofire cofigs for pops, slings, etc.
```

(continues on next page)

(continued from previous page)

```
ball_devices:
  #Add your ball devices


playfields:
  #Define your playfields


machine:
  balls_installed: 3 #How many balls are physically in your game
  min_balls: 3 #How few balls can be accounted for before you can start a game


game:
  balls_per_game: 3
  max_players: 4


modes:
  #Add all your mode names here
  #- attract
  #- base
  #- etc


keyboard:   #use to drive your game from the computer for testing
  z:
    switch: s_left_flipper
  "/":
    switch: s_right_flipper
```

NeoSeg (CobraPin) Serial Segment Displays

Video about CobraPin serial segment displays (This was a prototype 7-digit 16-segment version versus the production 8-digit 14-segment version):

https://youtu.be/iMeX1qC4EA0

Add Segment Display Platform

NeoSeg displays consist of a string of serial LED controllers connected to segment display LEDs. Each LED channel is connected to a segment. You must add light_segment_displays as the segment_displays platform so MPF knows to send segment display commands to the light controller:

```
hardware:
  segment_displays: light_segment_displays
```

Create the NeoSeg Light Groups

The creation of the mpf lights for a NeoSeg display is handled by creating a light_group using "neoseg_displays." This is much easier than defining each light for each of the 120 segments in an 8-digit display.

```
neoseg_displays:
  neoSeg_0:
    start_channel: 0-0-0
    size: 8digit
    light_template:
      type: w
      subtype: led
      color_correction_profile: NeoSeg_orange
```

Here we have created a neoseg_display light_group named "neoSeg_0." The group starts at light channel 0-0-0, which is the first light channel on Board 0 on CobraPin, for example. It is an 8-digit display.

Note that we are dealing with light *channel* numbers, not light numbers. If you have all RGB LEDs, the channel number is 3 times your light number. An 8-digit NeoSeg display occupies 120 channels while the 2-digit display occupies 30 channels. So if we added another NeoSeg display after neoSeg_0, its start_channel would be 0-0-0 plus 120. . . so 0-0-120.

We had to add a light template so that we could identify it as a single channel LED.

Also, the use of a color_correction_profile enables you to change the brightness of a NeoSeg display. This is especially handy when using NeoSeg displays of different colors since each color has a different default brightness. Use the whitepoint setting to vary the brightness:

```
light_settings:
  color_correction_profiles:
    NeoSeg_orange:
      whitepoint: [.9, .9, .9]
```

Create Segment Displays

Once you have the light groups defined, you can arrange them into displays. These are the displays that can be targeted by a segment_display_player.

```
segment_displays:
  neoSegTop:
    number: 1
    size: 16
    integrated_dots: true
    use_dots_for_commas: true
    default_transition_update_hz: 30
    platform_settings:
      light_groups:
        - neoSeg_0
        - neoSeg_1
      type: 14segment
```

Here we create a 16-digit display called "neoSegTop" built from 2 8-digit displays – "neoSeg_0" and "neoSeg_1". The "integrated_dots" and "use_dots_for_commas" settings are required to use the comma segments built into NeoSeg displays. NeoSeg displays have a type of "14segment."

Be sure to change "size" to the total number of digits that you intend to use in the display. In some cases, the opening in your backglass may only be wide enough for 7 digits for example. In that case, change the size to 7 and the 8th digit will remain unused.

Complete Example Config

```
#config_version=5

hardware:
  platform: opp
  driverboards: gen2
  segment_displays: light_segment_displays

#create light group for each NeoSeg display
neoseg_displays:
  neoSeg_0:
    start_channel: 0-0-0
    size: 8digit
    light_template:
      type: w
      subtype: led
      color_correction_profile: NeoSeg_orange
  neoSeg_1:
    start_channel: 0-0-120
    size: 8digit
    light_template:
      type: w
      subtype: led
      color_correction_profile: NeoSeg_white

  neoSeg_7:
    start_channel: 0-0-660
    size: 2digit
    light_template:
      type: w
      subtype: led
      color_correction_profile: NeoSeg_blue
  neoSeg_8:
    start_channel: 0-0-690
    size: 2digit
    light_template:
      type: w
      subtype: led
      color_correction_profile: NeoSeg_red

#use light groups to arrange into a segment display
segment_displays:
  neoSegTop:
    number: 1
    size: 16
    integrated_dots: true
    use_dots_for_commas: true
```

(continues on next page)

(continued from previous page)

```
    default_transition_update_hz: 30
    platform_settings:
      light_groups:
        - neoSeg_0
        - neoSeg_1
      type: 14segment

  neoSegBot:
    number: 1
    size: 4
    integrated_dots: true
    use_dots_for_commas: true
    default_transition_update_hz: 30
    platform_settings:
      light_groups:
        - neoSeg_8
        - neoSeg_7
      type: 14segment

#use color_correction_profile whitepoint to adjust the brightness of each
#NeoSeg display
light_settings:
  color_correction_profiles:
    NeoSeg_red:
      whitepoint: [.8, .8, .8]
    NeoSeg_white:
      whitepoint: [.55, .55, .55]
    NeoSeg_blue:
      whitepoint: [.5, .5, .5]
    NeoSeg_orange:
      whitepoint: [.9, .9, .9]
    NeoSeg_yellow:
      whitepoint: [1, 1, 1]
    NeoSeg_green:
      whitepoint: [.5, .5, .5]
```

Troubleshooting OPP

If you got problems with your hardware platform we first recommend to read our *troubleshooting guide*. Here are some hardware platform specific steps:

Run Hardware Scan

Using `mpf hardware scan` you can find out if your OPP boards are talking properly to MPF using USB:

```
$ mpf hardware scan

Connected CPUs:
 - Port: com1 at 115200 baud
 -> Board: 0x20 Firmware: 0x10100
 -> Board: 0x21 Firmware: 0x10100
```

(continues on next page)

(continued from previous page)

```
Incand cards:
 - CPU: com1 Board: 0x20 Card: 0 Numbers: [16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
↪31]

Input cards:
 - CPU: com1 Board: 0x20 Card: 0 Numbers: [0, 1, 2, 3, 8, 9, 10, 11, 12, 13, 14, 15]
 - CPU: com1 Board: 0x21 Card: 1 Numbers: [0, 1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
↪18, 19, 20, 21, 22, 23, 24, 25, 26, 27]

Solenoid cards:
 - CPU: com1 Board: 0x20 Card: 0 Numbers: [0, 1, 2, 3]
 - CPU: com1 Board: 0x21 Card: 1 Numbers: [12, 13, 14, 15]

LEDs:
 - CPU: com1 Board: 0x21 Card: 1
```

See *mpf hardware (command-line utility)* for details.

Enable Debugging

If you got problems with your platform try to enable `debug` first. As described in the *general debugging section* of our *troubleshooting guide* this is done by adding `debug: true` to your `opp` config section:

```
opp:
  debug: true
```

This will add a lot more debugging and might slow down MPF a bit. We recommend to disable/remove it after finishing debugging.

Reducing the polling rate

If you encounter issues with the polling rate (in other words: Your OPP processor boards can't answer MPF's polls fast enough) you may want to change it. This can be done by simply adding the `poll_hz:` line to the `opp:` section:

```
opp:
  ports: COM7
  poll_hz: 50    # defaults to 100
```

Note: You only want to do this if you encounter issues. This will increase the time between two switches beeing read. If you set this too log you could miss hits if multiple hits happened between two polls.

Coils Are Not Firing

What to do if your coils are not working?

Check if Your Hardware is Working at all

Sounds stupid but this is a good start: Is the hardware working at all? Do you see switch hits in the logs? If not, check our section `Your hardware is not working at all`.

Check the Watchdog

If switches (or other features of the platform) are working but coils are not we have to dig deeper. Most hardware platforms have some kind of watchdog. Often there is some LED which indicates if the watchdog is received. The MPF log might also contain clues (especially if you have enabled `debug` and run MPF with verbose flags `-v -V`). If the watchdog is not received by your platform it will not enable coils.

In most cases watchdog related problems indicate wiring problems. Check if your boards are properly wired.

Test Your Coil Numbers using MPF Service CLI

Hardware is connected and generally working, watchdog is good but still your coils are not working? Maybe something with the numbering is odd. Lets tests that using the *MPF Service CLI*. Alternatively, you can also use *service mode* if you have already configured it. Both ways work similarly.

To use service cli:

1. Open two consoles
2. Start your game (e.g. using `mpf both`)
3. Start the service cli from within your game folder using `mpf service`.
4. Type `list_coils` and press `ENTER` to see a list of coils.
5. Type `coil_pulse your_coil` and press `ENTER` to pulse it.

Does it work? If not check the log and try verify the coil number. If you do not specify `default_pulse_ms` MPF will use `10ms` which might not be enough for some mechs. Try to increase that gently (maybe `20ms` or `30ms`).

Reducing light update rate

If you got a lot of lights you might run into bus contention issues. You can reduce the light update rate in MPF:

```
mpf:
  default_light_hw_update_hz: 30   # defaults to 50
```

If you set this too low fades will be less smooth but otherwise it should not affect your game.

Your hardware is not working at all

If your hardware is not working at all make sure that you removed the options `-X`, `-x` and `--vpx` from your `mpf both` or `mpf game` command line. Those options will overwrite the settings in your `hardware` section and MPF will not even try to connect to your hardware. If you got config errors we suggest you add `-X` to figure things out without interfacing real hardware all the time. Just keep that option in mind.

Another stupid thing to check: Is your hardware connected to your PC? We know it is stupid but a loose USB connector has happened to most of us.

On Linux you might want to run the command `lsusb` which should show both of your micro controllers connected. You should see two lines similar to

```
Bus 002 Device 014: ID 0483:5740 STMicroelectronics Virtual COM Port
Bus 002 Device 015: ID 0483:5740 STMicroelectronics Virtual COM Port
```

If you are unsure about the output, run the command once with your controllers connected and once without. If there is no difference, then for sure the USB device is not properly connected.

Add debugging to related devices

If you got problems with some switches also add `debug: true` to those as it will give to more insights into the intentions of those devices. Same will work for flippers, coils, lights, servos, steppers and more. See *general debugging section* for details.

Run MPF with verbose flag

See *general debugging section* for details. TLDR: run `mpf both -t -v -V`.

Report Your Issue and Ask For Help

If you cannot find the issue yourself please prepare some information about your issue according to our *troubleshooting guide* and ask in our forum.

Consider Improving the Documentation

Did you solve your issue but found that some relevant information in the documentation is missing or should be linked/located elsewhere? Either tell us in the forum or consider *improving the documentation* yourself to save future users some troubles the same way others saved you some troubles by writing this documentation.

How to use MPF with the LISY platform

Related Config File Sections
hardware:
lisy:
switches:
coils:

MPF can directly control Gottlieb System 1 or System 80 machines via the LISY1 or LISY80 controller boards (with firmware 4.02+). Additionally, LISY35 can control Bally and Stern Games manufactured from 1977 to 1985 with MPU AS-2518-17 or AS-2518-35.

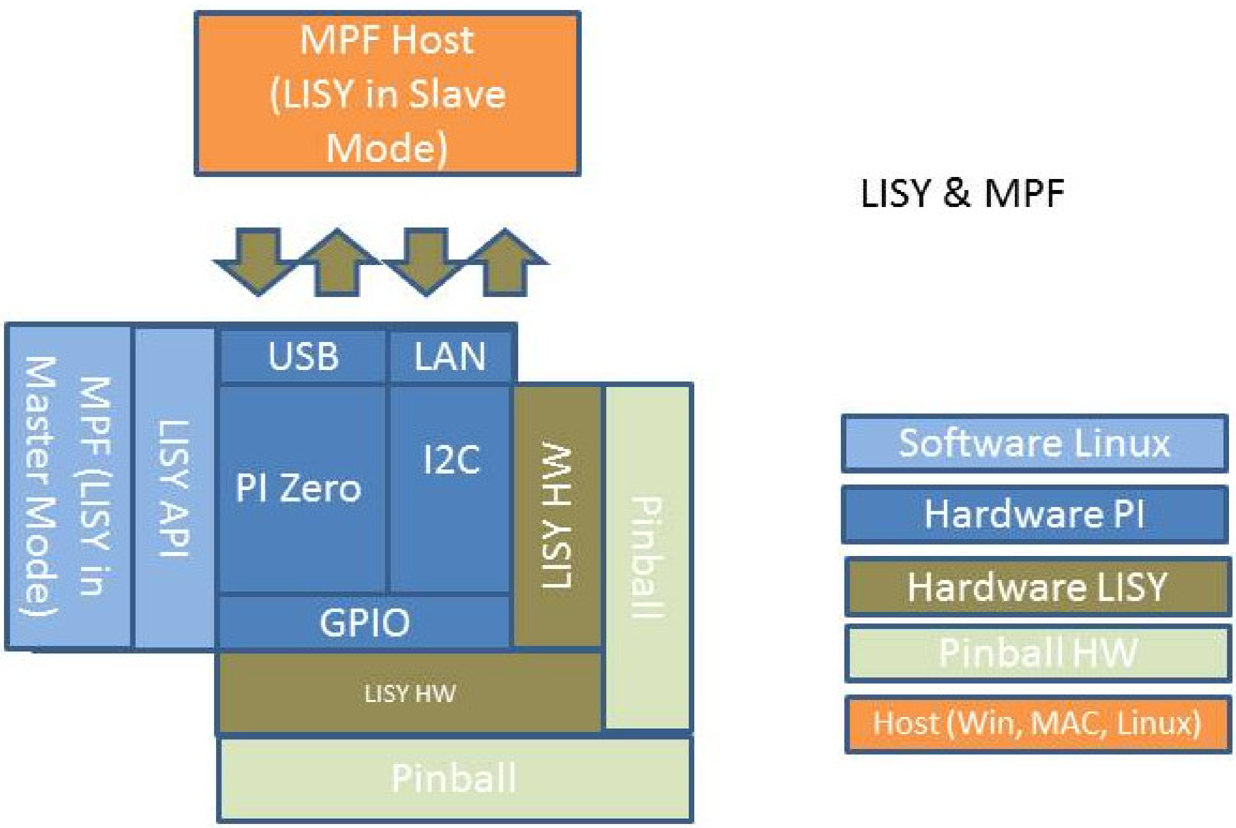
Note: For general installation instruction and some background information on the LISY hardware platform, visit www.lisy80.com.

There are two ways this can be done:

a. Run MPF on a standalone PC which connects to the LISY hardware operating in "slave" mode via Ethernet, WiFi, or serial. This is generally recommended during development since it's easier to work on your MPF config using your own computer. You can also use this configuration if you want to add an LCD or DMD to the older Gottlieb machine.

b. Run MPF on the LISY hardware directly ("master" mode). (Technically MPF is running on the LISY controller's Raspberry Pi Zero.) This option is nice when your game is finished and you no longer want to connect a PC. Note that the Raspberry Pi on the LISY is not powerful enough to run the MPF media controller, so this option is really only valid for simpler, segment display type games. If you want to run a full LCD or DMD, then just run MPF on a separate computer (which can still be small and inside your machine) and connect to the LISY controller via Option (a) above.

See the following image for an architecture overview:



LISY can controll all features of your Gottlieb System1/80 machine. This includes:

- Switches (*LISY1* and *LISY80*)

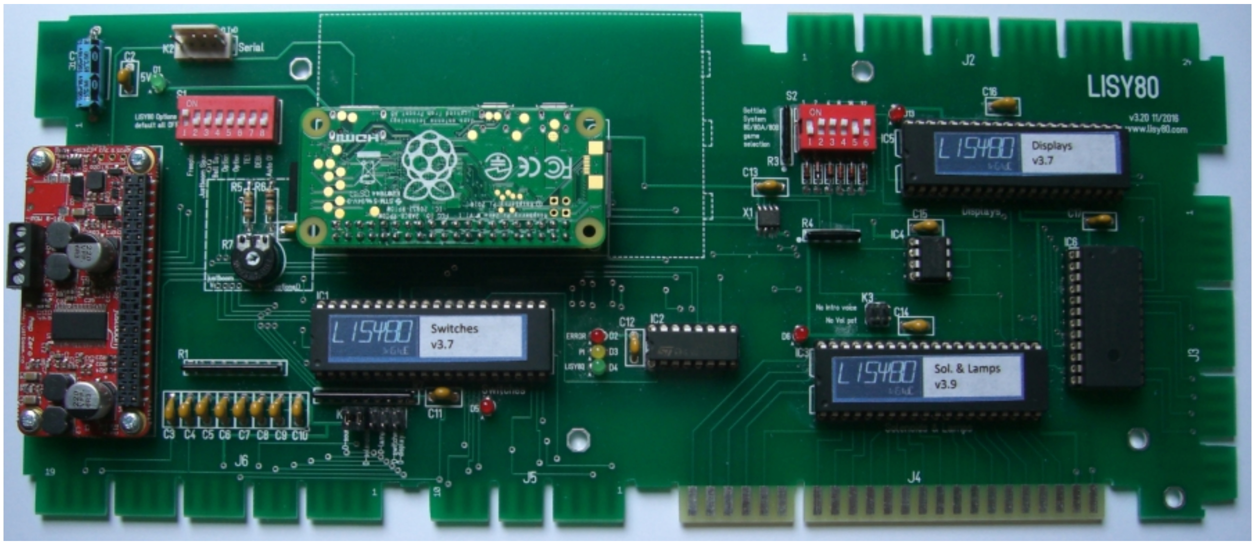
- *Coils*
- *Lights*
- *Enabling/disabling flipper, slings and popbumpers*
- *Segment displays*
- *Original sounds of your game*
- *Text to speech and additional sounds*

Connecting a System1/80 Machine to LISY1/80

1. Replace your original MPU with LISY1/80.

For Gottlieb System80/80A/80B games, replace the existing Gottlieb CPU with the "LISY80" board. For System 1 machines, replace the existing Gottlieb CPU board with the "LISY1" board.

Note: See documentation at www.lisy80.com for details. Basically you replace the MPU with the LISY board. You can still play the original ROM using PinMAME on LISY.



More details can be found in the LISY user manual.

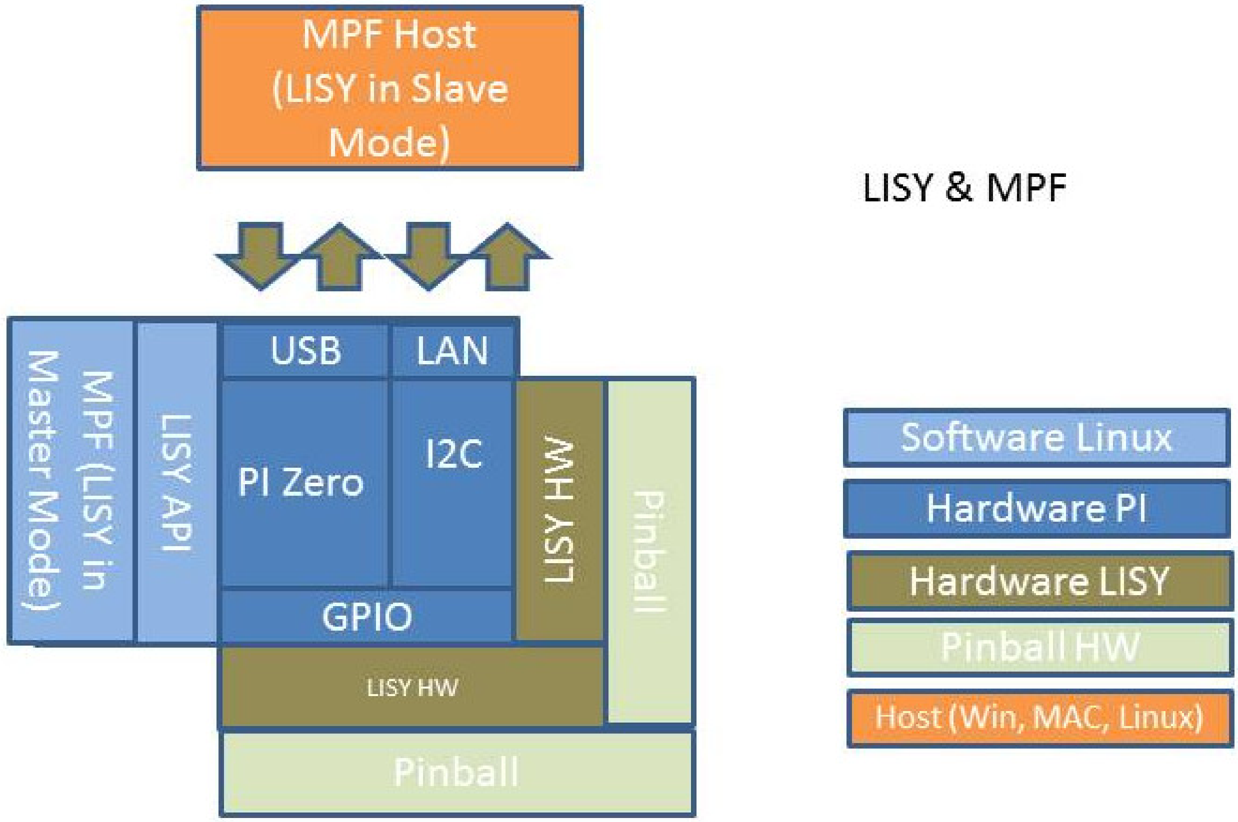
2. Configure LISY to Run MPF

There are two ways this can be done:

a. Run MPF on a standalone PC which connects to the LISY hardware operating in "slave" mode via Ethernet, WiFi, or serial. This is generally recommended during development since it's easier to work on your MPF config using your own computer. You can also use this configuration if you want to add an LCD or DMD to the older Gottlieb machine.

b. Run MPF on the LISY hardware directly ("master" mode). (Technically MPF is running on the LISY controller's Raspberry Pi Zero.) This option is nice when your game is finished and you no longer want to connect a PC. Note that the Raspberry Pi on the LISY is not powerful enough to run the MPF media controller, so this option is really only valid for simpler, segment display type games. If you want to run a full LCD or DMD, then just run MPF on a separate computer (which can still be small and inside your machine) and connect to the LISY controller via Option (a) above.

See the following image for an architecture overview:



If you want to run MPF on the LISY controller itself, set DIP 4 (option1) and DIP 8 (autostart) to 'ON' and all other DIPs on that switch to 'OFF'. This will configure the LISY board to boot to MPF instead of the default PinMAME.

If you want to use the LISY board in "slave" mode where you run MPF on a separate computer and remotely control the LISY board, set DIP 6 to 'ON'. Then to control the mode that the LISY board will communicate with the host PC running MPF, set DIP 2 to 'ON' for network mode or 'OFF' for serial mode.

Note: If you are using a USB connection you have also to disconnect it in order to be able to reboot, as it will power the Raspberry Pi over the USB connection.

Switch S1								Mode
S1	S2	S3	S4	S5	S6	S7	S8	
off	off	off	on	off	off	off	on	MPF Master Mode
off	off	off	on	off	on	off	on	MPF Slave Mode (Serial)
off	on	off	on	off	on	off	on	MPF Slave Mode (Network)

3. Configure your Game

As usual, configure your specific Game Hardware via Switch 'S2'. For instance, for *Devils Dare*, which is internal number '18', set S2 DIP 2 and DIP 5 to 'ON' and all others to 'OFF' (binary coding of decimal 18).

A list of all game numbers is included in the LISY user manual.

4a. Add MPF config to SD Card (only needed for MPF Master Mode)

If you're using the "master" mode where MPF runs on the LISY board itself, you need to get your MPF config installed onto the LISY board. You can do this via the SD card.

Place your MPF config in the folder `/boot/mpfcfg/lisyx/yyy/` on the SD Card (replace "y" with 1 for LISY1 and with "80" for LISY80. Replace "xxx" with your game number with leading zeros if it's shorter than three digits). For instance with *Dare Devil*, the game would be at `/boot/mpfcfg/LISY80/018/` on the SD card.

It's easiest to do this with an SD card reader on your computer, though you could also copy the files using SSH connected to a running LISY controller (see the LISY user manual for details).

Again, we only recommend this option for your "final" config, as it's much easier to use the LISY board in slave mode and run MPF off your computer while you're developing your game.

Warning: This mode of operation will not allow you to run the MPF-MC since the LISY's Raspberry Pi Zero is not powerful enough. If you want to add an LCD or DMD to your machine, use the slave option detailed below.

4b. Connect your PC running MPF to LISY via network or serial (only needed for MPF Slave Mode)

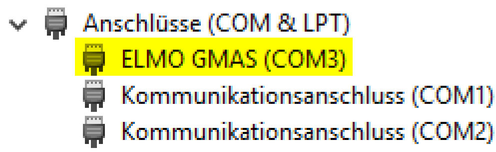
If you're using the "slave" mode where you run MPF on a standalone computer and then connect to the LISY board via the network or serial, once you configure the LISY board's DIP switches from Step 2 then you need to update your machine config file for MPF running on your computer to be able to connect to the LISY board.

Serial mode

If you want to use the serial port, add/update the following sections in your machine config:

```
hardware:
  platform: lisy
lisy:
  connection: serial
  port: com1              # replace this with your com port
  baud: 115200
```

Connection to LISY can be made via IP or via direct USB connection. For the USB connection no special driver Software nor a special USB cable is needed, a "normal" USB charging cable (Micro-USB cable) will do the job. Once connected to the host computer, it will (hopefully) identify a new serial device. This is usually `COMX` on windows:



Or `/dev/ttyACMX` on Linux:

```
dmesg

usb 1-3: new high-speed USB device number 11 using ehci-pci
usb 1-3: New USB device found, idVendor=0525, idProduct=a4a7
usb 1-3: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 1-3: Product: Gadget Serial v2.4
usb 1-3: Manufacturer: Linux 4.4.50+ with 20980000.usb
cdc_acm 1-3:2.0: ttyACM0: USB ACM device
usbcore: registered new interface driver cdc_acm
cdc_acm: USB Abstract Control Model driver for USB modems and ISDN adapters
```

Network mode

Alternatively, if you want to connect using WiFi or Ethernet, add/update the following sections in your machine config:

```
hardware:
  platform: lisy
lisy:
  connection: network
  network_port: 5963
  network_host: a.b.c.d    # replace this with the IP of LISY
```

LISY is configured to get its IP address by DHCP, the default hostname is `lisy`. For WLAN your WLAN-SSID and password can be put into a text file on the SD-card (see the LISY user manual for details). LISY will show the IP address on the first two displays of the pinball during boot time (or "NO IP" if no IP address could be found).

5. Power up LISY

Power up your system and enjoy.

5a. Start MPF (only needed for MPF Slave Mode)

Start MPF on you PC. Optionally start MPF-MC (if you want to use an additional DMD or LCD).

What if it did not work?

Have a look at our *LISY troubleshooting guide*.

Configuring Switches with LISY1

Related Config File Sections
switches:

LISY1 supports the System 1 switch matrix which consists of a maximum of 40 switches. The switch number in the manual of your machine can be used within MPF. However, some of the switches in Gottlieb System 1 games are **not** part of the switch matrix. These are the outhole switch, the SLAM switch and the "RESET" switch on the board itself. The `mpfserver` for LISY1 is numbering these switches in the same way as pinmame does it:

- SLAM: #76
- Outhole: #66
- Reset: #56

Note: As the SLAM switch is usually closed, the logic is inverted here. A closed SLAM switch is interpreted as open within `mpfserver` and does not have to be configured as normally closed `NC` in MPF.

You can start with this config:

```
switches:
  slam:
    number: 76
  outhole:
    number: 66
  reset:
    number: 56
```

Then just add your switches according to the manual of your machine. See *switches:* for more details about switches.

What if it did not work?

Have a look at our *LISY troubleshooting guide*.

Configuring Switches with LISY80

Related Config File Sections
switches:

LISY80 supports the System 80 switch matrix which consists of a maximum of 64 switches. The switch number in the manual of your machine can be used within MPF. However, you may not find all switches in your game manual as some switches are the same along all System80/80A/80B games and Gottlieb decided not to document them ;-). Those are the following (according to pinwiki.com):

- 06 - left advance button (Sys80B only)
- 07 - play / test switch

- 16 - right advance button (Sys80B only)
- 17 - left coin switch
- 27 - right coin switch
- 37 - center coin switch
- 47 - replay button
- 57 - plumb bob and ball roll tilts (these have the same switch assignment as the playfield tilt switch)

Note: The SLAM switch in system80, which is **not** part of the switch matrix and cannot be used in `mpfserver` for LISY80 in the current release.

You can start with this config:

```
switches:
  tilt:
    number: 57
```

Then just add your switches according to the manual of your machine. See *switches:* for more details about switches.

What if it did not work?

Have a look at our *LISY troubleshooting guide*.

Configuring Drivers in LISY

Related Config File Sections
coils:

Warning: Please ensure that you have established *common ground between logic and coil power* before turning on high voltage on your coils (especially on homebrew machines). Ignoring this might lock on your coils, overheat them, burn down your house or kill you. We are serious, floating grounds are dangerous. If you are not an electrical engineer read the *guide about voltages and power*.

In a nutshell: You need to connect your logic ground (5V/12V) and your high voltage ground (48V or 80V). A *power entry or power filter board* is a convenient solution to solve this (and more) issues.

Always turn all PSUs off when connecting power or you might fry all boards at once. This is generally a good idea but even more important when connecting more than one power supply to a board.

IF YOU DID NOT UNDERSTAND WHAT THIS WARNING MEANS STOP NOW AND TRY TO UNDERSTAND IT. OTHERWISE YOUR HARDWARE WILL LIKELY BURST INTO FLAMES AND YOU NEED TO WAIT A FEW DAYS FOR A REPLACEMENT OR EVEN WORSE IT MIGHT KILL YOU. IGNORING THIS IS THE MOST COMMON CAUSE FOR BROKEN DRIVER BOARDS.

Configure drivers according to the manual of your machine. LISY does not support any `hold_power` or `pulse_power` other than 1.0. So the coil will always enable with full power (which is fine in older machines and should not break things). However, you can still choose the pulse length using `pulse_ms`.

```
coils:
  c_some_coil:
    number: 04
    default_pulse_ms: 10
    allow_enable: true
```

In some Gottlieb machines coils were connected to the lights bank. To address those you have to add 100 to their number from the manual. For instance, to address a coil which is connected to the light output `05` use coil `105`:

```
coils:
  c_coil_on_light_bank:
    number: 107
    default_pulse_ms: 10
```

What if it did not work?

Have a look at our *LISY troubleshooting guide*.

Related How To guides

- *Coil Resistance and Hardware Details*
- *Wiring Dual Wound Coils*
- *Dual-Wound versus Single-Wound coils*
- *Adjust coil hold power*
- *Adjust coil strength (pulse times)*
- *Recycle / "Cool Down" Time*
- *Details About Flippers*
- *How to configure single-wound flippers*
- *How to configure dual-wound flippers*
- *Flipper end-of-stroke (EOS) switches*

Configuring and Enabling Flippers/Pop Bumpers/Slingshots in LISY

Related Config File Sections
digital_outputs:

System 1/80 does not support rules in software for flippers/pop bumpers/slingshots because CPUs were not fast enough at that time. Instead, they installed a hardware relay to enable flippers/pop bumpers/slingshots by connecting them physically to the corresponding switches (similar to fliptronics).

All you have to do is to configure the game_over_relay (which is connected as light) in LISY1 and LISY80:

```
digital_outputs:
  game_over_relay:
    number: 1
    type: light
    enable_events: ball_started
    disable_events: ball_will_end
```

In LISY35 the same relay is connected to a driver. You can use this example to enable flippers:

```
digital_outputs:
  flipper_enabling_relay:
    type: driver
    number: 16
    enable_events: ball_started
    disable_events: ball_will_end
```

This config will automatically enable the flippers on ball start and disable them on ball end. You can add more events to enable/disable them during the game.

What if it did not work?

Have a look at our *LISY troubleshooting guide*.

Configuring Lights in LISY

Related Config File Sections
lights:

Lights in LISY can be configured as *lights* using their number from the game manual.

This is an example:

```
lights:
  your_light:
    number: 03
```

There are some features in the light list like the game_over_relay which are not real lights. Those can be configured as *digital outputs*. See *Configuring and Enabling Flippers/Pop Bumpers/Slingshots in LISY* for details about the game_over_relay.

What if it did not work?

Have a look at our *LISY troubleshooting guide*.

Configuring Segment Displays in LISY

Related Config File Sections
segment_displays:
segment_display_player:

MPF can control all segment displays on your machine with LISY. Configure them like this:

```
segment_displays:
  info_display:
    number: 0
  player1_display:
    number: 1
  player2_display:
    number: 2
  player3_display:
    number: 3
  player4_display:
    number: 4
```

Note that the *Alpha-Numeric / Segment Displays* guide has more details on using alpha numeric and segment displays.

Video about segment displays:

https://youtu.be/Jyf3jxGXnTw

What if it did not work?

Have a look at our *LISY troubleshooting guide*.

Configuring Sound in LISY

Related Config File Sections
hardware_sound_systems:
hardware_sound_player:

With LISY your can use the sound card of your original game including all the sounds of your game.

Note: You can alternatively use the *built-in MPF sound system* which supports more modern audio features. In that case you need to connect the sound card of your PC to the audio amp of your machine (not covered here).

You can configure the external LISY *hardware sound interface* like this:

```
hardware_sound_systems:
  default:
    label: LISY
```

Built-in sounds

Any built-in sounds can be played using their number in the original game:

```
hardware_sound_player:
  some_event_to_play_sound2:
    2:
      action: play
  some_event_to_stop_any_playing_sound: stop
```

Whatever those sounds loop or do not depends on the sound and the game. In this case the event `some_event_to_play_sound2` will play the sound number 2. The event `some_event_to_stop_any_playing_sound` will stop any sound.

Additional sounds

You can play additional sounds by placing mp3 files on the SD-card. Soundfiles need to be placed in the mpf config directory on the SD card of the LISY system in the subdirectory `hardwaresounds`. For LISY1 this is `/boot/mpfcfg/LISY1/xxx` and for LISY80 this is `/boot/mpfcfg/LISY80/xxx` (where `xxx` is the game number set via S2 according to the appendix in the LISY user manual).

```
hardware_sound_player:
  play_file:
    "some_file": play_file
  play_file_loop:
    "some_file":
      action: play_file
      platform_options:
        loop: true
        no_cache: false
```

Text-to-speech

LISY can also do text-to-speech:

```
hardware_sound_player:
  event_to_play_text:
    text:
      action: text_to_speech
      value: "Hello MPF"
      platform_options:
        loop: false
        no_cache: true
```

Changing volume

Similarly, you can change volume:

```
hardware_sound_player:
  event_to_set_volume_to_05:
    set_volume:
      action: set_volume
      value: 0.5
  increase_volume:
    increase_volume:
      action: increase_volume
      value: 0.1
  decrease_volume:
    decrease_volume:
      action: decrease_volume
      value: 0.1
```

Sounds in a show

You can also use any of the actions above in a show instead of in a standalone *Hardware Sound player*:

```
##! show: test
- hardware_sounds:
    text:
      action: text_to_speech
      value: "Hello MPF"
      platform_options:
        loop: false
        no_cache: true
  duration: 2s
```

What if it did not work?

Have a look at our *LISY troubleshooting guide*.

LISY Protocol

The LISY protocol is a generic serial protocol to control pinball machines. It was developed for the *LISY platform* but is also used in other custom pinball platforms such as *APC*.

Theory of operations

All communication is initiated from the host PC. Commands are binary and generally have a fixed length. They may contain a length byte to indicate how many entries are to expect (i.e. three color values). Strings are zero terminated in both command and response.

At startup MPF resets the hardware and queries the count of all peripherals (i.e. switches, coils, lamps). Afterwards, it will query the state of switches and configure coils/lamps.

During the runtime MPF periodically polls changed switches and sends a watchdog every 500ms. The platform is expected to disable all outputs after 1s without watchdog.

Limitations

Let us know if you hit any of those and we can develop a plan forward.

- Max 127 switches are supported (because polling uses the upper bit as state)
- Max 256 hardware sounds (alternatively, you can use the MPF sound system)
- Max 256 simple lamps (on/off only)
- Max 256 lights (with fading and brightness)
- Max 256 coils (MPF currently only supports 100 of them; let us know if you need more)
- Max 7 alphanumeric displays (BCD, 7-segment or 14-segment displays)
- No error correction on the wire (your serial should be reliable)

Protocol reference (v0.08)

Table 1: General command format

Byte	Length	Description
0	1	Command Byte (see table below
1 - n	n - 1	Payload (n-1 bytes)

Table 2: String format (in both payload and response)

Byte	Length	Description
0 to (n-1)	n - 1	String
n	1	Null byte

Get Connected Hardware (0x00)

Get the name of the connected hardware. Does not have any payload.

Example:

Table 3: Command 0x00 - Get Connected Hardware

Byte	Length	Example	Comment
0	1	0	Command 0 - Get Connected Hardware

Returns a null terminated string.

Example: `LISY80`.

MPF uses this string to identify the platform and might perform certain quirks based on this info if necessary. Currently, there is quirk `coils_start_at_one` for `LISY1`, `LISY35` and `APC` to index coils starting at one instead of zero.

Get Firmware Version (0x01)

Get firmware version of the hardware board. Does not have any payload.

Example:

Table 4: Example Command 0x01 - Get Firmware Version

Byte	Length	Example	Comment
0	1	1	Command 1 - Get Firmware Version

Returns a null terminated string.

Example: `4.01`.

MPF parses this string as semantic version. It exposes the version as variable and in the logs. This might be used to perform quirks around known bugs.

Get API Version (0x02)

Get the API version. Does not have any payload.

Example:

Table 5: Example Command 0x02 - Get API Version

Byte	Length	Example	Comment
0	1	2	Command 2 - Get API Version

Returns a null terminated string.

Example: `0.08`.

MPF parses this string as semantic version. This is expected to be `0.08` for this version. MPF might refuse old API versions at some point.

Get Simple Lamp Count (0x03)

Get count of lamps connected to the hardware platform. Does not have any payload.

Example:

Table 6: Example Command 0x03 - Get Simple Lamp Count

Byte	Length	Example	Comment
0	1	3	Command 3 - Get Simple Lamp Count

Returns one byte:

Table 7: Response to 0x03 - Get Simple Lamp Count

Byte	Length	Description
0	1	Simple Lamp count 1 (0 to 255). 0 if no simple lamps exist.

Example:

Table 8: Example Response to 0x03 - Get Simple Lamp Count

Byte	Length	Example	Comment
0	1	64	Platform supports 64 simple lamps with numbers 0 to 63.

MPF uses this number to refuse any lights with a number larger or equal than `l` and subtype `lamp`. Lamps in LISY are expected to be `on/off` type devices and do not support fading or dimming. Use this for older style lamps and GIs.

Get Solenoid Count (0x04)

Get count of solenoids connected to the hardware platform. Does not have any payload.

Example:

Table 9: Example Command 0x04 - Get Solenoid Count

Byte	Length	Example	Comment
0	1	4	Command 4 - Get Solenoid Count

Returns one byte:

Table 10: Response to 0x04 - Get Solenoid Count

Byte	Length	Description
0	1	Solenoid count c (0 to 127). 0 if no solenoids exist.

Example:

Table 11: Example Response to 0x04 - Get Solenoid Count

Byte	Length	Example	Comment
0	1	64	Platform supports 64 solenoids with numbers 0 to 63.

MPF uses this number to refuse any solenoids with a number larger or equal than `c`.

Get Sound Count (0x05)

Get count of sounds available. Does not have any payload.

Example:

Table 12: Example Command 0x05 - Get Sound Count

Byte	Length	Example	Comment
0	1	5	Command 5 - Get Sound Count

Returns one byte:

Table 13: Response to 0x05 - Get Sound Count

Byte	Length	Description
0	1	Sound count o (0 to 255). 0 if no sounds exist.

Example:

Table 14: Example Response to 0x05 - Get Sound Count

Byte	Length	Example	Comment
0	1	128	Platform supports 128 sounds with numbers 0 to 127.

MPF uses this number to refuse any sounds with a number larger or equal than o. This is used for older machines with a hardware soundcard. In *LISY* it can be used to play sounds from the ROM of the original game. Return `0` if you do not support sounds in your platform.

Get Segment Display Count (0x06)

Get count of segment displays available. Does not have any payload.

Example:

Table 15: Example Command 0x06 - Get Segment Display Count

Byte	Length	Example	Comment
0	1	6	Command 6 - Get Segment Display Count

Returns one byte:

Table 16: Response to 0x06 - Get Segment Display Count

Byte	Length	Description
0	1	Segment display count sd (0 to 255). 0 if no sounds exist.

Example:

Table 17: Example Response to 0x06 - Get Segment Display Count

Byte	Length	Example	Comment
0	1	6	Platform supports 6 segment displays with numbers 0 to 5.

MPF uses this number to refuse any segment display with a number larger or equal than sd. Return `0` if you do not support displays in your platform.

Get Segment Display Details (0x07)

Get type of segment displays.

Table 18: Payload of Command 0x07 - Get Segment Display Details

Byte	Length	Description
1	1	Index `sd` of the segment display to query

Example:

Table 19: Example Command 0x07 - Get Segment Display Details

Byte	Length	Example	Comment
0	1	7	Command 7 - Get Segment Display Details
1	1	0	Query the first display

Returns two bytes:

Table 20: Response to 0x07 - Get Segment Display Details

Byte	Length	Description
0	1	Type of segment display (see list below)
1	1	Number of segments `sw(sd)` (0-255)

`sw(sd)` is the segment width for display index `sd`.

Example:

Table 21: Example Response to 0x07 - Get Segment Display Details

Byte	Length	Example	Comment
0	1	1	Segment display is a BCD7 display
1	1	12	Segment display is 12 segments wide

Options are:

Table 22: Types in Response of 0x07 - Get Segment Display Details

Byte of segment type `st`	Name	Description	Bytes per Segment `bs(st)`
0	Invalid	Display index is invalid or does not exist in machine.	•
1	BCD7	BCD Code for 7 Segment Displays without comma	1 byte (4 bit BCD in the first four byte)
2	BCD8	BCD Code for 8 Segment Displays (same as BCD7 but with comma)	1 byte (4 bit BCD in the first four byte, 7th byte is the comma)
3	SEG7	Fully addressable 7 Segment Display (with comma)	1 byte (a-g encoded as bit 0 to 6 and bit 7 as comma)
4	SEG14	Fully addressable 14 Segment Display (with comma)	2 bytes (a-g encoded as bit 0 to 6 in first byte. h to r encoded as bit 0 to 6 in second byte. comma as bit 7 in second byte)
5	ASCII	ASCII Code	1 ascii byte per segment
6	ASCII_DOT	ASCII Code with comma (every segment has an additional comma)	1 ascii byte per segment. Additionally bit 7 encodes the comma.

Not yet used in MPF but will be added soon.

Get Game Info (0x08)

Get the game number. Does not have any payload.

Example:

Table 23: Example Command 0x08 - Get Game Info

Byte	Length	Example	Comment
0	1	8	Command 8 - Get Game Info

Returns null terminated string. This is the internal Gottlieb number in LISY. MPF does not use the command at all (and we are not planning to). It is used in PinMAME on LISY.

Get Switch Count (0x09)

Get count of switches available. Does not have any payload.

Example:

Table 24: Example Command 0x09 - Get Switch Count

Byte	Length	Example	Comment
0	1	9	Command 9 - Get Switch Count

Returns one byte:

Table 25: Response to 0x09 - Get Switch Count

Byte	Length	Description
0	1	Switch count s (0 to 127)

Example:

Table 26: Example Response to 0x09 - Get Switch Count

Byte	Length	Example	Comment
0	1	70	Platform supports 70 switches with numbers 0 to 69.

MPF uses this number to refuse any switches with a number larger or equal than s. Please note that the procotol is currently limited to 127 switches since the upper byte is used to indicate inverted switches in commands.

Get Status of Simple Lamp (0x0A)

Get the status of a simple lamp. Payload is the lamp index:

Table 27: Payload of Command 0x0A - Get Status of Simple Lamp

Byte	Length	Description
1	1	Index 1 of the lamp to query

Example:

Table 28: Example Command 0x0A - Get Status of Simple Lamp

Byte	Length	Example	Comment
0	1	10	Command 10 - Get Status of Simple Lamp
1	1	25	Query status of lamp 25

Returns one byte:

Table 29: Response to 0x0A - Get Status of Simple Lamp

Byte	Length	Description
0	1	0=Off, 1=On, 2=Lamp not existing

Example:

Table 30: Example Response to 0x0A - Get Status of Simple Lamp

Byte	Length	Example	Comment
0	1	0	Status of lamp is off

MPF will not use this. After init/reset MPF assumes all lights to be in state off.

Set Status of Simple Lamp to On (0x0B)

Set simple lamp to on. Payload is the lamp index:

Table 31: Payload of Command 0x0B - Set Status of Simple Lamp to On

Byte	Length	Description
1	1	Index 1 of the lamp to set to on

Example:

Table 32: Example Command 0x0B - Set Status of Simple Lamp to On

Byte	Length	Example	Comment
0	1	11	Command 11 - Set Status of Simple Lamp to On
1	1	25	Set lamp 25 to on

No response is expected.

Set Status of Simple Lamp to Off (0x0C)

Set simple lamp to off. Payload is the lamp index:

Table 33: Payload of Command 0x0C - Set Status of Simple Lamp to Off

Byte	Length	Description
1	1	Index 1 of the lamp to set to off

Example:

Table 34: Example Command 0x0C - Set Status of Simple Lamp to Off

Byte	Length	Example	Comment
0	1	12	Command 12 - Set Status of Simple Lamp to Off
1	1	25	Set lamp 25 to off

No response is expected.

Get Status of Solenoid (0x14)

Get the status of a solenoid. Payload is the solenoid index:

Table 35: Payload of Command 0x14 - Get Status of Solenoid

Byte	Length	Description
1	1	Index c of the solenoid to query

Example:

Table 36: Example Command 0x14 - Get Status of Solenoid

Byte	Length	Example	Comment
0	1	20	Command 20 - Get Status of Solenoid
1	1	25	Query status of solenoid 25

Returns one byte:

Table 37: Response to 0x14 - Get Status of Solenoid

Byte	Length	Description
0	1	0=Off, 1=On, 2=Solenoid not existing

Example:

Table 38: Example Response to 0x14 - Get Status of Solenoid

Byte	Length	Example	Comment
0	1	0	Status of solenoid is off

MPF will not use this. After init/reset MPF assumes all solenoids to be in state disabled.

Enable Solenoid at Full Power (0x15)

Enable solenoid at full power. Payload is the solenoid index:

Table 39: Payload of Command 0x15 - Enable Solenoid at Full Power

Byte	Length	Description
1	1	Index c of the solenoid to enable

Example:

Table 40: Example Command 0x15 - Enable Solenoid at Full Power

Byte	Length	Example	Comment
0	1	21	Command 21 - Enable Solenoid at Full Power
1	1	25	Enable solenoid 25 at full power

No response is expected. This is mostly used in older machines where solenoids could be enabled without PWM.

Disable Solenoid (0x16)

Disable solenoid. Payload is the solenoid index:

Table 41: Payload of Command 0x16 - Disable Solenoid

Byte	Length	Description
1	1	Index c of the solenoid to disable

Example:

Table 42: Example Command 0x16 - Disable Solenoid

Byte	Length	Example	Comment
0	1	22	Command 22 - Disable Solenoid
1	1	25	Disable solenoid 25

No response is expected.

Pulse Solenoid (0x17)

Pulse solenoid with it's configured pulse time. Payload is the solenoid index:

Table 43: Payload of Command 0x17 - Pulse Solenoid

Byte	Length	Description
1	1	Index c of the solenoid to pulse

Example:

Table 44: Example Command 0x17 - Pulse Solenoid

Byte	Length	Example	Comment
0	1	23	Command 23 - Pulse Solenoid
1	1	25	Pulse solenoid 25

No response is expected. Use command 0x18 to configure the pulse time.

Set Solenoid Pulse Time (0x18)

Configure the pulse time of a solenoid in milliseconds. Payload is the solenoid index and pulse time.

Table 45: Payload of Command 0x18 - Set Solenoid Pulse Time

Byte	Length	Description
1	1	Index c of the solenoid to configure
2	1	Pulse time in ms (0-255)

Example:

Table 46: Example Command 0x18 - Set Solenoid Pulse Time

Byte	Length	Example	Comment
0	1	21	Command 24 - Set Solenoid Pulse Time
1	1	25	Configure solenoid 25
2	1	50	Set pulse time to 50ms

No response is expected. This will affect pulses in command 0x17.

Set Segment Display 0-6 (0x1E - 0x24)

Set content of segment display `d` 0-6. Payload is a null terminated string. Content encoding depends on the type of the display (from command 0x7).

Table 47: Command 0x1E - 0x24 - Set Segment Display `d`

Byte	Length	Value	Comment
0	1	30 + d	Command byte for set segment depending on segment number `d`
1	1	`sw(sd) * bs(st)`	Bytes which will follow. Number of segments (0-127) multiplied by bytes per segment for this display (1 or 2 bytes).
2	`sw(sd) * bs(st)`	Number of segments (0-127) multiplied by bytes per segment for this display (1 or 2 bytes)	One or two bytes per segment for all segments. Encoding depends on segment type (see command 0x7).

Example:

Table 48: Example Command 0x1E - 0x24 - Set Segment Display `d`

Byte	Length	Example	Comment
0	1	31	Command 31 - Set Segment display 1
1	1	12	12 Bytes will follow
2	12	Hello World!	Set display1 to hello world (ASCII type display)

No response is expected.

Get Status of Switch (0x28)

Get the status of a switch. Payload is the switch index:

Table 49: Payload of Command 0x28 - Get Status of Switch

Byte	Length	Description
1	1	Index s of the switch to query

Example:

Table 50: Example Command 0x28 - Get Status of Switch

Byte	Length	Example	Comment
0	1	40	Command 40 - Get Status of Switch
1	1	25	Query status of switch 25

Returns one byte:

Table 51: Response to 0x28 - Get Status of Switch

Byte	Length	Description
0	1	0=Off, 1=On, 2=Switch not existing

Example:

Table 52: Example Response to 0x28 - Get Status of Switch

Byte	Length	Example	Comment
0	1	0	Status of switch is off

MPF will read all switches at startup using this command.

Get Changed Switches (0x29)

Check is switches changed. Does not have any payload.

Example:

Table 53: Example Command 0x29 - Get Changed Switches

Byte	Length	Example	Comment
0	1	41	Command 41 - Get Changed Switches

Returns one byte:

Table 54: Response to 0x29 - Get Changed Switches

Byte	Length	Description
0	1	127=No change. Otherwise: The numer of changed switch. Bit 7 is the status of that switch.

Example:

Table 55: Example Response to 0x29 - Get Changed Switches

Byte	Length	Example	Comment
0	1	10	Switch 10 turned off

MPF will poll this at 100 Hz by default.

Play Sound (0x32)

Play a sound on a hardware sound card. This is used to trigger sounds on existing sound interfaces on older machines. The behavior of sounds usually differs per sound number (looping/not looping/stop other sounds etc) and cannot be influenced by the CPU.

Payload is the sound number.

Table 56: Payload of Command 0x32 - Play Sound

Byte	Length	Description
1	1	Track to play (default track is 1)
2	1	Index of sound to play

Example:

Table 57: Example Command 0x32 - Play Sound

Byte	Length	Example	Comment
0	1	50	Command 50 - Play Sound
1	1	1	Play on track 1
2	1	42	Play sound 42

No response is expected.

Stop Sound (0x33)

Stop the current playing sound.

Payload is the sound number.

Table 58: Payload of Command 0x33 - Stop Sound

Byte	Length	Description
1	1	Track to stop (default track is 1)

Example:

Table 59: Example Command 0x33 - Stop Sound

Byte	Length	Example	Comment
0	1	51	Command 51 - Stop Sound
1	1	1	Stop all sounds on track 1

No response is expected.

Play Sound File (0x34)

Play a sound file on external hardware. This is used to extend sound capabilities on older machines in LISY. Alternatively, you can use the MPF sound system.

Payload is a null terminated string containing track, flags and the filename of the sound.

Table 60: Payload of Command 0x34 - Play Sound File

Byte	Length	Description
1	1	Track to play (default track is 1)
2	1	Flags (bit 0=loop, 1=no cache)
3	n	Filename (length n)
3 + n	1	Null terminator

Example:

Table 61: Example Command 0x34 - Play Sound File

Byte	Length	Example	Comment
0	1	52	Command 52 - Play Sound File
1	1	1	Use Track 1
2	1	1	Loop file
3	9	test.mp3	Play sound test.mp3. Last character is null byte.

No response is expected.

Text to speech (0x35)

This is used to extend sound capabilities on older machines in LISY.

Payload is a null terminated string containing track, flags and the text to play.

Table 62: Payload of Command 0x35 - Text to speech

Byte	Length	Description
1	1	Track to play (default track is 1)
2	1	Flags (bit 0=loop, 1=no cache)
3	n	Text to play (length n)
3 + n	1	Null terminator

Example:

Table 63: Example Command 0x35 - Text to speech

Byte	Length	Example	Comment
0	1	53	Command 53 - Text to speech
1	1	Track to play (default track is 1)	
2	1	1	No loop. Use Cache.
3	6	Hello	Play text 'hello'. Last character is null byte.

No response is expected.

Set Sound Volume (0x36)

Set volume of amplifier. This may be connected either to a hardware soundcard or to the output of the MPF sound system.

Payload is the sound number.

Table 64: Payload of Command 0x36 - Set Sound Volume

Byte	Length	Description
1	1	Volume in percent (0-100)
2	1	Track to change (default track is 1)

Example:

Table 65: Example Command 0x36 - Set Sound Volume

Byte	Length	Example	Comment
0	1	54	Command 54 - Set Sound Volume
1	1	Change track 1	
2	1	50	Set volume to 50%

No response is expected.

Init/Reset (0x64)

Reset and initialize the platform. MPF will expect this command to reset all coil configs and to disable all coils and lights. Does not have any payload.

Example:

Table 66: Example Command 0x64 - Init/Reset

Byte	Length	Example	Comment
0	1	100	Command 100 - Init/Reset

Returns one byte:

Table 67: Response to 0x64 - Init/Reset

Byte	Length	Description
0	1	0=OK. Otherwise an error code. MPF will retry on error.

Example:

Table 68: Example Response to 0x64 - Init/Reset

Byte	Length	Example	Comment
0	1	0	Reset ok.

This will be the first command send by MPF.

Watchdog (0x65)

Will be send every 500ms. The hardware is expected to disable all solenoids and light if it did not get a watchdog for 1s. Does not have any payload.

Example:

Table 69: Example Command 0x65 - Watchdog

Byte	Length	Example	Comment
0	1	101	Command 101 - Watchdog

Returns one byte:

Table 70: Response to 0x65 - Watchdog

Byte	Length	Description
0	1	0=OK. Otherwise an error code

Example:

Table 71: Example Response to 0x65 - Watchdog

Byte	Length	Example	Comment
0	1	0	Watchdog ok.

This be send periodically at 2 Hz in MPF.

Protocol reference (v0.09) - RFC

This section contains a proposal for new methods. This is still in development. Requests and commends are welcome. All commands are considered in “Request for Comments (RFC)” state. They will likely end up in v0.09 in some way.

Get Count of Modern Lights (0x13)

Get count of modern lights available. Does not have any payload.

Example:

Table 72: Example Command 0x13 - Get Count of Modern Lights

Byte	Length	Example	Comment
0	1	19	Command 19 - Get Count of Modern Lights

Returns one byte:

Table 73: Response to 0x13 - Get Count of Modern Lights

Byte	Length	Description
0	1	Light count `m` (0 to 255). 0 if no modern lights exist.

Example:

Table 74: Example Response to 0x13 - Get Count of Modern Lights

Byte	Length	Example	Comment
0	1	128	Platform supports 128 modern lights with numbers 0 to 127.

MPF uses this number to refuse any lights with a number larger or equal than `m` and subtype `light`. Return `0` if you do not support modern lights in your platform.

Fade Modern Light (0x0d)

Fade a group of modern lights.

Table 75: Payload of Command 0x0d - Fade Modern Light

Byte	Length	Description
1	1	Index `m` of the first light
2	2	Fade time in ms (0-65535). Can be 0 to set the brightness instantly.
4	1	Number `n` of lights to fade. Can be 1 to set or fade a single light.
5	`n`	One byte of brightness per light (0-255). `n` bytes in total

Example:

Table 76: Example Command 0x0d - Fade Modern Light

Byte	Length	Example	Comment
0	1	19	Command 13 - Fade Modern Light
1	1	42	First light is 42
2	2	50	Fade to color in 50ms.
4	1	3	Fade three lights (i.e. RGB in sync)
5	1	127	Fade light 42 to 50% brightness
6	1	0	Fade light 43 to 0% brightness
7	1	255	Fade light 44 to 100% brightness

No response is expected.

Set Solenoid Recycle Time (0x19)

Configure the recycle time of a solenoid in milliseconds. The platform will prevent any new pulse/enable until recycle time has passed after a pulse end or disable. This prevents overheating through "machine gunning" on pops, flaky switches or repeated pulses through bad code. By default MPF will set recycle to two times the pulse time but it can be changed.

Payload is the solenoid index and recycle time.

Table 77: Payload of Command 0x19 - Set Solenoid Recycle Time

Byte	Length	Description
1	1	Index c of the solenoid to configure
2	1	Recycle time in ms (0-255)

Example:

Table 78: Example Command 0x19 - Set Solenoid Recycle Time

Byte	Length	Example	Comment
0	1	25	Command 25 - Set Solenoid Recycle Time
1	1	25	Configure solenoid 25
2	1	50	Set recycle time to 100ms

No response is expected. This will affect pulses, enables and all hardware rules.

Pulse and Enable Solenoid with PWM (0x1A)

Pulse solenoid and then enable solenoid with PWM. Payload is the solenoid index, pulse time, pulse power and hold power:

Table 79: Payload of Command 0x1A - Pulse and Enable Solenoid with PWM

Byte	Length	Description
1	1	Index c of the solenoid to enable
2	1	Pulse time in ms (0-255)
3	1	Pulse PWM power (0-255). 0=0% power. 255=100% power
4	1	Hold PWM power (0-255). 0=0% power. 255=100% power

Example:

Table 80: Example Command 0x15 - Pulse and Enable Solenoid with PWM

Byte	Length	Example	Comment
0	1	26	Command 26 - Enable Solenoid with PWM and Pulse
1	1	25	Enable solenoid 25
2	1	30	30ms initial pulse
3	1	191	191/255 = 75% pulse power
4	1	64	25% hold power

No response is expected. This command can also be used to just pulse a coil with PWM if "Hold PWM power" is set to 0.

Configure Hardware Rule for Solenoid (0x3C)

Program a hardware rule into the controller to control a solenoid based on one to three switches. This is used in modern machines to implement low latency responses (because responding to switch hits in software causes too much latency and jitter). There can be only one hardware rule per solenoid. A new rule will always overwrite an old one for the solenoid.

Flags decide what the three switches do:

Table 81: Flags for Command 0x3C - Configure Hardware Rule for Solenoid

Bit	Description
0	When switch becomes active trigger the rule. Usually set on the first switch to trigger the rule. Sometimes a second switch is used just to disable a rule (such as on EOS of a flipper).
1	When switch becomes inactive disable the rule. This is what you want on flipper fingers but not on slings/pops.
2	reserved
3	reserved
4	reserved
5	reserved
6	reserved
7	reserved

Payload is the solenoid index, one to three switches, pulse time, pulse power, hold power and some flags:

Table 82: Payload of Command 0x3C - Configure Hardware Rule for Solenoid

Byte	Length	Description
1	1	Index `c` of the solenoid to configure
2	1	Switch `sw1`. Set bit 7 to invert the switch.
3	1	Switch `sw2`. Set bit 7 to invert the switch.
4	1	Switch `sw3`. Set bit 7 to invert the switch.
5	1	Pulse time in ms (0-255)
6	1	Pulse PWM power (0-255). 0=0% power. 255=100% power
7	1	Hold PWM power (0-255). 0=0% power. 255=100% power
8	1	Flag for `sw1`
9	1	Flag for `sw2`
10	1	Flag for `sw3`

Example:

Table 83: Example Command 0x3C - Configure Hardware Rule for Solenoid

Byte	Length	Example	Comment
0	1	60	Command 60 - Configure Hardware Rule for Solenoid
1	1	25	Configure rule for solenoid 25
2	1	5	Use Switch 5 as `sw1`
3	1	134	Use inverted Switch 6 as `sw2`
4	1	127	No switch as `sw3`
5	1	30	30ms initial pulse
6	1	191	191/255 = 75% pulse power
7	1	64	25% hold power
8	1	3	`sw1` will enable the rule and disable it when released.
9	1	2	`sw2` will disable the rule if it closes (because it is inverted).
10	1	0	Do not use `sw3`

No response is expected. To disable a rule just set all flags to 0.

Troubleshooting LISY

If you got problems with your hardware platform we first recommend to read our *troubleshooting guide*. Here are some hardware platform specific steps:

Run Hardware Scan

Using `mpf hardware scan` you can find out if your LISY based platform is talking properly to MPF. Additionally, it will show you details about the hardware:

```
$ mpf hardware scan

LISY connected via network at localhost:1234
Hardware: LISY1 Lisy Version: 4.01 API Version: 0.8
Input count: 88 Input map: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13',
↪'14', '15', '16', '17', '18', '19', '20', '21', '22', '23', '24', '25', '26', '27', '28', '29', '30',
↪'31', '32', '33', '34', '35', '36', '37', '38', '39', '40', '41', '42', '43', '44', '45', '46', '47',
↪'48', '49', '50', '51', '52', '53', '54', '55', '56', '57', '58', '59', '60', '61', '62', '63', '64',
↪'65', '66', '67', '68', '69', '70', '71', '72', '73', '74', '75', '76', '77', '78', '79', '80', '81',
↪'82', '83', '84', '85', '86', '87']
Coil count: 9
Modern lights count: 0
Traditional lights count: 40
Display count: 5
```

See *mpf hardware (command-line utility)* for details about the command.

Enable Debugging

If you got problems with your platform try to enable debug first. As described in the *general debugging section* of our *troubleshooting guide* this is done by adding `debug: true` to your `lisy` config section:

```
lisy:
  debug: true
```

This will add a lot more debugging and might slow down MPF a bit. We recommend to disable/remove it after finishing debugging.

Coils Are Not Firing

What to do if your coils are not working?

Check if Your Hardware is Working at all

Sounds stupid but this is a good start: Is the hardware working at all? Do you see switch hits in the logs? If not, check our section `Your hardware is not working at all`.

Check the Watchdog

If switches (or other features of the platform) are working but coils are not we have to dig deeper. Most hardware platforms have some kind of watchdog. Often there is some LED which indicates if the watchdog is received. The MPF log might also contain clues (especially if you have enabled debug and run MPF with verbose flags `-v -V`). If the watchdog is not received by your platform it will not enable coils.

In most cases watchdog related problems indicate wiring problems. Check if your boards are properly wired.

Test Your Coil Numbers using MPF Service CLI

Hardware is connected and generally working, watchdog is good but still your coils are not working? Maybe something with the numbering is odd. Lets tests that using the *MPF Service CLI*. Alternatively, you can also use *service mode* if you have already configured it. Both ways work similarly.

To use service cli:

1. Open two consoles
2. Start your game (e.g. using `mpf both`)
3. Start the service cli from within your game folder using `mpf service`.
4. Type `list_coils` and press `ENTER` to see a list of coils.
5. Type `coil_pulse your_coil` and press `ENTER` to pulse it.

Does it work? If not check the log and try verify the coil number. If you do not specify `default_pulse_ms` MPF will use `10ms` which might not be enough for some mechs. Try to increase that gently (maybe `20ms` or `30ms`).

Reducing light update rate

If you got a lot of lights you might run into bus contention issues. You can reduce the light update rate in MPF:

```
mpf:
  default_light_hw_update_hz: 30   # defaults to 50
```

If you set this too low fades will be less smooth but otherwise it should not affect your game.

Your hardware is not working at all

If your hardware is not working at all make sure that you removed the options `-X`, `-x` and `--vpx` from your `mpf both` or `mpf game` command line. Those options will overwrite the settings in your `hardware` section and MPF will not even try to connect to your hardware. If you got config errors we suggest you add `-X` to figure things out without interfacing real hardware all the time. Just keep that option in mind.

Another stupid thing to check: Is your hardware connected to your PC? We know it is stupid but a loose USB connector has happened to most of us.

On Linux you might want to run the command `lsusb` which should show both of your micro controllers connected. You should see two lines similar to

```
Bus 002 Device 014: ID 0483:5740 STMicroelectronics Virtual COM Port
Bus 002 Device 015: ID 0483:5740 STMicroelectronics Virtual COM Port
```

If you are unsure about the output, run the command once with your controllers connected and once without. If there is no difference, then for sure the USB device is not properly connected.

Add debugging to related devices

If you got problems with some switches also add `debug: true` to those as it will give to more insights into the intentions of those devices. Same will work for flippers, coils, lights, servos, steppers and more. See *general debugging section* for details.

Run MPF with verbose flag

See *general debugging section* for details. TLDR: run `mpf both -t -v -V`.

Report Your Issue and Ask For Help

If you cannot find the issue yourself please prepare some information about your issue according to our *troubleshooting guide* and ask in our forum.

Consider Improving the Documentation

Did you solve your issue but found that some relevant information in the documentation is missing or should be linked/located elsewhere? Either tell us in the forum or consider *improving the* *documentation* yourself to save future users some troubles the same way others saved you some troubles by writing this documentation.

Arduino Pinball Controller

MPF can control System3 to System11c machines directly using the Arduino Pinball Controller (APC). It contains CPU and drivers so it can also be used to build full custom machines. Uses the *LISY protocol*.

This is how APC generally works:

https://youtu.be/w4Po8OE5Zkw

See Arduino Pinball Controller Documentation on github for details.

This is an example config:

```
#config_version=5
hardware:
  platform: lisy
lisy:
  connection: serial
  port: com1      # change this for your setup
  baud: 115200
digital_outputs:
  game_over_relay:
    number: 1
    type: light
    enable_events: ball_started
    disable_events: ball_will_end
segment_displays:
  info_display:
    number: 0
  player1_display:
    number: 1
  player2_display:
    number: 2
  player3_display:
    number: 3
  player4_display:
    number: 4
hardware_sound_systems:
  default:
    label: APC
```

See the *LISY platform* for more details on configuring hardware.

Connecting a System3 to System11c Machine to APC

1. Replace your original MPU and driver board with APC

See the APC documentation for build up and installation instructions.

2. Configure APC to Run MPF

Select USB Control in APC to ensure that MPF can connect. If you did not install an SD card (not needed for MPF if you do not want to use the sound card of APC) this should be the default.

3. Connect your PC running MPF to APC via USB

Connect APC to your PC via USB. The arduino on APC will behave as a USB-serial device and your PC should show a new serial device. For the USB connection no special driver Software nor a special USB cable is needed, a "normal" USB A-B cable will do the job. APC uses the *LISY protocol* which is why we have to configure it similarly.

Add/update the following sections in your machine config:

```
hardware:
  platform: lisy
lisy:
  connection: serial
  port: com1              # replace this with your com port
  baud: 115200
```

Once connected to the host computer, it will (hopefully) identify a new serial device. This is usually `COMx` on windows or `/dev/ttyUSBx` on Linux.

4. Power up APC

Power up your system and enjoy.

5. Start MPF

Start MPF and MPF-MC on you PC:

```
mpf both
```

What if it does not work?

Have a look at the *LISY troubleshooting guide*.

How to configure Multimorphic (P-ROC & P3-ROC) hardware

Here's a list of all the How To guides which explain how to use MPF with Multimorphic P-ROC and P3-ROC control systems. These guides include the numbering format (how you map specific entries in your config files to board and connector locations) as well as overall settings that affect how your hardware performs.

Overview video about the Multimorphic P3-Roc system:

https://youtu.be/rLKUrv4hWqI

3 steps to using a P-ROC/P3-ROC

1. *Install the hardware drivers to support the P-ROC/P3-ROC*.
2. *Configure your platform*.
3. Configure the individual pinball mechanisms from the list below.

P-ROC/P3-ROC pinball mech configuration

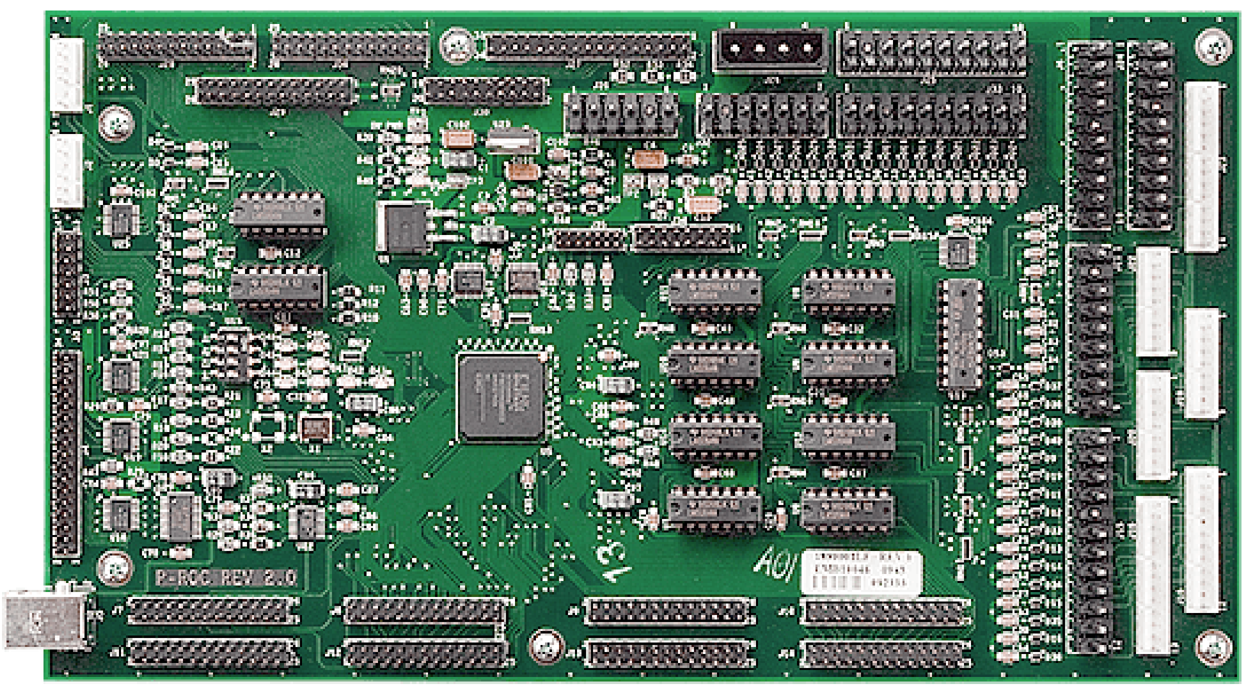
The following pinball mechanisms are supported by the P-ROC and/or P3-ROC. Click each one for details on how to configure these types of mechanisms for the P-ROC or P3-ROC.

Connecting P/P3-Roc to your Computer

This page is about connecting the P/P3-Roc to your computer. It roughly covers connecting the bus between the nodes. For electronic details see the P-Roc section in the pinballmakers.com Wiki.

P-Roc

If you got a P-Roc just connect it to your computer using USB.

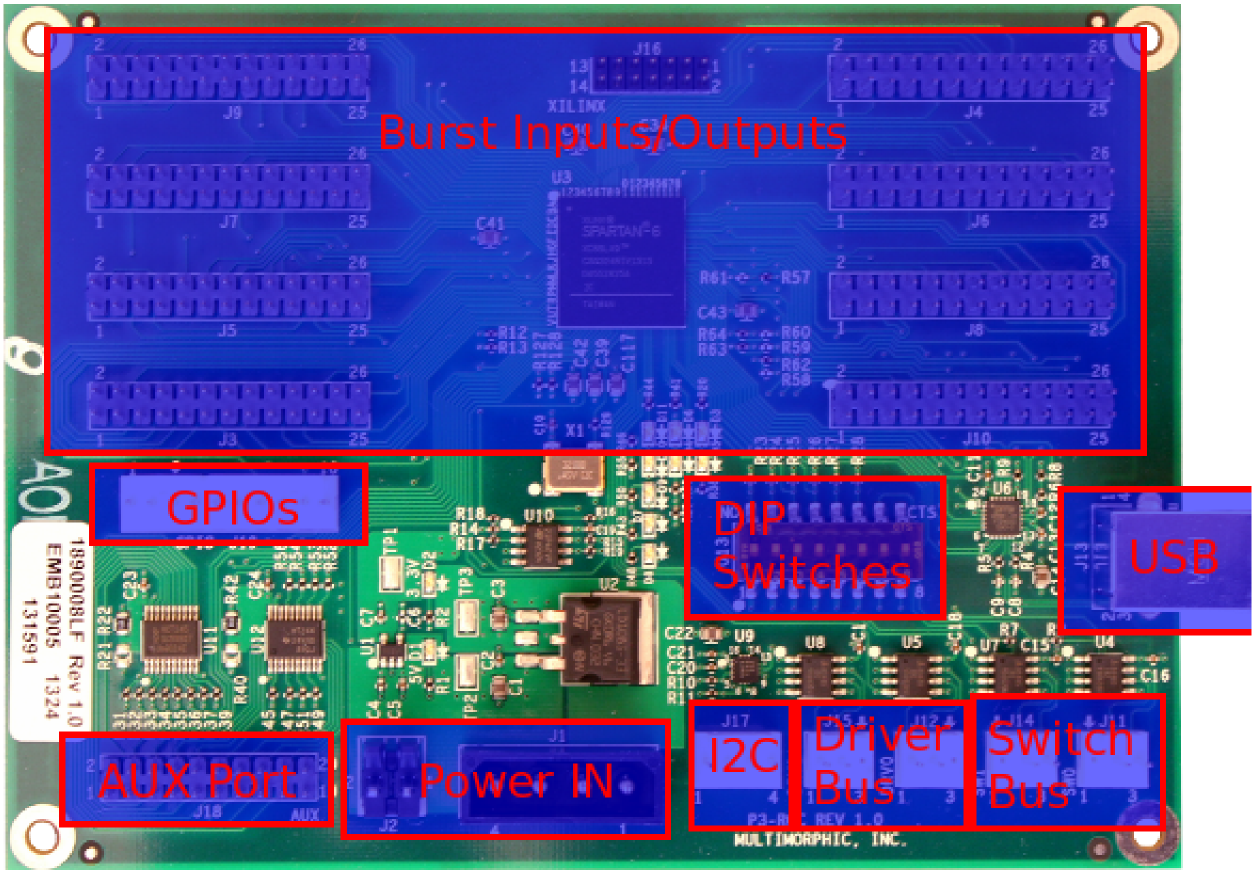


Then connect switches and driver according to the manual (see *Using MPF with existing pinball machines (Williams, Stern, Gottlieb, etc.)* for specific machines). If you are using a PD-Master board see below for switches and drivers.

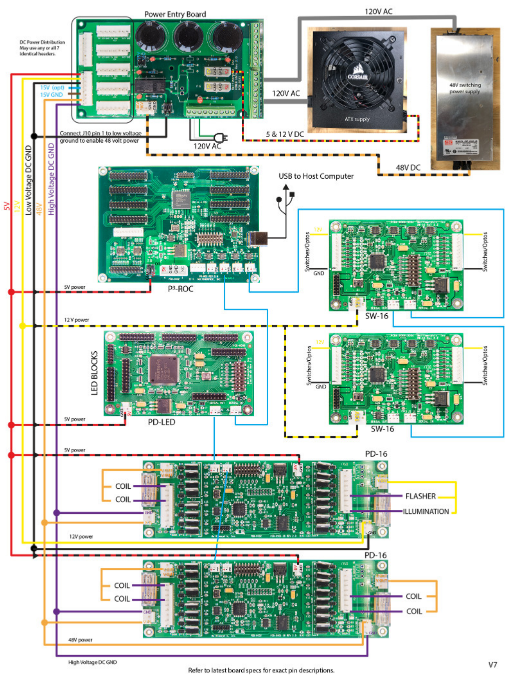
mpf hardware scan will show the firmware version and revision of your P-Roc if it is connected correctly.

P3-Roc

If you got a P3-Roc just connect it to your computer using USB.

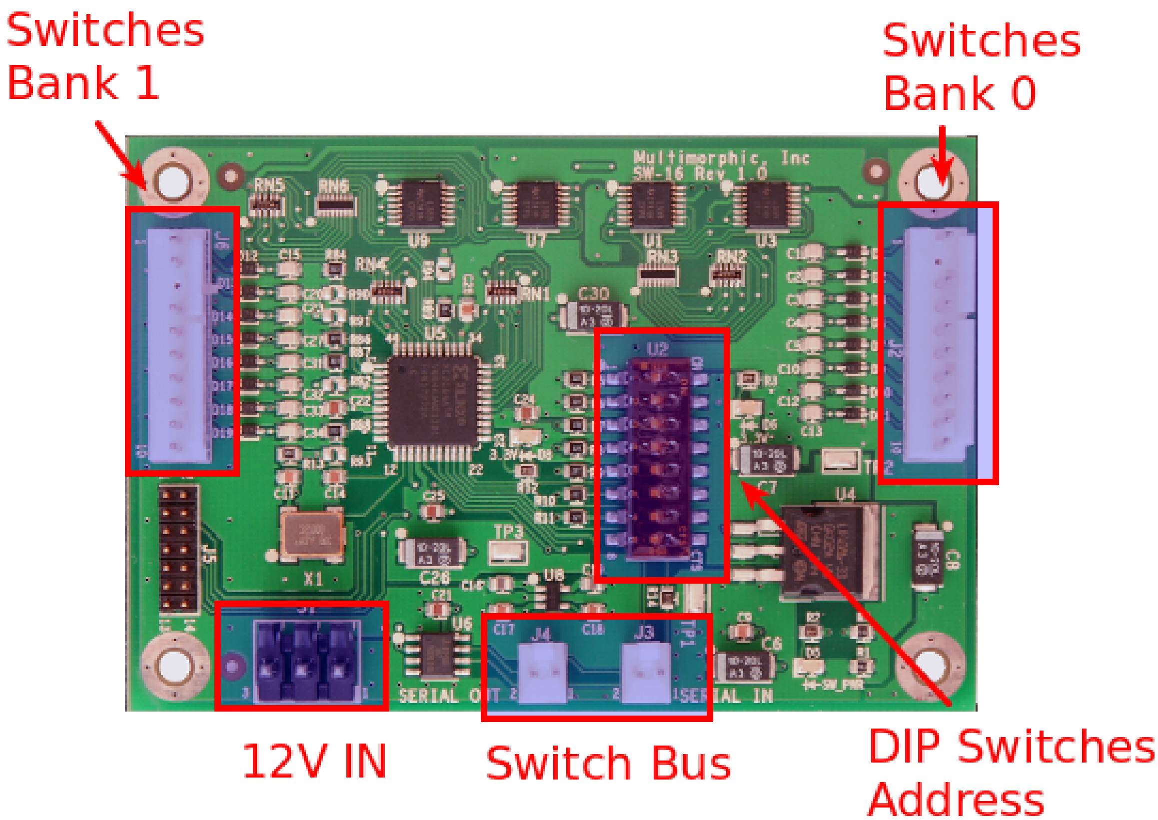


Connect all your SW-16 boards to the switch bus and all your PD-16 and PD-8x8 boards to your driver bus. Use twisted wires but connect + to + and - to - on all nodes.



mpf hardware scan will show the firmware version and revision of your P3-Roc if it is connected correctly.

SW-16

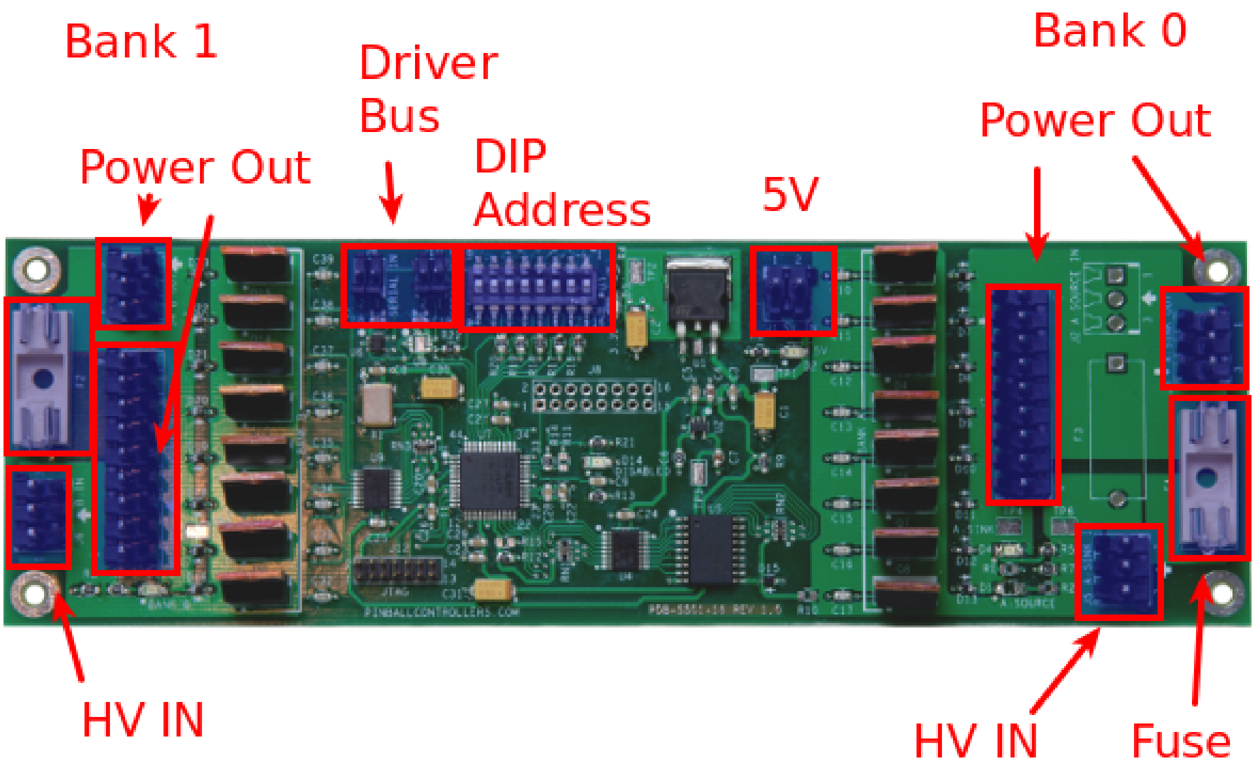


Set a unique address on every SW-16 board on your bus. Those addresses can overlap with the driver addresses. It does not matter on which of the two switch busses the boards are connected. Terminate the bus at the last board. See *How to configure switches (P3-ROC)* for how to configure those boards.

You can list all SW-16 using *mpf hardware scan*:

```
$ mpf hardware scan

Firmware Version: 2 Firmware Revision: 6 Hardware Board ID: 1
SW-16 boards found:
 - Board: 0 Switches: 16 Device Type: A3 Board ID: 0
 - Board: 1 Switches: 16 Device Type: A3 Board ID: 1
 - Board: 2 Switches: 16 Device Type: A4 Board ID: 2
```

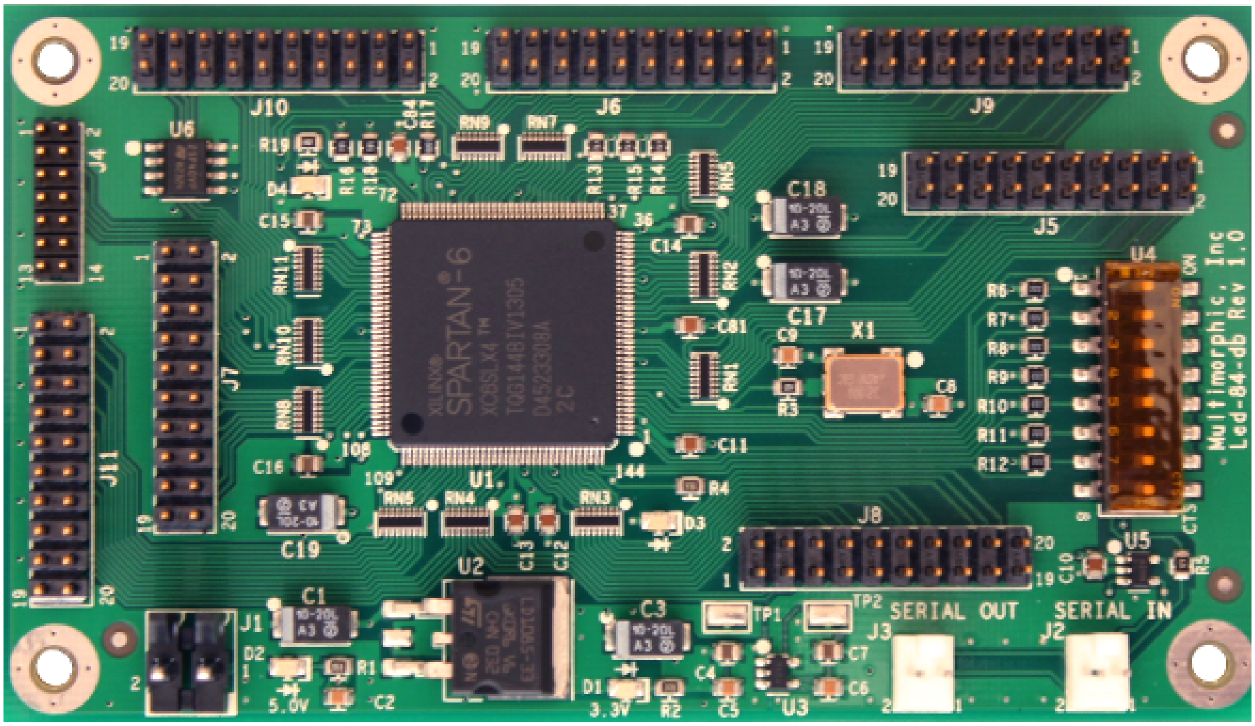
PD-16/PD-8x8



Set a unique address on every PD-16/PD-8x8 board on your bus. Those addresses can overlap with the switch addresses. However, they overlap with the PD-LED addresses so plan accordingly. It does not matter on which of the two driver busses the boards are connected. Terminate the bus at the last board. See *How to configure coils/drivers/magnets (P-ROC/P3-ROC)* and *How to configure Matrix Lights (P-ROC/P3-ROC)* for how to configure those boards.

MPF and the P3-Roc do not know if those boards are connected as the communication is one-way only.

PD-LED



Set a unique address on every PD-LED board on your bus. Those addresses can overlap with the switch addresses. However, they overlap with the PD-16 addresses so plan accordingly. It does not matter on which of the two driver busses the boards are connected. Terminate the bus at the last board. See *How to configure LEDs on the PD-LED (P-ROC/P3-ROC)* for how to configure those boards.

MPF and the P3-Roc do not know if those boards are connected as the communication is one-way only.

What if it did not work?

Have a look at our *troubleshooting guide for the P/P3-Roc*.

How to use install drivers for the P-ROC / P3-ROC

Using a P-ROC or P3-ROC with MPF is pretty straightforward. The first step is to download and install the hardware drivers and libraries for your OS that the P-ROC/P3-ROC needs to communicate with your computer. The exact process for that is OS-specific, so click the link to follow the guide for your specific OS:

How to install P-ROC / P3-ROC drivers on Windows (32-bit)

This guide explains how to install the USB drivers for the P-ROC or P3-ROC on 32-bit Windows (x86).

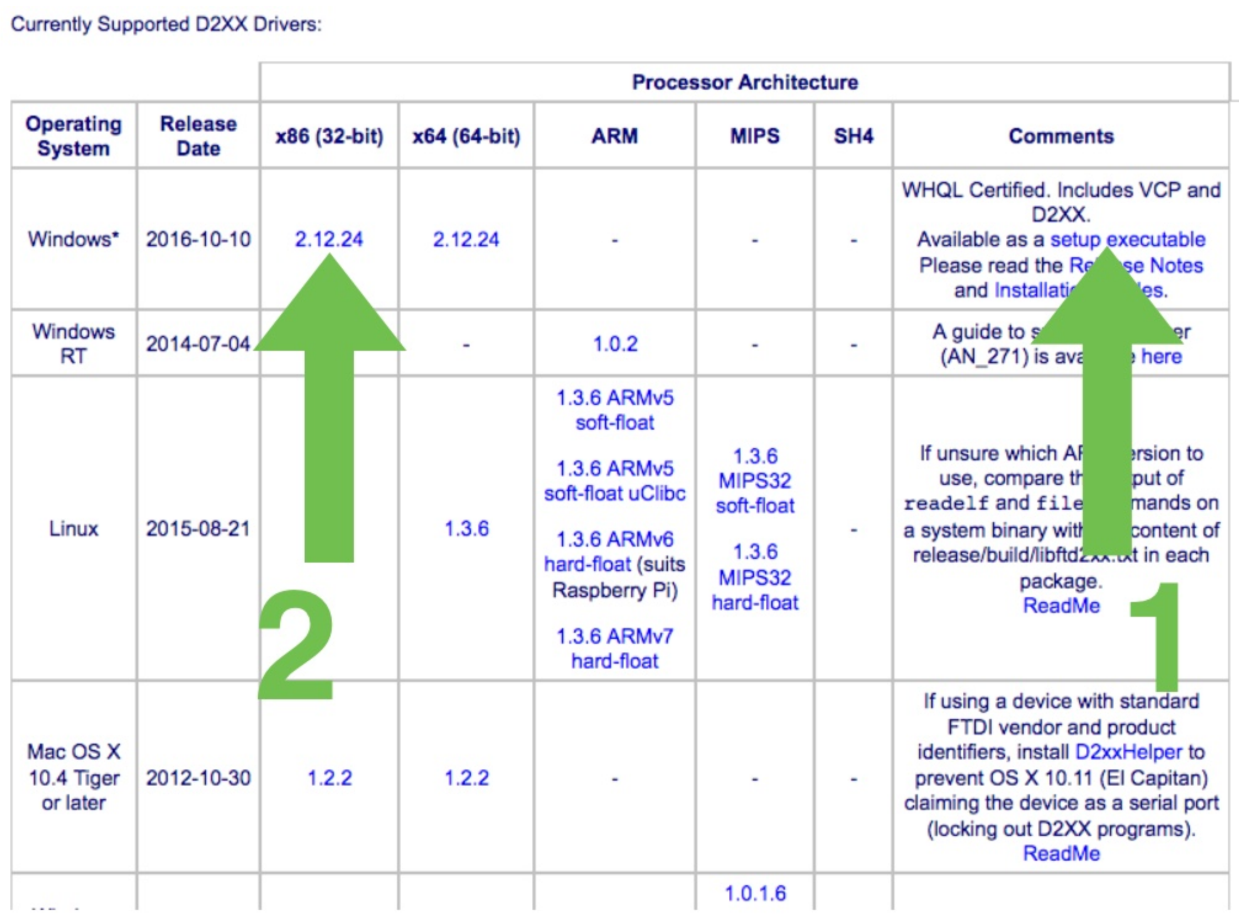
1. Download and install the FTDI drivers

The P-ROC and P3-ROC boards use a chip from a company called "FTDI Chip" to handle the USB communication, so you need to install the FTDI driver so Windows can properly see the device when you plug it in.

You can download the latest version from here:

http://www.ftdichip.com/Drivers/D2XX.htm

Here's a screen shot of the download section of that page. Note that the actual version number of the driver might be newer that the screen shot below. That should be ok.



Currently Supported D2XX Drivers:

Operating System	Release Date	Processor Architecture x86 (32-bit)	x64 (64-bit)	ARM	MIPS	SH4	Comments
Windows*	2016-10-10	2.12.24	2.12.24	-	-	-	WHQL Certified. Includes VCP and D2XX. Available as a setup executable Please read the Re se Notes and Installati es.
Windows RT	2014-07-04		-	1.0.2	-	-	A guide to s er (AN_271) is ava here
Linux	2015-08-21		1.3.6	1.3.6 ARMv5 soft-float 1.3.6 ARMv5 soft-float uClibc 1.3.6 ARMv6 hard-float (suits Raspberry Pi) 1.3.6 ARMv7 hard-float	1.3.6 MIPS32 soft-float 1.3.6 MIPS32 hard-float	-	If unsure which A rsion to use, compare t put of `readelf` and `file` mands on a system binary wit content of release/build/libftd2 t in each package. ReadMe
Mac OS X 10.4 Tiger or later	2012-10-30	1.2.2	1.2.2	-	-	-	If using a device with standard FTDI vendor and product identifiers, install D2xxHelper to prevent OS X 10.11 (El Capitan) claiming the device as a serial port (locking out D2XX programs). ReadMe
					1.0.1.6		

Download and run the setup executable from the "1" link in the screen shot. (We like to use that because it's easier than the manual process you get from using the "2" link in that screen shot.)

Now MPF will be able to communicate with the P-ROC or P3-ROC.

Continue on with the *How to configure MPF for the P-ROC/P3-ROC platform* documentation to finish your MPF configuration for the P-ROC/P3-ROC.

2. Install Visual C++ Redistributable for Visual Studio 2019

You might already have those but in case you do not install Visual C++ Redistributable for Visual Studio 2019 (32-bit).

What if it did not work?

MPF is erroring out on start-up? Cannot connect to your P/P3-Roc? Have a look at the *troubleshooting guide for P/P3-Roc*.

How to install P-ROC / P3-ROC drivers on Windows (64-bit)

This guide explains how to install the USB drivers for the P-ROC or P3-ROC on 64-bit Windows (x64).

1. Download and install the FTDI drivers

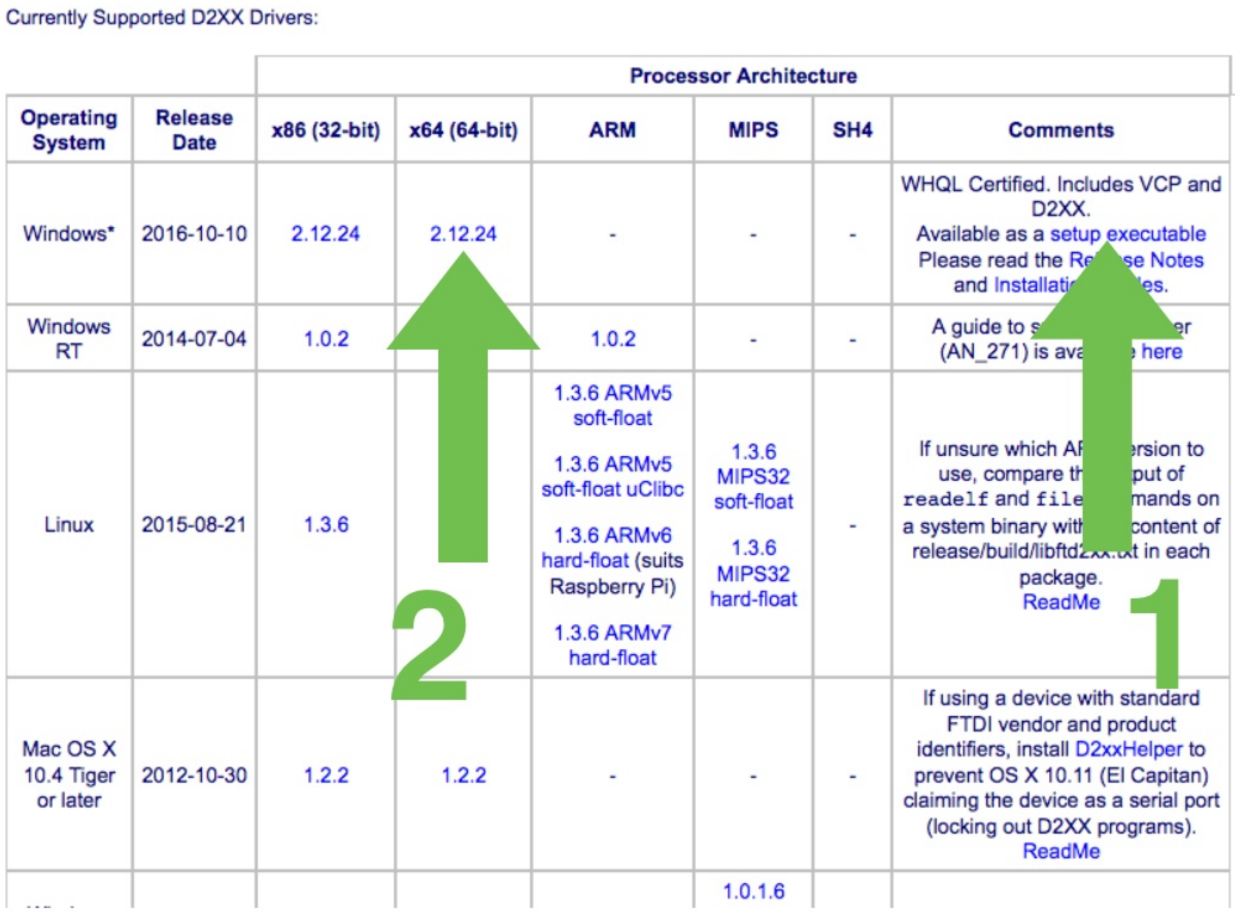
The P-ROC and P3-ROC boards use a chip from a company called "FTDI Chip" to handle the USB communication, so you need to install the FTDI driver so Windows can properly see the device when you plug it in.

You can download the latest version from here:

http://www.ftdichip.com/Drivers/D2XX.htm

Here's a screen shot of the download section of that page. Note that the actual version number of the driver might be newer that the screen shot below. That should be ok.

Download and run the setup executable from the "1" link in the screen shot.



Currently Supported D2XX Drivers:

		Processor Architecture					
Operating System	Release Date	x86 (32-bit)	x64 (64-bit)	ARM	MIPS	SH4	Comments
Windows*	2016-10-10	2.12.24	2.12.24	-	-	-	WHQL Certified. Includes VCP and D2XX. Available as a setup executable Please read the Re se Notes and Installati es.
Windows RT	2014-07-04	1.0.2		1.0.2	-	-	A guide to s er (AN_271) is ava here
Linux	2015-08-21	1.3.6		1.3.6 ARMv5 soft-float 1.3.6 ARMv5 soft-float uClibc 1.3.6 ARMv6 hard-float (suits Raspberry Pi) 1.3.6 ARMv7 hard-float	1.3.6 MIPS32 soft-float 1.3.6 MIPS32 hard-float	-	If unsure which A rsion to use, compare th put of `readelf` and `file` mands on a system binary with content of release/build/libftd2xx.txt in each package. ReadMe
Mac OS X 10.4 Tiger or later	2012-10-30	1.2.2	1.2.2	-	-	-	If using a device with standard FTDI vendor and product identifiers, install D2xxHelper to prevent OS X 10.11 (El Capitan) claiming the device as a serial port (locking out D2XX programs). ReadMe
					1.0.1.6		

2. Now download and unzip the other package

Next you need to download the other package (from the "2" link in the screen shot) which is a zip file.

Unzip it and find the file called `ftd2xx64.dll`. (Probably in the `amd64` folder.)

Copy it to `C:\Windows\System32`

Rename it from `ftd2xx64.dll` to `ftd2xx.dll`

Now MPF will be able to communicate with the P-ROC or P3-ROC.

Continue on with the *How to configure MPF for the P-ROC/P3-ROC platform* documentation to finish your MPF configuration for the P-ROC/P3-ROC.

3. Install Visual C++ Redistributable for Visual Studio 2019

You might already have those but in case you do not install Visual C++ Redistributable for Visual Studio 2019 (64-bit).

What if it did not work?

MPF is erroring out on start-up? Cannot connect to your P/P3-Roc? Have a look at the *troubleshooting guide for P/P3-Roc*.

How to install P-ROC / P3-ROC drivers on Mac OS

Installing the P-ROC drivers (libpinproc and pypinproc) on the Mac is a manual process that requires a few prerequisites and some supporting software. We chose to use the homebrew package manager to help us with the install, which is similar to the apt-get package manager in Linux. The following instructions will help you get homebrew installed, along with everything else.

These instructions assume you have already installed MPF.app. If you haven't, you will need to *go back and do that first*, since it has to be installed before you can build the P-ROC drivers.

1. Install Brew

Open a Terminal and paste in the following commands (and then press <Enter> after each one):

```
cd /usr/local
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

If you're prompted to install Xcode, click Install, followed by Agree.

After Xcode installs (or right away if you already had it), press Return to continue and then enter your password.

You'll see a bunch of stuff scroll by as things are downloaded and installed.

2. Create a folder in your user folder called "proc"

From the same terminal window, run:

```
mkdir ~/proc
```

3. Download osx-proc-support

Change to that new folder:

```
cd ~/proc
```

And run the following command which will clone (download) the files you need to make the P-ROC run on the Mac. (Even though this is called "osx-proc-support", it also works with MacOS Sierra.)

```
git clone https://github.com/missionpinball/osx-proc-support
```

4. Install prerequisites via Brew

Now run:

```
brew install libftdi libusb-compat cmake
```

5. Install yaml-cpp

The P-ROC/P3-ROC requires a library called yaml-cpp. While there is a yaml-cpp package in brew, it's too new to use here. Adding to the fuss is that the version we need is no longer available, so we included it on the osx-proc-support package that you downloaded earlier.

Run the following commands to compile it from scratch:

```
cd ~/proc/osx-proc-support
tar -xzf yaml-cpp-0.2.5.tar.gz
cd yaml-cpp-0.2.5
mkdir bin
cd bin
cmake ..
make
sudo make install
```

6. Download & install libpinproc

Libpinproc is the P-ROC/P3-ROC library that lets the host computer talk to the P-ROC/P3-ROC hardware. Run the following commands:

```
cd ~/proc
git clone --branch=dev https://github.com/missionpinball/libpinproc
```

Copy the Mac version of CMakeLists.txt to the libpinproc folder:

```
cp -r ~/proc/osx-proc-support/CMakeLists.txt ~/proc/libpinproc
```

That avoids having to edit the file manually. It should work for nearly all situations, but if libpinproc won't compile in the next steps, you should make sure the paths in `include_dirs` within CMakeLists.txt are correct.

```
cd libpinproc
mkdir bin
cd bin
cmake -DBUILD_SHARED_LIBS=ON ..
make
sudo make install
```

7. Download & install pypinproc 2.1

Pypinproc is a wrapper library that allows Python apps (like MPF) to talk to the libpinproc that you installed in the previous step. Unfortunately the version that is available from the multimorphic.com website only works with Python 2.x, and MPF uses Python 3.x, so you have to download a version that we modified to work with Python 3:

```
cd ~/proc
git clone https://github.com/missionpinball/pypinproc
cd pypinproc
python3 setup.py build
sudo python3 setup.py install
```

(If you prefer to install pypinproc in a virtualenv, make sure it's activated before this step, and omit sudo from the last line.)

8. Install D2xxHelper

D2xxHelper is provided by FTDI Chips, the maker of the chip which acts as the USB interface on the P-ROC/P3-ROC boards. Mac OS comes with its own FTDI driver that's loaded by default and prevents other FTDI drivers from running. D2xxHelper adjusts the priorities of FTDI driver loading so that the FTDI driver we need loads first, preventing the Apple FTDI driver from loading. This is Apple Support's recommended method of solving the problem, so you're safe. You'll be prompted by Gatekeeper to enter your password to accept installation of the package- this is normal. You'll also be warned that this package may be incompatible with future versions of macOS.

```
cd ~/proc/osx-proc-support
sudo installer -pkg D2xxHelper_v2.0.0.pkg -target /
```

9. Reboot

You have to reboot in order to have the changes D2xxHelper made take effect. After that, you should be all set and can continue on with the *How to configure MPF for the P-ROC/P3-ROC platform* documentation to finish your MPF configuration for the P-ROC/P3-ROC.

What if it did not work?

MPF is erroring out on start-up? Cannot connect to your P/P3-Roc? Have a look at the *troubleshooting guide for P/P3-Roc*.

How to install P-ROC / P3-ROC drivers on Linux

If you want to use MPF on a Debian-based version of Linux (which includes Ubuntu), you can use our all-in-one Debian installer which is detailed in the *Installing MPF on Linux* guide.

Note that when you run that installation script, it will ask you what type of hardware you'll be using. If you choose the "P3 or P-ROC" option, then it will install all of the libraries and drivers you need, and everything should work.

After that, you can continue on with the *How to configure MPF for the P-ROC/P3-ROC platform* documentation to finish your MPF configuration for the P-ROC/P3-ROC.

What if it did not work?

MPF is erroring out on start-up? Cannot connect to your P/P3-Roc? Have a look at the *troubleshooting guide for P/P3-Roc*.

How to configure MPF for the P-ROC/P3-ROC platform

Related Config File Sections
hardware:
p_roc:

Once you have your *P-ROC/P3-ROC drivers installed*, you need to configure your machine to use the P-ROC or P3-ROC.

1. Set your platform

In your machine-wide config file, set the platform.

For the P-ROC:

```
hardware:
  platform: p_roc
```

For the P3-ROC:

```
hardware:
  platform: p3_roc
```

2. Set your driver boards:

Next, configure the driver boards setting which tells MPF which type of driver boards you're using. If you're using the P-ROC driver boards (like the PD-16 or PD-8x8), then you set it like this:

For the P-ROC:

```
hardware:
  platform: p_roc

p_roc:
  driverboards: pdb
```

For the P3-ROC:

```
hardware:
  platform: p3_roc

p_roc:
  driverboards: pdb
```

Note that if you're using a P-ROC with an existing machine, then your driver boards will be either wpc, stern, etc. See the documentation on *configuring MPF for use in existing machines* for details.

3. Configure your watch dog timeout

The P-ROC has the ability to use a *watch dog* timer. This is enabled by default with a timeout of 1 second. If you would like to disable this, or you'd like to change the timeout, you can do so in either the `p_roc:` or `p3_roc:` section of your machine-wide config.

For the P-ROC or P3-Roc:

```
p_roc:
  use_watchdog: true
  watchdog_time: 1s
```

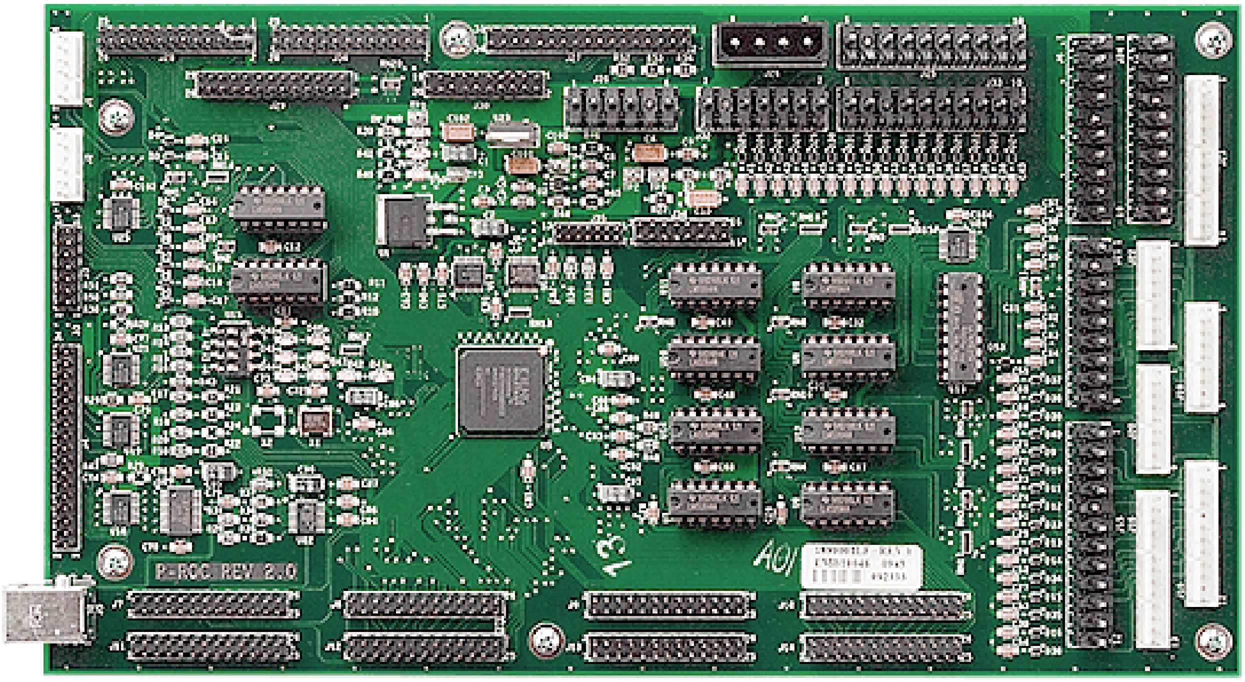
What if it did not work?

Have a look at our *troubleshooting guide for the P/P3-Roc*.

How to configure switches (P-ROC)

Related Config File Sections
switches:

To configure switches on a P-ROC, you can follow the guides and instructions in the *Switches* docs.



However there are a few things to know about using switches with a P-ROC.

number:

Switches are directly connected to the P-ROC board itself. There are two types of switches—matrix and direct—and they each have a different number format.

Note: If you're using your P-ROC in an existing machine, then don't use the number settings here. Instead use the numbers from the *existing machine section* of the documentation.

Direct Switches

The P-ROC has 32 direct switch inputs (which are switches that are directly connected to the P-ROC that do not require a switch matrix). Direct switches are numbered 0-31. (See the P-ROC documentation for the connector mappings.)

Direct switches are configured in your machine config file by starting the number with "SD", like this:

```
switches:
  my_switch:
```

(continues on next page)

(continued from previous page)

```
    number: SD0
  my_other_switch:
    number: SD1
  another_switch:
    number: SD12
```

Matrix Switches

If you're using a switch matrix, then the switch numbers are entered using the column number, then a slash, then the row number.

```
switches:
  my_switch:
    number: 0/0    # column 0, row 0
  my_other_switch:
    number: 0/1    # column 0, row 1
  another_switch:
    number: 3/4    # column 3, row 4
```

Mixing and matching direct and matrix switches

You can mix-and-match direct and matrix switches. However you should be aware of the hardware limitations of combining both. The P-ROC gives you the ability for ONE of the following:

- 32 direct switches and an 8x8 (64 switches) matrix
- 24 direct switches and an 8x16 (128 switches) matrix

Basically the P-ROC has the ability to repurpose 8 of the direct switch inputs as row inputs to extend the switch matrix from 8 to 16 rows. This means that valid values are:

- Direct switches, `SD0` - `SD31`
- Matrix switches, `0/0` - `7/7`

OR

- Direct switches, `SD8` - `SD31`
- Matrix switches, `0/0` - `7/15`

In other words, if any switch uses a row number (the number after the slash) greater than 7, then you can't use direct switches 0 through 7.

The configuration of this is automatic based on the numbers you use, but currently there is no error checking to ensure that SD0 - SD7 are not used if you have any switch which a row that's 8-15.

Choosing direct versus matrix switches

The only difference between direct and matrix switches is in how they're wired. Matrix switches use less wire, but require diodes on the switches and are harder to troubleshoot. Direct switches are easier to wire, but they require more wire and you're limited to 24 (or 32) of them.

If you're using *opto switches* then you must connect the IR receivers to direct switch inputs, since the direct switch inputs are always powered.

There's a misconception that direct switches are "faster" than matrix switches. That is false. The P-ROC scans the 8 columns of the matrix (one at a time), then it reads the direct switches, then the matrix switches again, then the directs, etc. So from a practical sense, the directly switches are really like a single column matrix with either 24 or 32 rows, and they're scanned after the rows of the matrix are scanned. So whether a switch is direct or in the matrix doesn't affect the scanning speed or response time of the switch.

Debounce options

The P-ROC has the ability to configure *debounce settings* for switches. A non-debounced switch which report its state change immediately, while a debounced switch will report its state change after it's been in the new state for two consecutive reads.

By default, MPF will enable debouncing in both directions (open and close) for all switches. However you can override this on a per-switch basis with a switch's `debounce:` setting.

Valid options are `normal`, `quick`, and `auto`.

To disable debouncing for a switch, add `debounce: quick` to the switch config, like this:

```
switches:
  my_switch:
    number: 0/0
    debounce: quick
```

To force debouncing to always be used (which is also the default on the P-ROC, so not really needed), configure it like this:

```
switches:
  my_switch:
    number: 0/0
    debounce: normal
```

What if it did not work?

Have a look at our *troubleshooting guide for the P/P3-Roc*.

How to configure switches (P3-ROC)

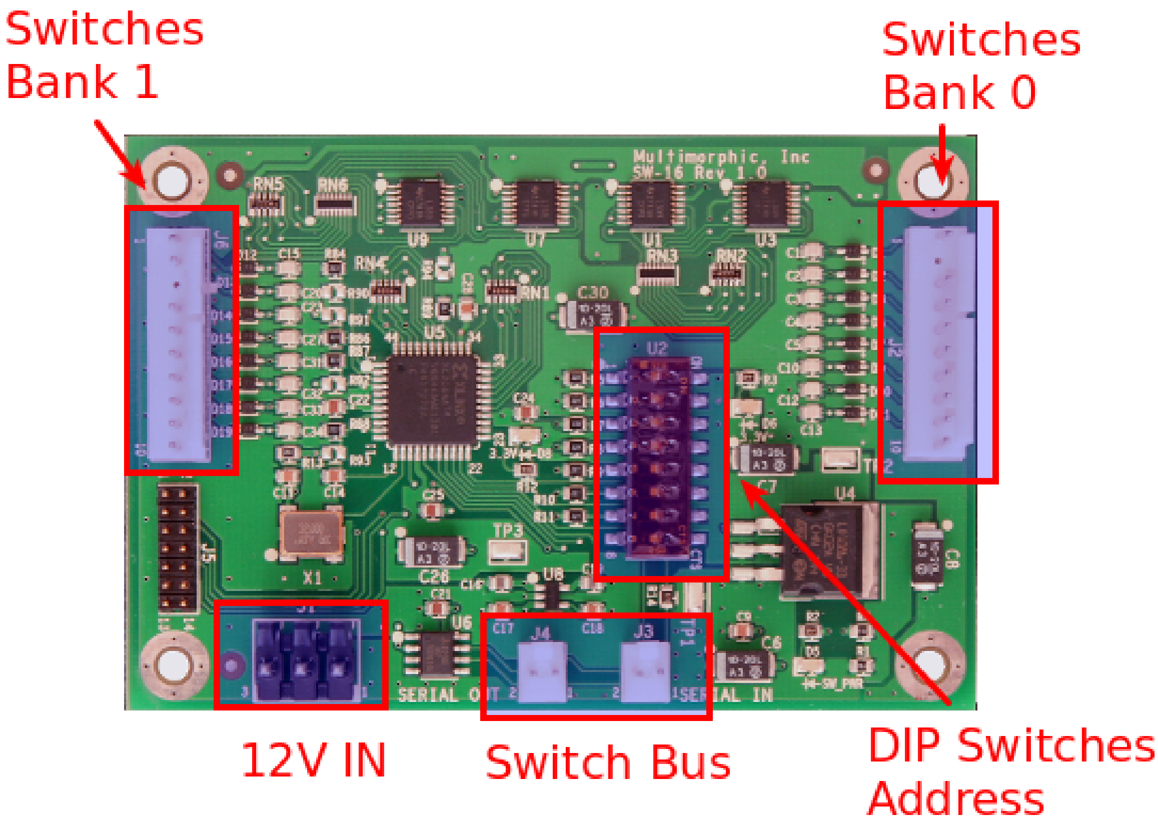
Related Config File Sections
switches:

To configure switches on a P3-ROC, you can follow the guides and instructions in the *Switches* docs.

However there are a few things to know about using switches with a P3-ROC.

number:

Unlike the P-ROC, the P3-ROC does not have switch inputs on the P3-ROC itself. Instead, you add SW-16 boards which each have 16 direct switch inputs. (e.g. there is no switch matrix.) You can connect up to 16 SW-16s to support as many as 256 switches.



Each SW-16 has a unique `board number` which is set using DIP switches (find that out now). On each board there are two banks (A and B) of 8 switches. Then each switch has an `input number` (0 to 7).

To configure the `number:` of a switch connected to an SW-16 board and a P3-ROC, you have two options:

Board Bank Switch Syntax

The first (and easier) option is to enter the number as a combination of the SW-16 `board address` (0-15, as configured by the DIP switches), then the `bank number` (Bank A is 0, Bank B is 1), then the switch `input number` (0-7).

For example:

```
switches:
  my_switch:
    number: A0-B0-0    # SW-16 board at address 0, Bank A, Switch 0
  my_other_switch:
    number: A2-B1-5    # SW-16 board at address 2, Bank B, Switch 5
```

Direct Numbering

You can also use the internal number. As mentioned in the SW-16 manual you can calculate those using:

`Bank A switch: P3-ROC switch # = SW-16 address * 16 + Bank A switch input` `Bank B switch: P3-ROC switch # = SW-16 address * 16 + 8 + Bank B switch input`

However, we recommend the above syntax because it will perform this calcuation for you.

For example:

```
switches:
  my_switch:
    number: 0     # SW-16 board at address 0, Bank A, Switch 0
  my_other_switch:
    number: 45    # SW-16 board at address 2, Bank B, Switch 5
```

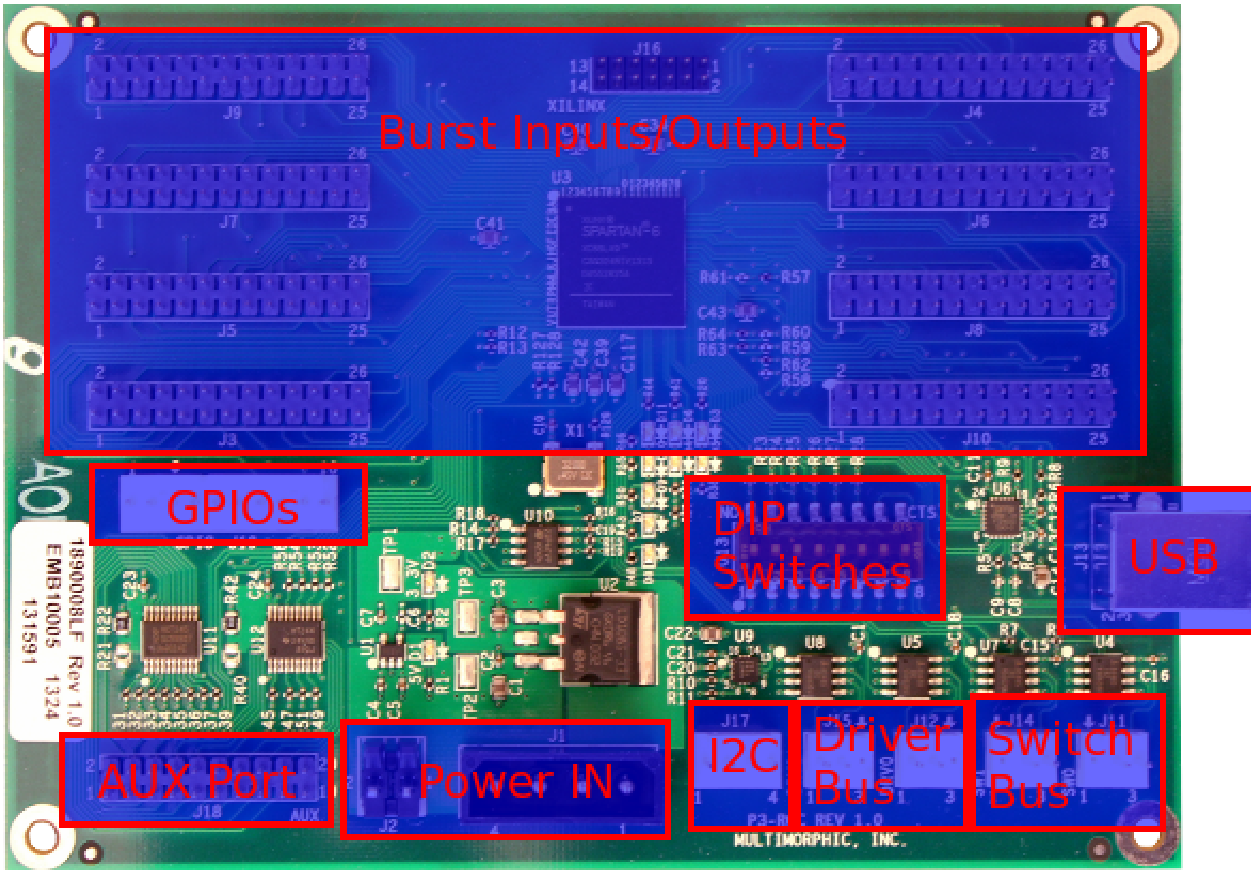
Connecting Switches

Switches are connected between the input pin and ground. On SW-16 revision 1 those are pins 1, 3-9 for switch 0 to 7 on bank A (J2) and 1, 2, 4-9 for switch 0 to 7 on bank B (J6). Ground is on 10 for both banks. On SW-16 revision 2 those are pins 2 to 9 for switch 0 to 7 and 10 for ground on headers J2 (Bank A) and J6 (Bank B). Revision two has an additional low current 12V output on pin 1.

Burst Switch Inputs

Burst switch inputs (J3 to J10) on the P3-Roc can be used in two ways in MPF.

- Burst Optos - As burst switch inputs in combination with a burst switcher driver can drive long-beam optos. This is how they were originally intended.
- Local Inputs - Alternatively you can use them as direct local inputs (and the burst drivers as outputs; see *How to configure coils/drivers/magnets (P-ROC/P3-ROC)* section for details).



Burst Switches as Burst Optos

You can configure burst optos using the following syntax:

```
switches:
  burst_opto_1_20:
    number: burst-1-20      # burst input with switch 1 and driver 20
  burst_opto_2_20:
    number: burst-2-20      # burst input with switch 2 and driver 20
  burst_opto_10_5:
    number: burst-10-5      # burst input with switch 10 and driver 5
```

Make sure to disable DIP switches 1 and 2 on the P3-Roc. You can configure up to five switches per driver.

Note: There is currently no reliable way to read the initial state of burst optos. MPF will assume that all burst IRs are open on start-up. This might cause if you use it to track balls in a ball lock which might contain balls from a previous game. This limitation will probably go away in future firmware update of the P3-Roc.

Burst Opto PCBs

Burst optos are unlike normal *optos* and you cannot use normal optos. Instead, they use 40kHz infrared transmitters and receivers. Multimorphic sells them as single transmitter/receiver (part number: PCBA-0011-0002) and as bank of 8 (part number: PCBA-0003-0003).

Burst Switches as Local Inputs

If you want to use burst switches as local inputs set DIP switch 2 to on on the P3-Roc. You can use those 64 inputs as direct inputs:

```
switches:
  local_switch0:
    number: direct-0       # local input 0
  local_switch20:
    number: direct-20      # local input 20
```

Make sure to assign IDs >= 4 to all SW-16 boards if you set DIP 2. Local switches behave just like any other switch on the P3-Roc. Hardware rules, debouncing etc. will behave exactly the same way.

Note: You need at least Firmware version 2.6 to use burst switches as local inputs on the P3-Roc.

Warning: There is no electronic protection on the P3-Roc for burst switches (neither as local inputs nor as burst optos). Do not use them without additional protection. Any voltage above 3.3V or below 0V will irreversibly damage the P3-Roc. Make sure you know what you are doing before turning this on. We advise to use SW-16 for normal playfield/mech inputs and only use local inputs with additional circuits (not directly). If you plan to use burst optos have a look at the PCBs offered from Multimorphic for that purpose.

Debounce options

The P-ROC has the ability to configure *debounce settings* for switches. A non-debounced switch which report its state change immediately, while a debounced switch will report its state change after it's been in the new state for two consecutive reads.

By default, MPF will enable debouncing in both directions (open and close) for all switches. However you can override this on a per-switch basis with a switch's `debounce:` setting.

Valid options are `normal`, `quick`, and `auto`.

To disable debouncing for a switch, add `debounce: quick` to the switch config, like this:

```
switches:
  my_switch:
    number: A0-B0-0
    debounce: quick
```

To force debouncing to always be used (which is also the default on the P-ROC, so not really needed), configure it like this:

```
switches:
  my_switch:
    number: A0-B0-0
    debounce: normal
```

What if it did not work?

Have a look at our *troubleshooting guide for the P/P3-Roc*.

How to configure coils/drivers/magnets (P-ROC/P3-ROC)

Related Config File Sections
coils:

To configure coils, drivers, motors, and/or magnets (basically anything connected to PD-16 board's driver outputs) with P-ROC/P3-ROC hardware, you can follow the guides and instructions in the *Coils (Solenoids)* docs.

(If you're using a P-ROC with an existing machine's driver board, like a WPC machine, then see the *existing machine documentation*.) If you are using the P-Roc with PDB drivers you can use the local drivers as number 0 to 31.

Warning: Please ensure that you have established *common ground between logic and coil power* before turning on high voltage on your coils (especially on homebrew machines). Ignoring this might lock on your coils, overheat them, burn down your house or kill you. We are serious, floating grounds are dangerous. If you are not an electrical engineer read the *guide about voltages and power*.

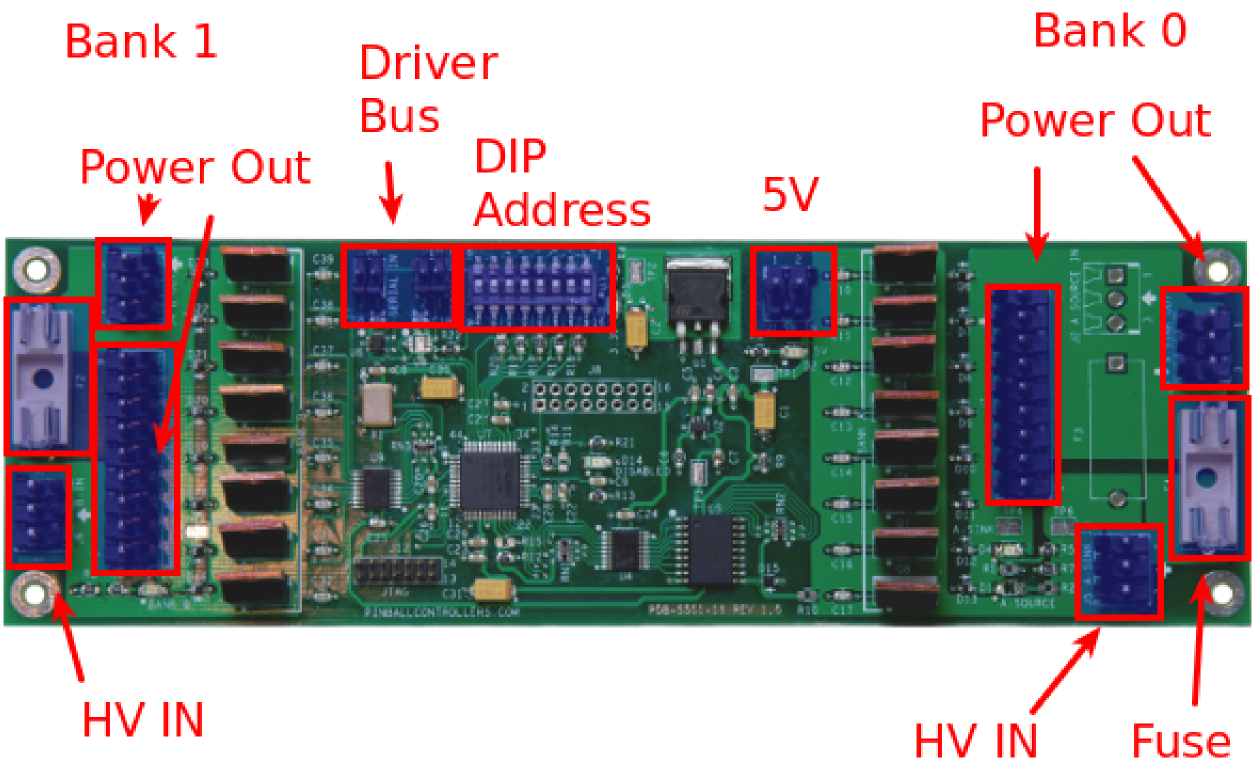
In a nutshell: You need to connect your logic ground (5V/12V) and your high voltage ground (48V or 80V). A *power entry or power filter board* is a convenient solution to solve this (and more) issues.

Always turn all PSUs off when connecting power or you might fry all boards at once. This is generally a good idea but even more important when connecting more than one power supply to a board.

IF YOU DID NOT UNDERSTAND WHAT THIS WARNING MEANS STOP NOW AND TRY TO UNDERSTAND IT. OTHERWISE YOUR HARDWARE WILL LIKELY BURST INTO FLAMES AND YOU NEED TO WAIT A FEW DAYS FOR A REPLACEMENT OR EVEN WORSE IT MIGHT KILL YOU. IGNORING THIS IS THE MOST COMMON CAUSE FOR BROKEN DRIVER BOARDS.

The only specific thing you have to know for this platform is the number format:

number:



For PD-16-based devices, the numbering format is:

```
number: Ax-By-z
```

The "A" and "B" capital letters are required. (A means Address, B means Bank). The lowercase x, y, and z letters should be replaced with numbers to represent the following on a PD-16 driver board:

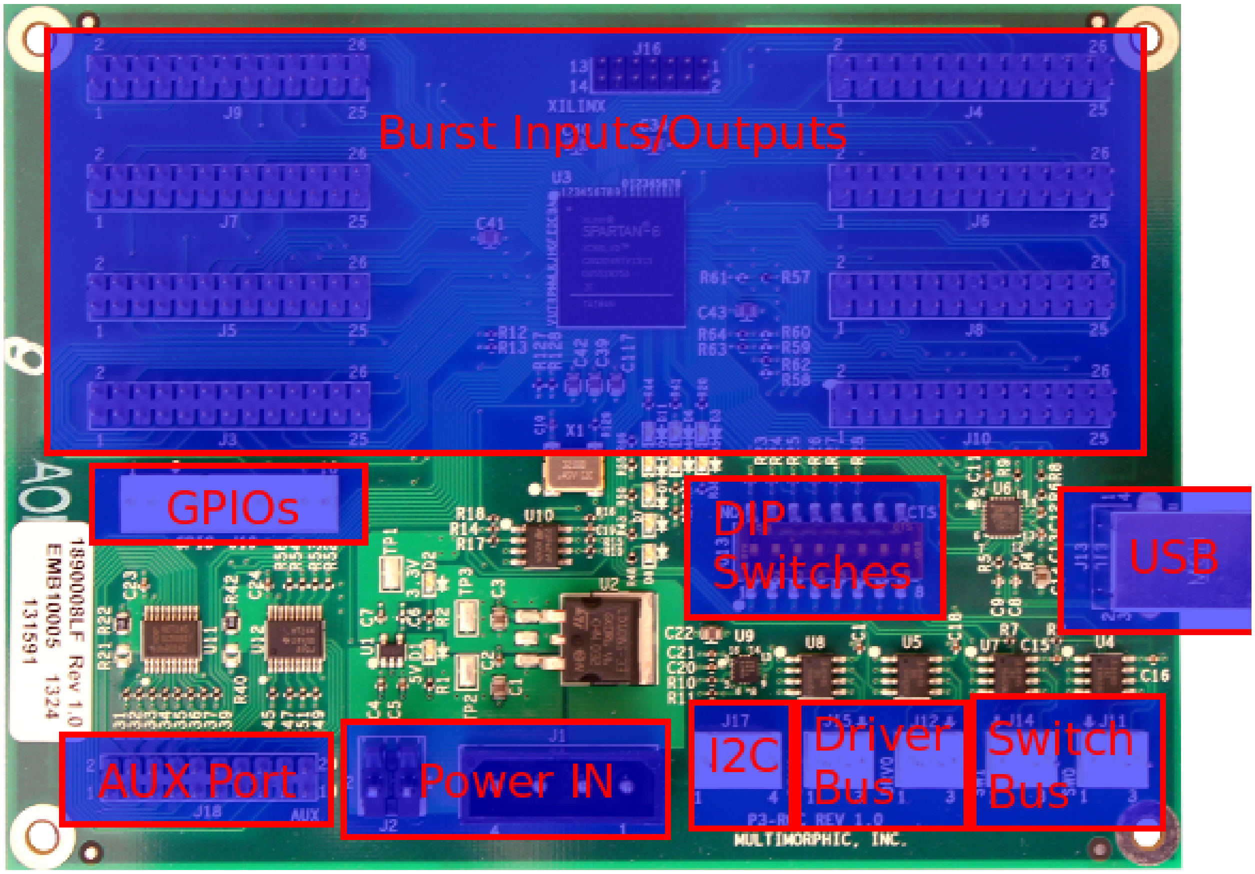
- x : Board address (0-31)
- y : Bank address (0 for A, 1 for B)
- z : Output number (0-7)

Note: The output number is the logical number, *not* the pin number. For example, Output 0 is on Pin 1, and there is a key pin at 2 or 3. Check the manual for the exact mapping.

For example:

```
coils:
  some_coil:
    number: A0-B1-6
    default_pulse_ms: 30
```

Burst Switches as Local Outputs (P3-Roc only)



If you want to use burst switches as local outputs set DIP switch 1 to on on the P3-Roc. You can use those 64 output as direct outputs:

```
coils:
  local_output0:
    number: direct-0    # direct driver 0
  local_output20:
    number: direct-20   # direct driver 20
```

Make sure to assign IDs >= 2 to all PD-16 boards if you set DIP 1 (MPF cannot check this for you). Local outputs behave just like any other output on the P3-Roc. Hardware rules, pulse, hold, pwm etc. will behave exactly the same way.

You may also use outputs as `digital_outputs`. For instance, to control a motor driver circuit:

```
digital_outputs:
  motor_left:
    number: direct-5
    type: driver
  motor_rigth:
    number: direct-6
    type: driver
```

Note: You need at least Firmware version 2.6 to use burst switches as local outputs on the P3-Roc.

Warning: There is no electronic protection on the P3-Roc for burst switches (neither as local outputs nor as burst optos). Additionally, there are no drivers attached to the outputs and they cannot drive any pinball mechs. Make sure not to draw too much current out of those outputs. Also, any voltage above 3.3V or below 0V will irrevisibly damage the P3-Roc. Make sure you know what you are doing before turning this on. We advise to use PD-16 for normal playfield/mech drivers and only use local outputs with additional circuits (not directly).

Pulse time

The P-Roc, P3-Roc and/or PD-16 have the ability to specify the "pulse time". Pulse time is the coil's initial kick time. For example, consider the following configuration:

```
coils:
  some_coil:
    number:
    default_pulse_ms: 30
```

When MPF sends this coil a pulse command, the coil will be fired for 30ms.

Pulse Power

You can also set the power of pulses on your coil:

```
coils:
  some_coil:
    number:
    default_pulse_ms: 30
    default_pulse_power: 0.5
```

See the hold power section below for internal details about PWM times. With the P-Roc and P3-Roc it is not possible to use `default_hold_power` and `default_pulse_power` at the same time.

Hold Power

If you want to hold a driver on at less than full power, MPF does this by using `default_hold_power` parameter which works for all platforms. It can range from 0.0 to 1.0 and defines the time share the coil is on (0%-100%).

The P-Roc internally uses two parameters which determine how many milliseconds the coil will be on (pwm-on time) and off (pwm-off time). MPF will calculate those based on your power settings.

```
coils:
  some_coil:
    number:
    default_pulse_ms: 32
    default_hold_power: 0.5
```

When enabled, this driver will be pulsed for 32ms and then hold on at 50% duty which will convert to 1ms on, 1ms off, 1ms on, 1ms off and so on.

With the P-Roc it is not possible to use `default_hold_power` and `default_pulse_power` at the same time.

Recycle

You can set *recycle time* to your coil to prevent it from overheating by repeated pulses. The recycle time is not configurable on the P-Roc but you can turn it on or off (default on). Default recycle time (called reload in the P/P3-Roc) is `64ms`.

This is an example:

```
coils:
  some_coil_with_recycle:
    number:
    default_pulse_ms: 32
    default_recycle: true
  some_coil_without_recycle:
    number:
    default_pulse_ms: 32
    default_recycle: false
```

What if it did not work?

Have a look at our *troubleshooting guide for the P/P3-Roc*.

Related How To guides

- *Coil Resistance and Hardware Details*
- *Wiring Dual Wound Coils*
- *Dual-Wound versus Single-Wound coils*
- *Adjust coil hold power*
- *Adjust coil strength (pulse times)*
- *Recycle / "Cool Down" Time*
- *Details About Flippers*
- *How to configure single-wound flippers*
- *How to configure dual-wound flippers*
- *Flipper end-of-stroke (EOS) switches*

How to configure LEDs on the PD-LED (P-ROC/P3-ROC)

Related Config File Sections
lights:

This guide explains how to configure MPF to use LEDs attached to a Multimorphic PD-LED board with either a P-ROC or P3-ROC.

Note that if you're using a P-ROC/P3-ROC and you want to use serial-controlled LEDs (NeoPixels, etc.), then you can do that with a P-ROC/P3-ROC by using a *FadeCandy* instead of a PD-LED. You can also mix-and-match PD-LEDs and FadeCandy LEDs. If you are using a PD-8x8 or a local matrix on the P-Roc see the instructions about *Matrix lights for P/P3-Roc*.

Channel and Number Syntax

In MPF *lights* abstract a light source which emits arbitrary colors. However, this is not true for all real lights. Some support only white (GIs), others only a single-color (i.e. red inserts) and others support full RGB. For that reason MPF knows *light numbers and channel numbers*. Internally, a light consists of one or multiple channels. For instance, a single-color GI will contain a single white channel. While a RGB light will control a red, green and a blue channel. A white light behind a red insert should be a single red channel (because it cannot emit other colors through the red insert). You can configure those channels using the `channels` setting or use `start_channel` and `type` to define the channels. See *Lights* for details.

However, in most cases a platform supports one type of lights (per `subtype`) this would be overly verbose and we added the `number` setting for configuring lights in the common platform way. For instance a platform for GIs will configure single channel white lights or a serial LED controller will configure RGB lights with three channels.

The PD-LED assumes that you want to use RGB LEDs by default. For anything else you have to use channels.

Light Numbers

PD-LED numbers use the format: `board_number-led_index1-led_index2-led_index3`

Since the PD-LED board directly drives single color LED outputs, when you use it with RGB LEDs, you combine three outputs into a single RGB LED. The PD-LED supports both common cathode (common ground) and common anode (common 3.3v) LEDs, so each LED you buy has four pins (red, green, blue, and common). When you configure the hardware number for a PD-LED RGB LED, you specify four parts, separated by dashes:

1. The address of the PD-LED board on the serial chain (as configured via the DIP switches on the PD-LED.
2. The output number of the red element.
3. The output number of the green element.
4. The output number of the blue element.

You separate those with dashes, so an example PD-LED configuration might look like this:

```
lights:
  l_led0:
    number: 8-0-1-2
    subtype: led
```

The example above configures "l_led0" as the LED connected to PD-LED board at address 8, using outputs 0, 1, and 2 as its red, green, and blue connections.

`subtype: led` is only needed on the P-Roc since `subtype` defaults to `led` on the P3-Roc defaults. The P-Roc defaults to `matrix`.

Channels

Channels use the format: `board_number-led_index`

This is almost the same as above but it addresses only one output (instead of three). You can use the channel syntax as for *l_led0* above:

```
lights:
  l_led0:
    channels:
      red:
        - number: 8-0
      green:
        - number: 8-1
      blue:
        - number: 8-2
```

You might connect different color channels to your PD-LED. For instance you might have only a red channel:

```
lights:
  my_red_only_insert:
    channels:
      red:
        - number: 8-0       # board 8 and first channel
```

Or you can configure a white LED:

```
lights:
  my_white_light:
    channels:
      white:
        - number: 8-4
```

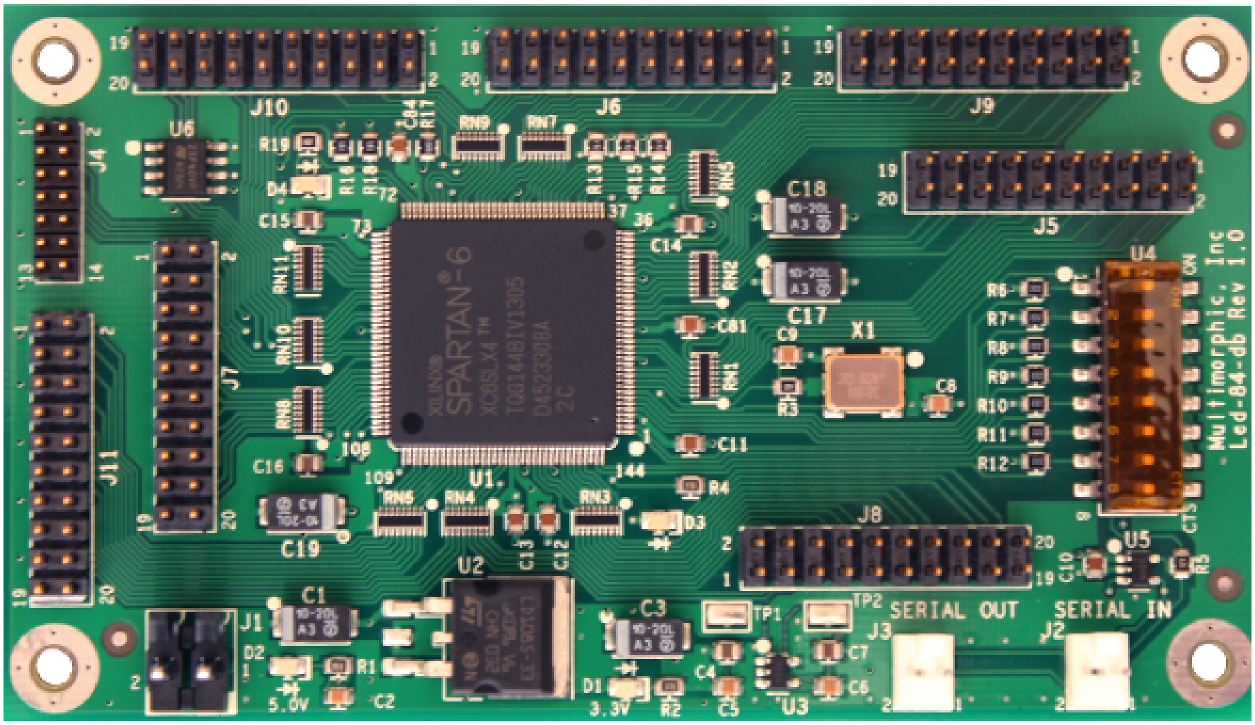
Starting from MPF 0.54 you can also have MPF calculate the numbers for you:

```
lights:
  led_0:
    start_channel: 8-0
    subtype: led
    type: rgb    # will use red: 8-0, green: 8-1, blue: 8-2
  led_1:
    previous: led_0
    subtype: led
    type: rgbw   # will use red: 8-3, green: 8-4, blue: 8-5, white: 8-6
  led_2:
    previous: led_1
    subtype: led
    type: rgbw   # will use red: 8-7, green: 8-8, blue: 8-9, white: 8-10
```

You can also configure two red channel, green plus white or any other combination. See *LEDs* for more details about how to configure channels for different types of LEDs.

Understanding the PD-LED board

The PD-LED controls up to 84 individual LED elements, which can be used to control individual single color LEDs, or (more likely), combined into groups to control RGB LEDs.



The PD-LED uses a "direct/parallel" connection method for LEDs, where each LED has connections for each color element running back to the PD-LED. This requires at least two wires per LED (or four for RGB LEDs). In addition you can also use serial LEDs starting with PD-LED v2 (see below).

Parallel LEDs

Those LEDs are wired individually to the PD-LED.

This is an example:

```
lights:
  l_led_1:
    number: 4-0-1-2
    subtype: led
```

LED number:

You can use number 0 to 83 to address your LEDs. The number format is defined above.

Polarity

The PD-LED allows you to use either common anode or common cathode LEDs. (See the PD-LED documentation for details. The type of LED would dictate whether you hook it up between the PD-LED's output and ground, or between the output and 3.3v.) You can then use the config file to specify which type of LED you have, such as:

```
lights:
  l_shoot_again:
    number: 8-60-61-62
    platform_settings:
      polarity: true
```

True = common cathode (or common ground), **False** = common anode (or common 3.3V)

Note that DIP Switch 6 on the PD-LED board controls whether the "default" state of the LEDs after a reset is high or low. Basically it's whether all the LEDs turn on or turn off when the board is reset. Which position does what is dependent on whether you're controlling the anode or the cathode with your outputs, so basically if you turn on your PD-LED and all your LEDs turn on, then flip DIP switch 6 on the PD-LED to the opposite position and power cycle the board. Note: If servos are connected to a PD-LED board, DIP switch 6 also effects servo signal on power up. See *Servos on a PD-LED (P-ROC/P3-ROC)* for additional information.

Breakout boards for parallel LEDs

You likely want to buy or build some breakout boards for your LEDs when you are using parallel LEDs in your machine. Otherwise, you might end up in wiring hell for your lights. Luckily, there breakout boards exist which connect via a ribbon cable to your PD-LED.

Breakout boards:

- Four LEDs breakout (Multimorphic) - PCBA-0025-0002
- Five equally spaced LEDs + three LEDs breakout (Multimorphic) - PCBA-0030-0001
- Breakout wire harness (PBL) for four LEDs - #600-0274-00

Part numbers of lights and flashers:

- GI RGB LED (PBL) - #600-0230-00
- RGB Insert LED (PBL) - #600-0220-01
- RGB Insert LED (Multimorphic) - PCBA-0004-0001
- Flasher (Multimorphic) - PCBA-0024-0001
- Pop bumber RGB LED (PBL) - #600-0258-00

Additionally, they got a PCB with five equally spaced LEDs which breaks out another three LEDs (part number:). Make sure to check those out because it will make your live easier. In your final machine you will probably build some larger PCBs and connect them using ribbon cables.

Serial LEDs on the PD-LED

Overview video about serial LEDs:

https://youtu.be/Q9BG9T7Kj4A

Starting with PD-LED v2 you can use the PD-LED to drive serial LEDs. To enable a serial LEDs you need to configure your PD-LED board in your `p_roc` section. Assuming your PD-LED has the address 4 you can use the following config to enable all serial LEDs and and define a few:

```
p_roc:
  pd_led_boards:
    4:
      use_lpd880x_0: true
      use_lpd880x_1: true
      use_lpd880x_2: true
      use_ws281x_0: true
      use_ws281x_1: true
      use_ws281x_2: true
lights:
  l_serial_chain_0_first:
    start_channel: 0-100
    type: rgb
    subtype: led
  l_serial_chain_0_second:
    previous: l_serial_chain_0_first
    type: rgb
    subtype: led
  l_serial_chain_1_first:
    start_channel: 4-250
    type: rgb
    subtype: led
  l_serial_chain_2_first:
    start_channel: 4-400
    type: rgb
    subtype: led
```

LED number:

By default MPF maps the first chain (of both LPD880x and WS281x) to LEDs 100 to 249. The second chain to 250 to 399 and the third to 400 to 599. You can change those settings in the *pd_led_boards:* section.

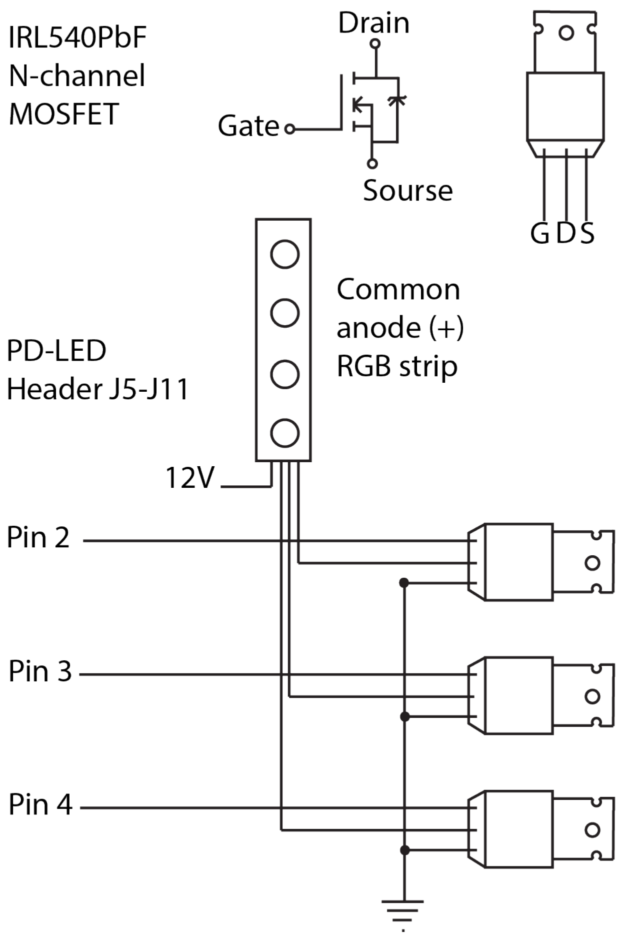
The number format is the same as for parallel LEDs (see above). Board number is the number the at the PD-LED's DIP switches. Index is the number of your LED (starting at 0) in the chains plus the chain start offset (100 for the first chain, 250 for the second or 400 for the third).

Color Correction

If you are using RGB LEDs, they might not be perfectly white when you turn them on. They might be pinkish or blueish instead depending on the brand of the LED. To a certain extend this is normal/expected and you can compensate for it by configuring *color_correction profiles in light_settings*.

Amplifying PD-LED channels with FETs

PD-LED drives LEDs with a current of 22mA. Also, it cannot exceed its output voltage of 3.3 V effectively limmiting it to a single LED per channel. If you want to drive more LEDs on a channel (e.g. GIs or long strips) you can connect a MOSFET (as stated in the manual). Choose a logic-level N-Channel MOSFET with an Output Characteristics curve showing current saturation meeting the needs of the strip with a voltage between the gate and source (VGS) of 3.3 V or less. This is an example of such a circuit:



Please make sure to connect your PD-LED and the FET to the same *common ground* or your FET will smoke when connecting power.

What if it did not work?

Have a look at our *troubleshooting guide for the P/P3-Roc*.

How to configure Matrix Lights (P-ROC/P3-ROC)

Related Config File Sections
lights:
p_roc:

To configure matrix lights connected to a PD-8x8 and a P-ROC or P3-ROC, you can follow the guides and instructions in the *Lights* docs. If you are using PD-LED with see the instructions about *LEDs on PD-LED for P/P3-Roc*.

However there are a few things to know about using matrix lights with a P3-ROC.

Note: If you're using your P-ROC in an existing machine, then don't use the number settings here. Instead use the numbers from the *existing machine section* of the documentation.

---

number:

Configure the number for each lamp in your *lights:* section with an entry that contains a bunch of letters and numbers which specify the specific columns and row outputs that make up each lamp. It's probably easiest to look at an example.

```
lights:
  some_light:
    subtype: matrix
    number: C-A2-B0-0:R-A2-B1-0
```

Notice there are two parts to the number, separated by a colon.

The first part is the column information:

- `C` means "Column"
- `A2` means "the PD-8x8 at Address 2"
- `B0` means "Bank 0"
- `0` means output "0"

The second part is the row information:

- `R` means "Row"
- `A2` means "the PD-8x8 at Address 2"
- `B1` means "Bank 1"
- `0` means input "0"

Luckily this is only something you have to work out once. :)

You only need `subtype: matrix` on the P3-Roc since `subtype` defaults to `led`. The P-Roc defaults to `matrix` so you may omit it there.

Fine tuning column strobe times

The lamp matrix works by quickly cycling through the columns and then activating the rows for the individual lamps that are supposed to be on in that specific column.

Back in the day when only incandescent bulbs were used, this pretty much worked the same everywhere and you didn't have to worry about any other settings. However now that it's possible to use LEDs replacement bulbs in your lamp matrices, and there are all sorts of LEDs like "anti-ghosting" and things like that, you may want to fine-tune the timing of how the columns are activated. You can do that in the `p_roc:` section of your machine-wide config.

For P-ROC:

```
p_roc:
  lamp_matrix_strobe_time: 100ms
```

For P3-ROC:

```
p_roc:
  lamp_matrix_strobe_time: 100ms
```

100ms is the default setting (which is used if you don't add this entry), but you can play with this value to see how it affects your lights or LEDs.

This is a system-wide setting, so if you have multiple lamp matrices on multiple PD-8x8 boards, then this setting will be used for all of them.

What if it did not work?

Have a look at our *troubleshooting guide for the P/P3-Roc*.

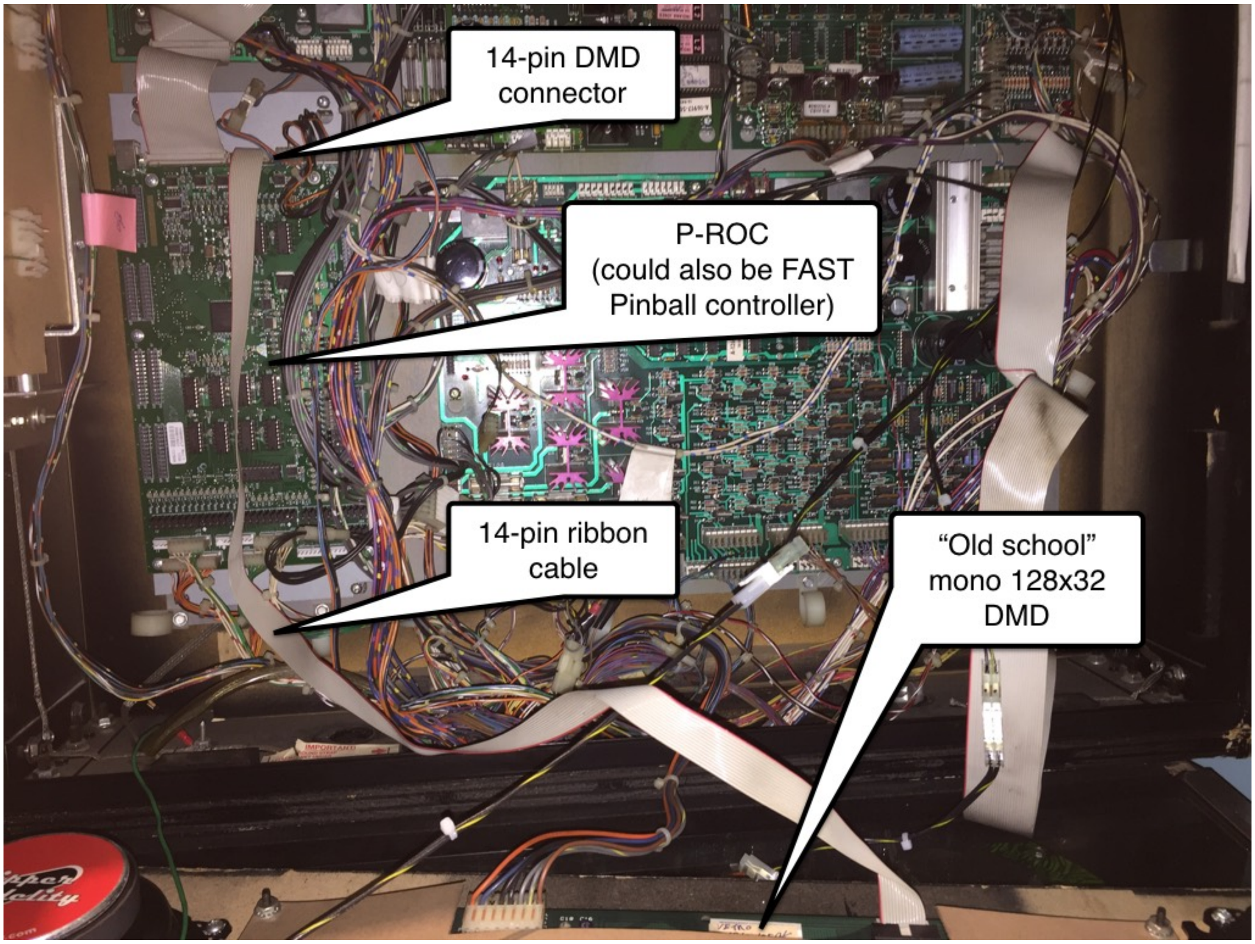
How to configure mono/traditional DMD (P-ROC)

Related Config File Sections
dmds:
p_roc:

The P-ROC can drive a traditional single-color pinball DMD via the 14-pin DMD connector cable that's been in most pinball machines for the past 25 years, like this:

Note: If you want to drive an RGB LED DMD and you're using a P-ROC, you can do that by adding a *SmartMatrix* or *RGB.DMD* board which you would then use in place of the P-ROC's 14-pin DMD connector.

1. Connect your hardware



2. Add a physical DMD device entry

Once you have your hardware and port set, you need to create the actual device entry for the DMD.

You do this in the `dmds:` section of the machine config. This section is like the other common sections (switches, coils, etc.) where you enter the name(s) of your device(s), and then under each one, you enter its settings.

(And yes, in case you're wondering, it's possible to have more than one physical DMD.)

To do this, create a section in your machine-wide config called `dmds:`, and then pick a name for the DMD, like this:

```
dmds:
  my_dmd:
    shades: 16
```

You need to have at least one setting for this to be a valid YAML file, so we usually just pick the shades and add that with a value of 16 (which means the DMD runs will convert the display content to 16 mono shades when it displays it).

The "shades" option is how many brightness shades you want. 1990s WPC machines supported 4 shades, and modern Stern DMD machines support 16. The P-ROC supports 16 shades (even on older 1990s plasma DMDs). Most modern games will probably be 16 shades, but you can do 4 (or even 2) if you want an old school look.

There are lots more options for the physical_dmd: section than just the "shades" option listed here. Check the *dmds:* for a list of all the options.

Note that one option you do NOT have for physical DMDs is the color. That's because the color of the DMD is determined by the DMD itself. You don't actually send it color values, rather, you just send it brightness levels, and the DMD shows those brightness levels with whatever color the DMD is.

3. Set a source display

Now that you have everything configured, the last step is to make sure the DMD knows what content to show. In MPF, you do this by mapping a physical DMD to an *MPF display*.

By default, the DMD will look for a display (in your *displays:* section called "dmd". However you can override this and configure the DMD to use whatever logical display you want by setting a `source_display:` setting. (Just make sure that the width and height of your source display match the physical pixel dimensions of the DMD or else it will be weird.)

4. Setting the DMD update rate

By default, MPF will send new DMD frames to the P-ROC at about 30 frames per second. (Technically it sends a new frame every 33ms.)

5. Fine tuning the DMD timing cycles

The P-ROC is able to drive a traditional DMD with 16 shades of intensity, ranging from off (0) to full on (15). Note that the P-ROC doesn't control (or even know) what color the DMD is as that's dictated by the DMD itself.

The P-ROC creates the appearance of 16 levels of brightness by rapidly turning individual dots on and off.

For years, DMD's have been high-voltage gas plasma displays, though more recently they're LED-based (even the single color ones with the 14-pin connectors).

Some people have reported less-than-optimal quality when using a P-ROC with certain types of DMDs. To address this, the P-ROC allows you to fine-tune the timings of the individual bit planes that make up the image.

For details on this, you can search the P-ROC forums (now defunct) for "high_cycles" to find a few threads where people are talking about these settings. Then you can set them in the `p_roc: dmd_timing_cycles:` section of your machine-wide config, like this:

```
p_roc:
  dmd_timing_cycles: 90, 190, 50, 377
```

Note that we do not have specific recommendations for values here and based on our experience, we haven't found a need to change this. However, if you do have issues and you get new values by talking to the P-ROC folks, this is how you adjust them in MPF.

Our recommendation is that you leave the `dmd_timing_cycles:` setting out of your `p_roc:` config unless you need it and really know what you're doing. (There's potential that bad values here could permanently damage your DMD hardware, so again, only change these if you know what you're doing.)

A final config you can test

At this point you're all set, and whatever slides and widgets are shown on the DMD's source display in MPF-MC should be shown on the physical DMD.

That said, all these options can be kind of confusing, so we created a quick example config you can use to make sure you have yours set right. (You can actually just save this config to `config.yaml` in a blank machine folder and run it to see it in action which will verify that you've got everything working properly.)

To run this sample config, you can either run `mpf both`.

When you run it, do not use the `-x` or `-X` options, because either of those will tell MPF to not use physical hardware which means it won't try to connect to the Teensy.

Note that the *Using a traditional (single color) physical DMD* guide has more details on the window and slide settings used in this machine config.

```
hardware:
  platform: p_roc
p_roc:
  driverboards: pdb
displays:
  window:  # on screen window
    width: 600
    height: 200
  dmd:  # source display for the DMD
    width: 128
    height: 32
    default: true
window:
  width: 600
  height: 200
  title: Mission Pinball Framework
  source_display: window
dmds:
  my_dmd:
    brightness: 1.0
slides:
  window_slide_1:  # slide we'll show in the on-screen window
    - type: display
      effects:
        - type: dmd
          dot_color: ff5500
      width: 512
      height: 128
    - type: text
```

(continues on next page)

(continued from previous page)

```
      text: MISSION PINBALL FRAMEWORK
      anchor_y: top
      y: top-3
      font_size: 30
    - type: rectangle
      width: 514
      height: 130
      color: 444444
  dmd_slide_1:  # slide we'll show on the physical DMD
    - type: text
      text: IT WORKS!
      font_size: 25
slide_player:
  init_done:
    window_slide_1:
      target: window
    dmd_slide_1:
      target: dmd
```

What if it did not work?

Have a look at our *troubleshooting guide for the P/P3-Roc*.

How to configure an RGB DMD (P-ROC/P3_ROC)

Related Config File Sections
rgb_dmds:
smartmatrix:

Neither the P-ROC nor the P3-ROC has direct support for RGB DMDs. However you can still use an RGB DMD with a P-ROC/P3-ROC by using one of the standalone RGB DMD controllers. (Basically you buy the RGB DMD hardware and another small controller, and then you have two USB connections from your computer—one to the P-ROC/P3-ROC, and a second to the RGB DMD controller.)

Standalone RGB DMD options which you can use with a P-ROC/P3-ROC include:

- *SmartMatrix*
- *RGB.DMD*

How to configure alpha-numeric displays (P-ROC)

Related Config File Sections
segment_displays:
p_roc:

The P-ROC includes support four alpha-numeric displays (0-3). You can configure them in MPF:

```
segment_displays:
  display1:
    number: 0
  display2:
    number: 1
  display3:
    number: 2
  display4:
    number: 3
```

Note that the *Alpha-Numeric / Segment Displays* guide has more details on using alpha numeric and segment displays.

Video about segment displays:

https://youtu.be/Jyf3jxGXnTw

What if it did not work?

Have a look at our *troubleshooting guide for the P/P3-Roc*.

How to configure the accelerometer (P3-ROC)

Related Config File Sections
accelerometers:

THe P3-ROC includes an accelerometer which you can use with MPF to detect g-force changes (to use as a tilt) as well as 3-axis leveling (to use to determine whether the machine is level).

To use the accelerometer on the P3-ROC, add it to your machine-wide config file like this:

```
accelerometers:
  p3_roc_accelerometer:
    number: 1
```

The name (which is "p3_roc_accelerometer" in the example above) doesn't really matter.

Other than that, use it like you would any accelerometer in MPF, by following the docs and guides in the *Accelerometers* section of the documentation.

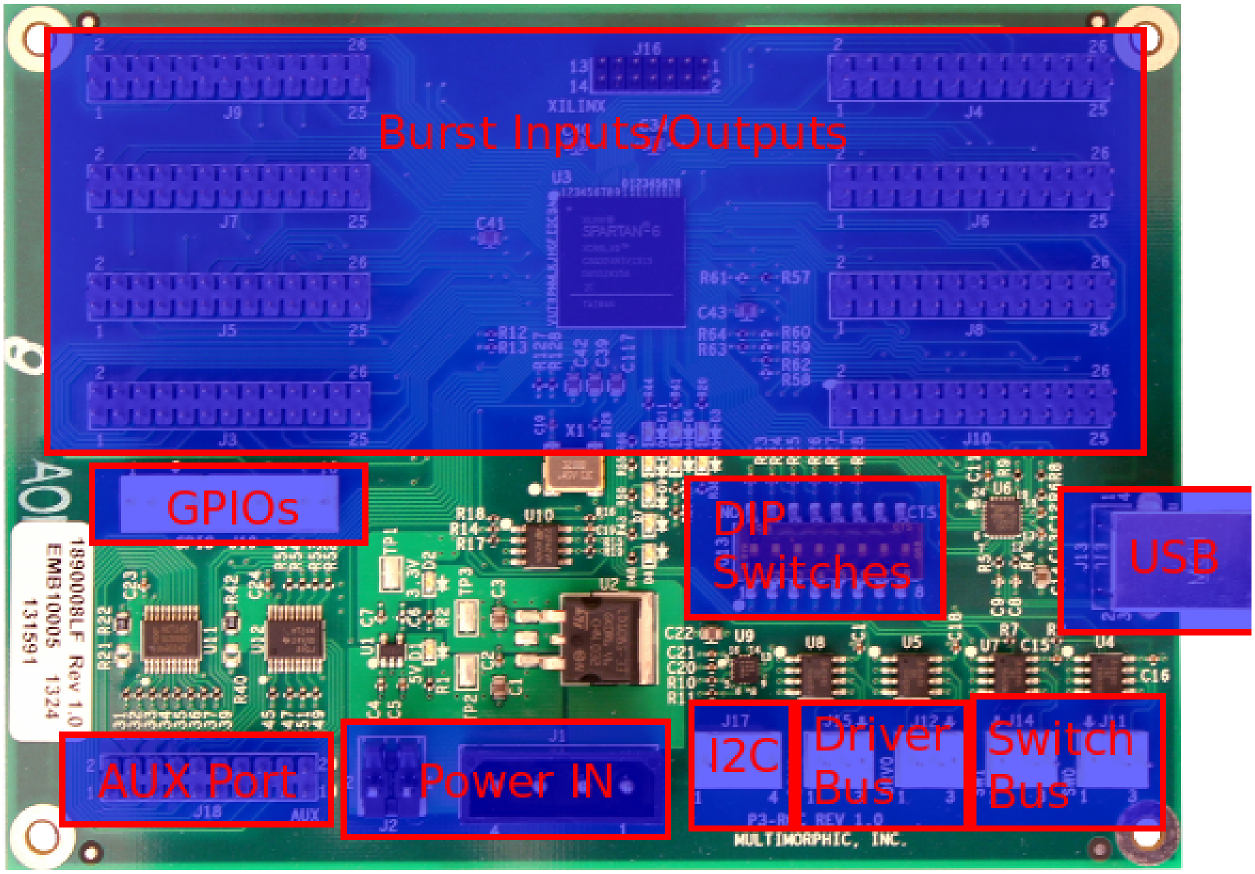
What if it did not work?

Have a look at our *troubleshooting guide for the P/P3-Roc*.

How to use I2C on the P3-ROC

Related Config File Sections
hardware:

The P3-ROC contains an I2C port (J17) which is accessible to MPF. You can use this port to control any I2C-based device.



You need to connect SDA, SCL and ground. You may not need the 3.3V from the P3-ROC as your controller might be a different voltage (which you can then get directly from your power supply), but again that depends on the board.

I2C Servo Controller

For instance you can connect a *servo controller via I2c*. You can't plug the servo directly into the P3-ROC, rather, you can buy an I2C-based servo controller and plug it into the P3-ROC. However, a better option would be to use a *servo on a PD-LED*.

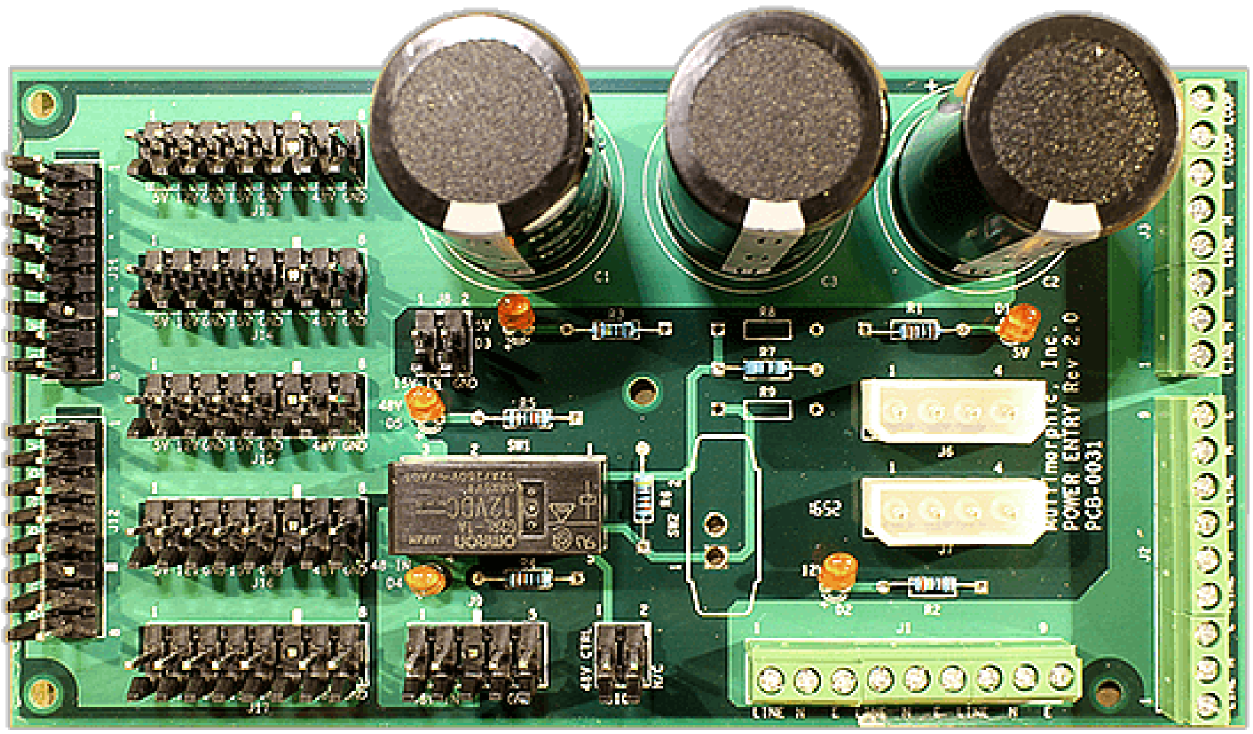
See *I2C Platforms in MPF* for other I2C hardware in MPF.

What if it did not work?

Have a look at our *troubleshooting guide for the P/P3-Roc*.

Power Entry Board

This board can be used to fan out your power rails. See *Voltages and Power in Pinball Machines* for details.



The Multimorphic Power Filter board serves four purposes. 1. It serves as a central connection point for 230V/110V AC and all your PSUs using connectors J1, J2 or J3. 2. It provides a bank of capacitors to buffer current surges on the high voltage rail. 3. It provides safety relay control of the high voltage rail. 4. It connects the ground (negative) terminals of each of the power supplies to prevent a differential in ground levels between the coil supply and the logic supply which is a common cause of unstable operation.

The Multimorphic Power Entry Board allows connections for up to four DC rails:

- 5V
- 12V
- 15V
- High Voltage (HV)

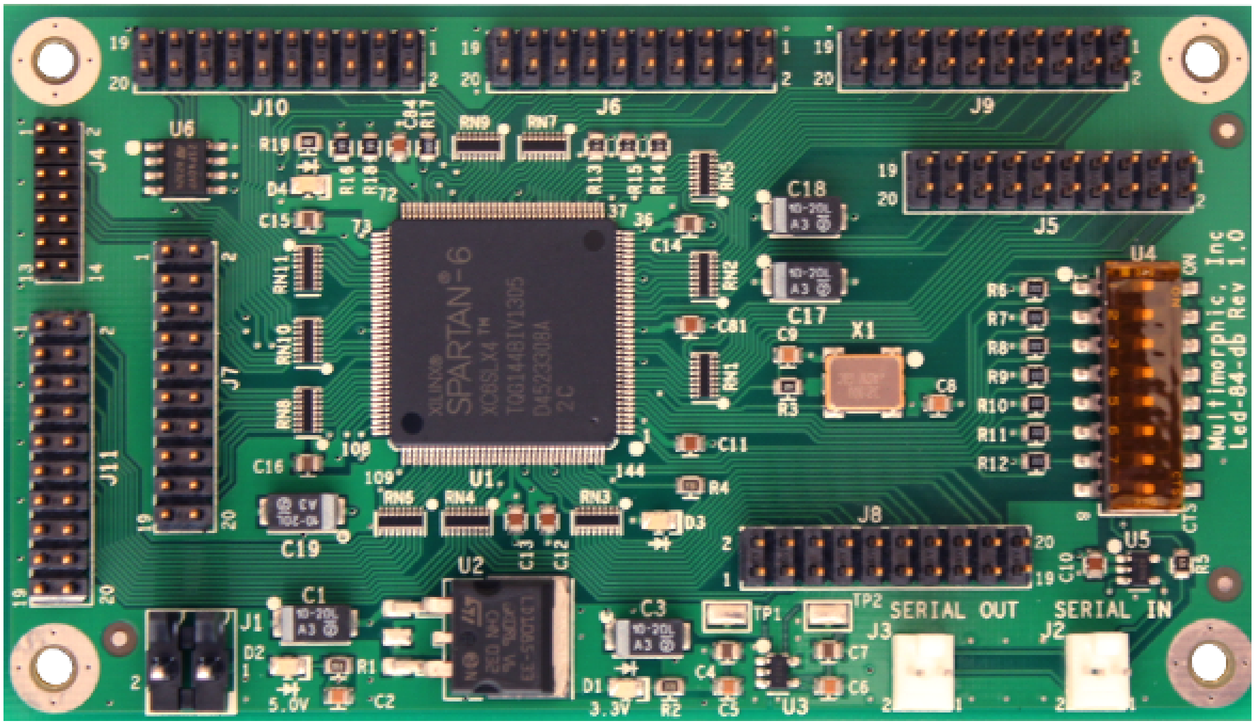
You might use different voltages but the LEDs might operate outside the spec in this case (consult the manual).

DC Outputs are J11 to J17. DC Inputs are J6 to J9. J10 will turn on HV on the ouput. You should connect it to your door switch to cut high power when the door is opened. Make sure that J10 is closed during development or HV will be off.

Servos on a PD-LED (P-ROC/P3-ROC)

Related Config File Sections
servos:
pd_led_boards:

Starting with PD-LED v3 you can configure up to twelve steppers on a PD-LED.



To enable servos you need to configure your PD-LED board in your p_roc section. Assuming your PD-LED has the ID 4 you can use the following config to enable all servos and and define two of them:

```
p_roc:
  pd_led_boards:
    4:
      max_servo_value: 300 # rougly maps to 2ms.
      use_servo_0: true
      use_servo_1: true
      use_servo_2: true
      use_servo_3: true
      use_servo_4: true
      use_servo_5: true
      use_servo_6: true
      use_servo_7: true
      use_servo_8: true
```

(continues on next page)

(continued from previous page)

```
      use_servo_9: true
      use_servo_10: true
      use_servo_11: true
servos:
  servos_4_0:
    number: 4-0
  servos_4_1:
    number: 4-1
```

The number of your servos has to be `id_of_your_ped_led-number`. In this case `4-0` and `4-1` for the first and second servo on PD-LED 4. You will not be able to use LED 72 to LED 83 on the PD-LED when enabling all servos.

max_servo_value determines the width of the pulses sent to the servo. This value can be altered to increase of decrease the servo arc within the physical limits of the device. Higher values widen pulsewidth increasing the range of motion.

DIP switch 6 of the PD-LED controls the default state of the LED outputs when the board first receives power. Because servos receive signal from LED outputs, placing this DIP switch in the on position can activate a servo prior to the PD-LED receiving instructions from the controller and MPF. This in turn may lead to a servo thermal overload state and failure. When using servos, DIP switch 6 should be maintained in the OFF position.

You should hook up your servos to an external power source (usually 5V) and not draw that power from the PD-LED. However, make sure to connect the ground of your power supply. See *Voltages and Power* for details.

Overview video about *servos*:

https://youtu.be/wA6KEODwQ5w

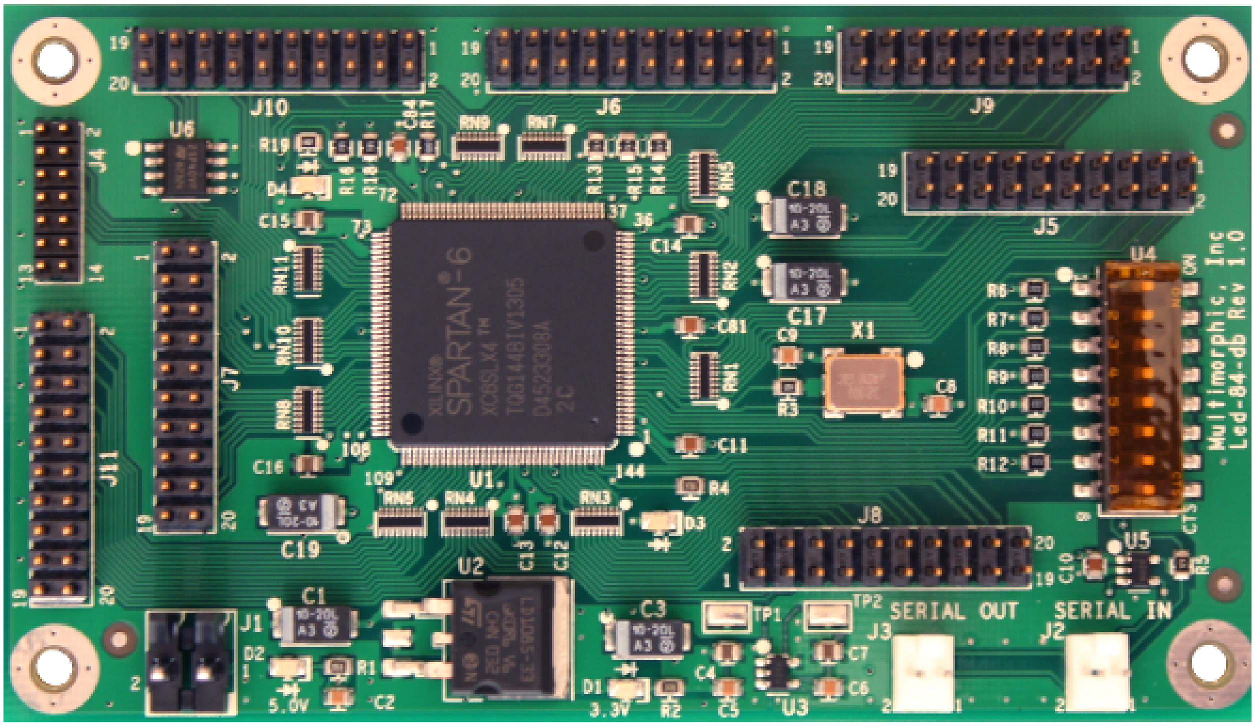
What if it did not work?

Have a look at our *troubleshooting guide for the P/P3-Roc*.

Steppers on a PD-LED (P-ROC/P3-ROC)

Related Config File Sections
steppers:
pd_led_boards:

Starting with PD-LED v3 you can configure up to two steppers on a PD-LED. You need an additional cheap external stepper driver to drive the load of the stepper. Those are sold for a few bucks as StepStick or DRV8825 on amazon, ebay, aliexpress or similar platforms.



To enable steppers you need to configure your PD-LED board in your p_roc section. Assuming your PD-LED has the ID 4 you can use the following config to enable and define two steppers:

```
p_roc:
  pd_led_boards:
    4:
      use_stepper_0: true
      use_stepper_1: true
      # stepper_speed: 13524    # uncomment to tune the speed
switches:
  s_stepper_4_0_home:
    number: A4-B0-0
  s_stepper_4_1_home:
    number: A4-B0-1
steppers:
  stepper_4_0:
    number: 4-0
    homing_mode: switch
    homing_switch: s_stepper_4_0_home
  stepper_4_1:
    number: 4-1
    homing_mode: switch
    homing_switch: s_stepper_4_1_home
```

The number of your stepper has to be id_of_your_ped_led-number. In this case 4-0 and 4-1 for the first and second stepper on PD-LED 4. Every stepper needs a homing switch so MPF can home it at startup. You will not be able to use LED 75 to LED 80 on the PD-LED when enabling both steppers.

You might have to fine-tune the `stepper_speed` setting to your steppers. Increasing the value will reduce the speed of your steppers.

You should hook up your steppers to an external power source and not draw that power from the PD-LED. However, make sure to connect the ground of your power supply. See *Voltages and Power* for details. Connect those stepper drivers as described in *How to use Step Stick Steppers in MPF* (but use the PD-LED outputs).

Overview video about *steppers*:

https://youtu.be/YaRNBU0OHGc

What if it did not work?

Have a look at our *troubleshooting guide for the P/P3-Roc*.

How to update the Firmware of the P-Roc or P3-Roc

If you experience problems around hardware rules or such consider upgrading your P/P3-Roc firmware. Sometimes bugs in the firmware get fixed or stuff becomes more robust. For some known cases MPF will crash intentionally and tell you to upgrade but there might be cases which we do not know.

Finding out the current firmware version

You can find out your current firmware version using `mpf hardware scan`:

```
$ mpf hardware scan

Firmware Version: 2 Firmware Revision: 6 Hardware Board ID: 1
SW-16 boards found:
 - Board: 0 Switches: 16 Device Type: A3 Board ID: 0
 - Board: 1 Switches: 16 Device Type: A3 Board ID: 1
 - Board: 2 Switches: 16 Device Type: A4 Board ID: 2
```

In this example the P3-Roc is running firmware `2.6`.

Upgrading the firmware of the P-Roc or P3-Roc

Warning: DO NOT POWER DOWN THE P/P3-ROC OR YOUR PC DURING THIS PROCESS!

1. Log on to your account on Multimorphic.com (or create one) and go to the Firmware page on the Multimorphic Wiki.
2. Read the Multimorphic upgrade instructions (they know their boards better than we do)
3. Download the firmware for your board (either P-Roc or P3-Roc)
4. Get the Upgrade tool

- On Windows: Download the `pinprocfw.exe` from the Multimorphic site as well
- On Linux: Change to `mpf-debian-installer/libpinproc/bin`
- On Mac: Change to `libpinproc/bin` (likely in `~/proc/libpinproc/bin` if you followed the installer)

5. Run the upgrade tool: `./pinprocfw path/to/the/firmware/file`

What if it did not work?

In case you got troubles with the upgrade we recommend you to contact the Multimorphic support team. If you got a problem with MPF have a look at the *Troubleshooting P-Roc/P3-Roc* section.

Troubleshooting P-Roc/P3-Roc

If you got problems with your hardware platform we first recommend to read our *troubleshooting guide*. Here are some hardware platform specific steps:

P/P3-Roc Does Not Show Up In Device Manager or dmesg Log

If your P/P3-Roc does not show up in device manager (Windows) or does not create a line in `dmesg` or `lsusb` (Linux/Mac) have a look at the USB cable and connection. Bad cables are a thing (especially for longer cables). Try removing USB hubs.

Is the board powered up? Are the four blue LEDs circling? If not check your power supply.

ImportError in MPF Log

If you see something along this in your log:

```
in <module>
   from mpf.platform.pinproc.x86.python36 import pinproc

ImportError: DLL load failed: The specified module could not be found
```

This usually means that the FTDI libs are not installed in the correct version. On Linux pinproc might not be installed at all. On Windows `Visual C++ Redistributable for Visual Studio` might also be missing in the correct version. Have a look a the *install instructions for your OS* to find and install the correct requirements.

Failed to reset P/P3-Roc

If you see this repeatedly in the log of MPF:

```
Failed to reset P/P3-Roc: OSError: Error in WriteData: wrote 0 of 8 bytes. Is your P/P3-Roc connected␣
↪and powered up?
Will retry creating PinPROC and resetting it in 1s.
```

This usually means that the P/P3-Roc is either not powered up or not connected.

Random Crashes of MPF

You might see errors such as the following (usually in p_roc_common.py):

```
OSError: Error in WriteData: wrote 0 of 8 bytes
```

This error is triggered by communication issues with the P/P3-Roc. Often this is caused by an unreliable power supply or overload on the 5V rail of that supply. This might also be caused by a bad USB cable. In any case you should also find USB communication errors in your operating system which might give you further clues.

Run Hardware Scan

Using mpf hardware scan you can find out if your P/P3-Roc is talking properly to MPF using USB. Additionally, it will show you which SW-16 are connected:

```
$ mpf hardware scan

Firmware Version: 2 Firmware Revision: 6 Hardware Board ID: 1
SW-16 boards found:
 - Board: 0 Switches: 16 Device Type: A3 Board ID: 0
 - Board: 1 Switches: 16 Device Type: A3 Board ID: 1
 - Board: 2 Switches: 16 Device Type: A4 Board ID: 2
```

Unfortunately, MPF cannot know which PD-16 or PD-LED are connected as this information is not available. See *mpf hardware (command-line utility)* for details.

Enable Debugging

If you got problems with your platform try to enable debug first. As described in the *general debugging section* of our *troubleshooting guide* this is done by adding debug: true to your p_roc config section:

```
p_roc:
  debug: true
```

This will add a lot more debugging and might slow down MPF a bit. We recommend to disable/remove it after finishing debugging.

Enable Bus Tracing

If your hardware behaves different from the way you told it to in MPF or if you are seeing lags or delays it might be wise to turn on bus tracing.

```
p_roc:
  debug: true
  trace_bus: true
```

This logs all calls to libpinproc. This will cause a lot of additional log lines and might considerably slow down MPF. Definitely disable this after you finished debugging.

Upgrade the Firmware of Your P/P3-Roc

If you experience problems around hardware rules or such consider upgrading your P/P3-Roc firmware. Sometimes bugs in the firmware get fixed or stuff becomes more robust. For some known cases MPF will crash intentionally and tell you to upgrade but there might be cases which we do not know.

See *How to update the Firmware of the P-Roc or P3-Roc* for details about firmware upgrades.

Switches Are Not Registering

If your coils are working but switches are not registering please check the following points:

On the P-Roc

- Is 12V power available? This will disable switches but not much else.
- Is ground connected properly to your switches?

On the P3-Roc

- Do your SW-16 show in *mpf hardware scan*? (see above)
- If not: Is the SW-16 bus connected properly (and not twisted)?
- Is ground connected properly to your switches? Should be connected to pin 10 on J2 or J6 of SW-16.

Some Drivers Are Not Working

If some drivers are working but other are not.

On the P-Roc

If you see the following message on your console (not log; you might have to use the `-t` commandline flag to see them):

```
Refusing to update driver #144; polarity differs on non-custom machine.
```

This means that the polarity which is defined for your machine type (i.e. WPC) does not match your driver definition. If you see this please tell us in the MPF user forum and we will investigate this with you.

All Coils Turn On When I Power Up My Machine

If this happens and MPF is not yet running you likely do not have common ground between high voltage and logic power. Turn your machine off and only turn it back on when you have fixed and verified common ground. Read the section about *common ground* for details or consult an electrical engineer.

If this happens shortly after MPF started and you are using a P-Roc this might have to do with the polarity of your coils. Check the `polarity` setting and make sure you configured the correct machine type as there are different defaults in different machine types.

In any case we recommend that you test this with either less voltage (i.e. 12V instead of 48V) or by using lamps instead of coils on your outputs as that will prevent hardware damage due to overcurrent.

Serial Bus Issues

Bad Cables/Interference

Each serial bus connector has a + and a - pin. The serial cables connect from board to board like jumper-cables + to + and - to -. Connecting ground pins on the serial bus is not required. A bad serial cable can be difficult to diagnose, particularly if it is the first serial cable in a chain as it will prevent signals to all boards downstream of the bad connection. One clue that a bad serial cable is present is if some of the boards function but others do not. Another clue which is sometimes present on the driver bus is discovered looking at the driver boards watchdog timer indicator. On the PD-LED the watchdog is indicated by a lit diode D3. On the PD-16 it is diode D11. The watchdog turns off when the board is receiving signal over the driver bus from the P3-ROC when MPF is running (including attract mode). If wiggling serial cables causes the watch dog to light, a loos connection or bad cable is present. While the switches bus does not have an equivelent watchdog, the game's switch status screen can be monitored while wiggling cables looking for a loose connection. It is possible for the vibration of a mechanism (notoriously from a pop bumper) to cause intermittent faults in a poorly connecting serial cable. Such intermittent faults are difficult to diagnose.

Termination

The P3-ROC interfaces to the playfield through two serial busses. The switches serial bus connects SW-16 boards through J11 and/or J14. The driver serial bus connects PD-16 and PD-LED boards through J12 and J15. The serial busses are designed to allow boards to be connected in a daisy chain fashion to each plug. A sourse of unreliable communication on the buses is improper termination. The last board on each chain (not to be confused with the board with the highest address) should have dipswitch 8 set to ON. For example is the switches serial bus has 6 boards with J11 connecting to board A B and C and J14 connecting to boards D, E and F, dipswitch 8 should be set to ON on boards C and F and set to OFF on all other SW 16 boards. (Terminating board B would prevent communication from board C on that side of the chain.) The same termination strategy also applies to driver boards. For example if a mix of PD-LED and PD-16 boards connect through J15 as A, B, C, D, and E with E being the last board, board E would have dipswitch 8 set to ON.

Additionally, the P3-ROC board itself also has termination dip switches (7 and 8) for the switches serial bus plugs. These should be set to ON. There are no termination dip switches for the driver bus on the P3-ROC board.

Correct Addressing

Each of the SW-16 boards requires a unique binary address which is set by the board's dipswitches 1 through 6. Although the P3-ROC has two serial switch connectors (J11 and J14) there is only one serial switch bus. Meaning, if one SW-16 board connects to the P3-ROC through J11 and another through J14 the SW-16 boards will still require separate addresses to be properly registered.

Similarly, the PD-16 and PD-LED driver boards also each require an unique address on the driver bus accessed through J12 and J15 on the P3-ROC. If for instance a PD-16 and a PD-LED share on the same address, commands through the driver serial bus meant to drive LEDs can acutate coils even if the boards are interfacing through different plugs.

On the SW-16, PD-16 and PD-LED boards themselves dipswitch addressing is somewhat counterintuitive. Switch one is the lowest address bit and on the SW-16 switch 6 is the highest. Reading the switch block from left (starting at switch 1) to right, binary address zero would be 000000, address one through four would be 100000, 010000, 110000 and 001000, respectively. The PD-LED sets addresses on dipswitches 1 through 5 and the PD-16 uses dipswitches 1 through 4 giving these boards fewer address possibilities than the SW-16 which uses switches 1 through 6.

Coils Are Not Firing

What to do if your coils are not working?

Check if Your Hardware is Working at all

Sounds stupid but this is a good start: Is the hardware working at all? Do you see switch hits in the logs? If not, check our section `Your hardware is not working at all`.

Check the Watchdog

If switches (or other features of the platform) are working but coils are not we have to dig deeper. Most hardware platforms have some kind of watchdog. Often there is some LED which indicates if the watchdog is received. The MPF log might also contain clues (especially if you have enabled `debug` and run MPF with verbose flags `-v -V`). If the watchdog is not received by your platform it will not enable coils.

In most cases watchdog related problems indicate wiring problems. Check if your boards are properly wired.

Test Your Coil Numbers using MPF Service CLI

Hardware is connected and generally working, watchdog is good but still your coils are not working? Maybe something with the numbering is odd. Lets tests that using the *MPF Service CLI*. Alternatively, you can also use *service mode* if you have already configured it. Both ways work similarly.

To use service cli:

1. Open two consoles
2. Start your game (e.g. using `mpf both`)
3. Start the service cli from within your game folder using `mpf service`.
4. Type `list_coils` and press `ENTER` to see a list of coils.
5. Type `coil_pulse your_coil` and press `ENTER` to pulse it.

Does it work? If not check the log and try verify the coil number. If you do not specify `default_pulse_ms` MPF will use `10ms` which might not be enough for some mechs. Try to increase that gently (maybe `20ms` or `30ms`).

Reducing light update rate

If you got a lot of lights you might run into bus contention issues. You can reduce the light update rate in MPF:

```
mpf:
  default_light_hw_update_hz: 30   # defaults to 50
```

If you set this too low fades will be less smooth but otherwise it should not affect your game.

Your hardware is not working at all

If your hardware is not working at all make sure that you removed the options `-X`, `-x` and `--vpx` from your `mpf both` or `mpf game` command line. Those options will overwrite the settings in your `hardware` section and MPF will not even try to connect to your hardware. If you got config errors we suggest you add `-X` to figure things out without interfacing real hardware all the time. Just keep that option in mind.

Another stupid thing to check: Is your hardware connected to your PC? We know it is stupid but a loose USB connector has happened to most of us.

On Linux you might want to run the command `lsusb` which should show both of your micro controllers connected. You should see two lines similar to

```
Bus 002 Device 014: ID 0483:5740 STMicroelectronics Virtual COM Port
Bus 002 Device 015: ID 0483:5740 STMicroelectronics Virtual COM Port
```

If you are unsure about the output, run the command once with your controllers connected and once without. If there is no difference, then for sure the USB device is not properly connected.

Add debugging to related devices

If you got problems with some switches also add `debug: true` to those as it will give to more insights into the intentions of those devices. Same will work for flippers, coils, lights, servos, steppers and more. See *general debugging section* for details.

Run MPF with verbose flag

See *general debugging section* for details. TLDR: run `mpf both -t -v -V`.

Report Your Issue and Ask For Help

If you cannot find the issue yourself please prepare some information about your issue according to our *troubleshooting guide* and ask in our forum.

Consider Improving the Documentation

Did you solve your issue but found that some relevant information in the documentation is missing or should be linked/located elsewhere? Either tell us in the forum or consider *improving the* *documentation* yourself to save future users some troubles the same way others saved you some troubles by writing this documentation.

How to configure MPF for FAST Pinball hardware

Here's a list of all the How To guides which explain how to use MPF with FAST Pinball hardware. These guides include the numbering format (how you map specific entries in your config files to board and connector locations) as well as overall settings that affect how your FAST hardware performs. (Watch dogs, update speeds, etc.)

Video about FAST:

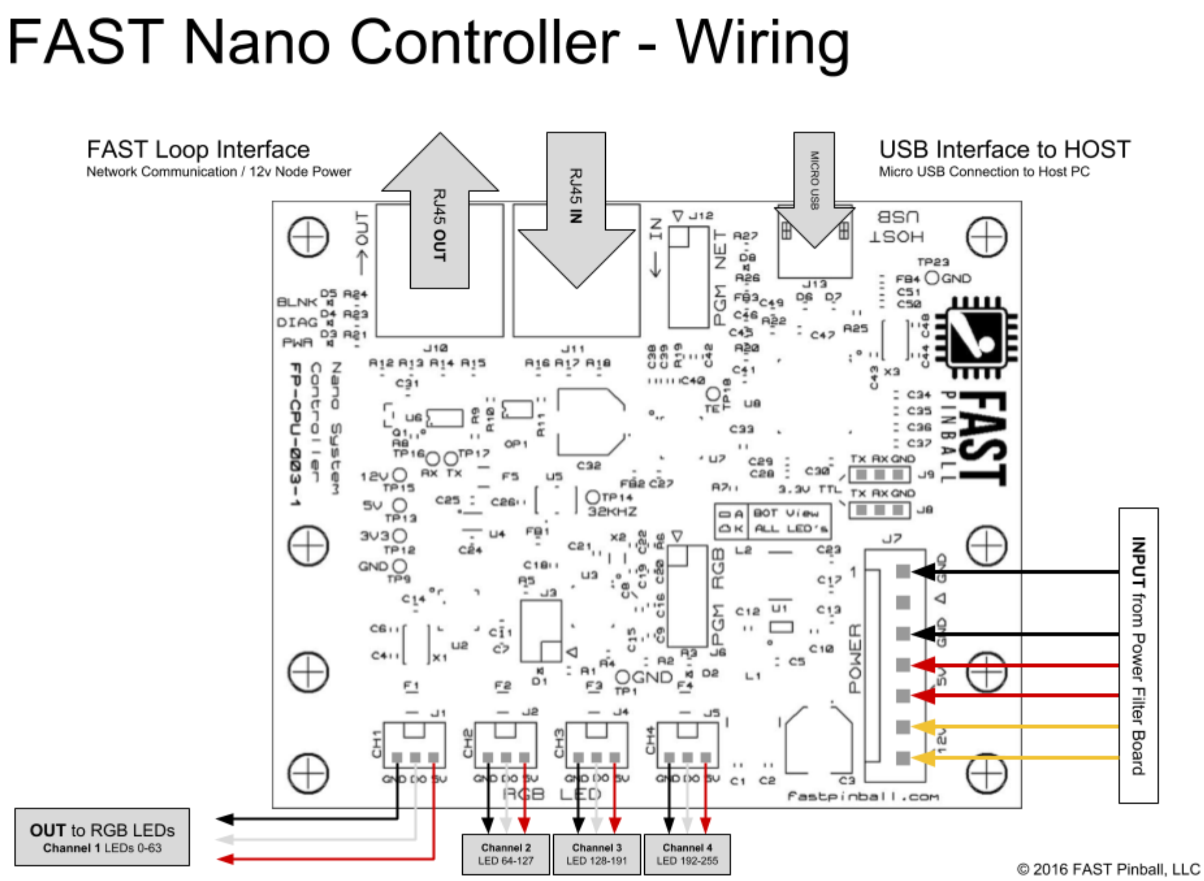
https://youtu.be/uS_dNWOL8mw

Connecting FAST to your Computer

This page is about connecting the FAST system to your computer. It roughly covers connecting the bus between the nodes. For electronic details see the FAST section in the pinballmakers.com Wiki.

FAST Nano

Connect your FAST NANO controller to your PC using USB.



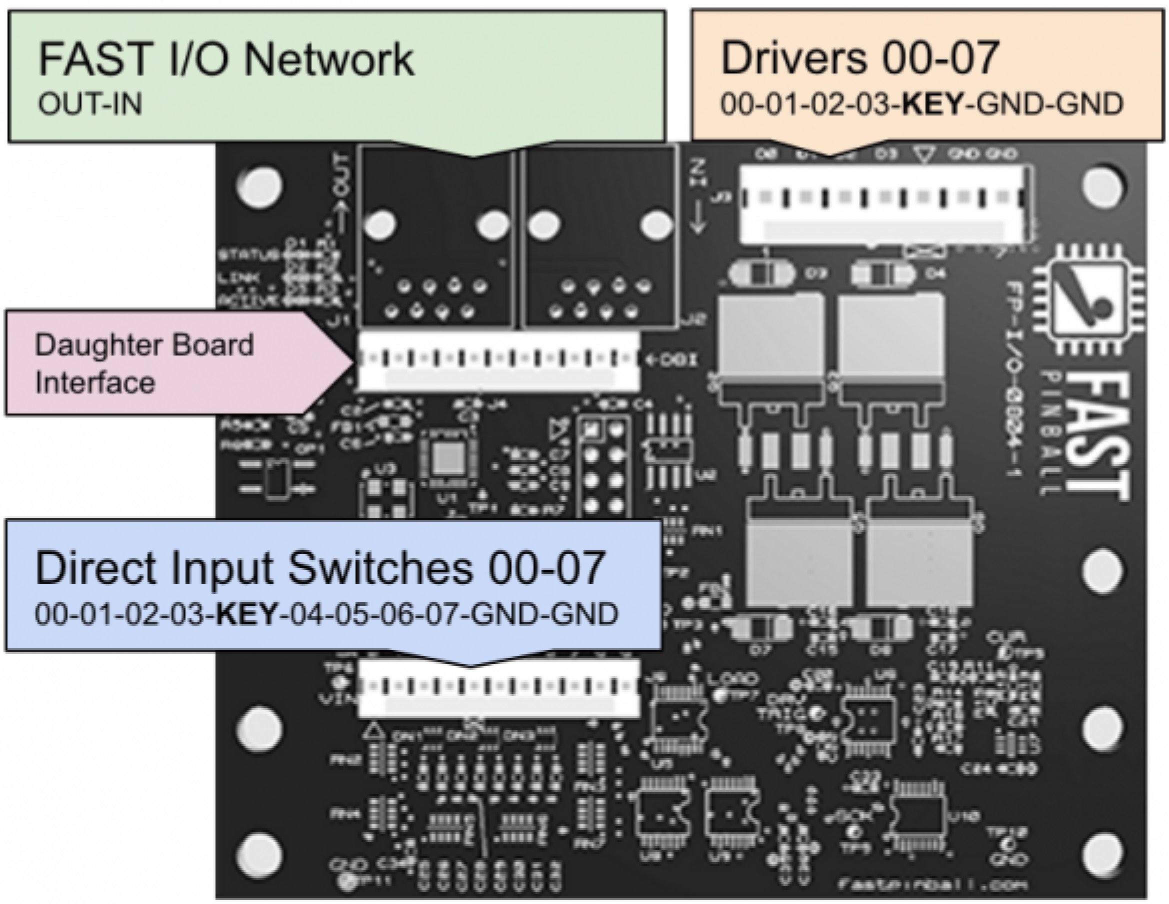
Then connect the OUT port of your NANO to the IN port of your first node board. Consequently, connect the OUT port of the first node to the IN port of your second board. Connect the OUT port of the last board back to the IN port of your NANO.

The `number:` setting for each driver/switch is its board's position number in the chain, then the dash, then the driver/switch number. Note that the position number starts with zero, so the first IO board in the chain is 0, the second is 1, etc.

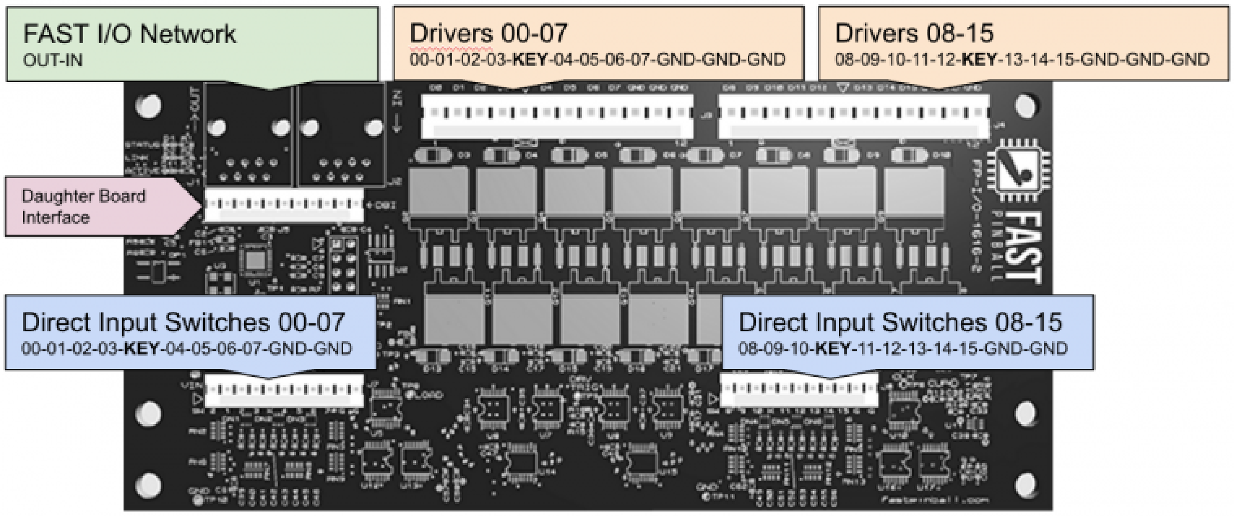
Node boards

Fast offers three different types of node boards:

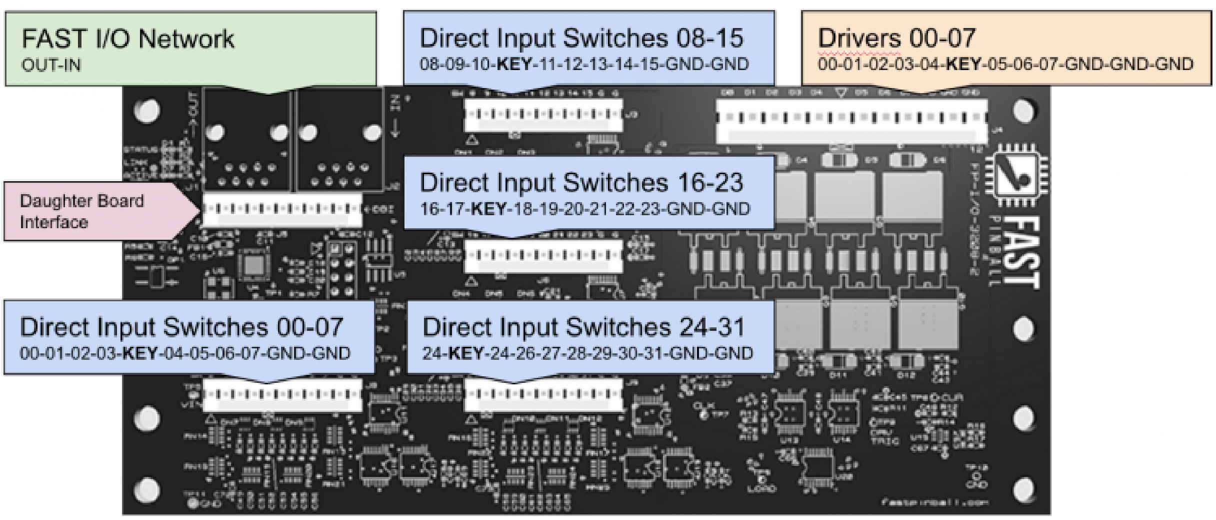
0804 - 8 Switches, 4 Drivers



1616 - 16 Switches, 16 Drivers



3208 - 32 Switches, 08 Drivers



Verify Connected Boards via mpf hardware scan

You can run *mpf hardware scan* to see all connected node boards:

```
$ mpf hardware scan

NET CPU: NET FP-CPU-002-1 01.03
RGB CPU: RGB FP-CPU-002-1 00.89
DMD CPU: DMD FP-CPU-002-1 00.88

Boards:
```

(continues on next page)

(continued from previous page)

```
Board 0 - Model: FP-I/O-3208-2    Firmware: 01.00 Switches: 32 Drivers: 8
Board 1 - Model: FP-I/O-0804-1    Firmware: 01.00 Switches: 8 Drivers: 4
Board 2 - Model: FP-I/O-1616-2    Firmware: 01.00 Switches: 16 Drivers: 16
Board 3 - Model: FP-I/O-1616-2    Firmware: 01.00 Switches: 16 Drivers: 16
```

If your boards do not show up checkout our *FAST troubleshooting guide*.

On Linux: Add udev rules to ensure persistent device names

If you have more than one ttyUSB device connected to your PC (e.g. the FAST Nano and a FAST DMD) you can assign a name to your ports based on the USB port they are connected to.

First identify the port of your FAST hardware. Usually it should be `/dev/ttyUSB0` or `/dev/ttyUSB5`.

Then run `udevadm info` on your port:

```
udevadm info /dev/ttyUSB0
```

This will show you the `DEVPATH`. Now replace the last part `ttyUSBX` with an asterisk and add an udev rules like this in `/etc/udev/rules.d/fast.rules`:

```
SUBSYSTEM=="tty", ACTION=="add", DEVPATH=="/devices/pci0000:00/0000:00:14.0/usb1/1-4/1-4:1.1/*",␣
↪SYMLINK+="ttyNET", GROUP="adm", MODE="0660"
```

After a reboot you should get a `/dev/ttyNET` device if you connect a FAST device to that specific USB port. You can use that port in your config.

What if it did not work?

Have a look at our *FAST troubleshooting guide*.

How to use install drivers & configure COM ports (FAST Pinball)

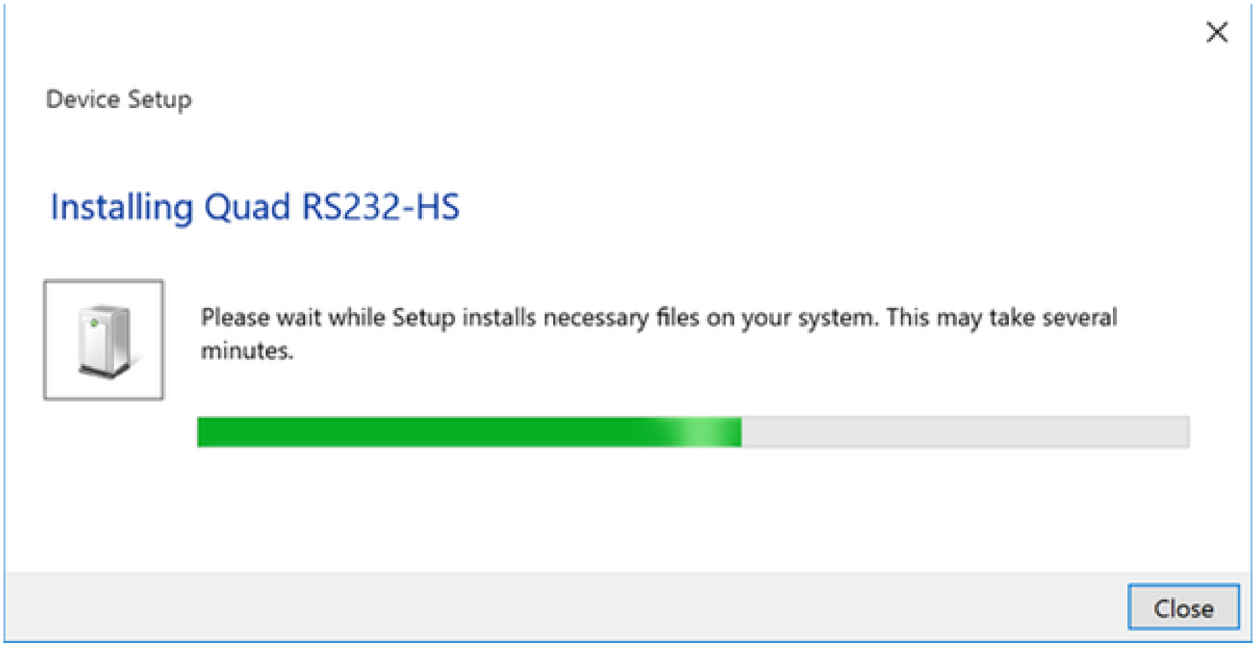
Related Config File Sections
hardware:
fast:

This guide explains how to configure MPF to work with a FAST Pinball controller. It applies to all three of their models—the Core, Nano, and WPC controllers.

1. Install the FAST USB driver

FAST Pinball controllers use a USB chip from FTDI, so you need to download and install the FTDI driver. It's pretty simple. Go to this this page and scroll down to the VCP Drivers section and download the driver for your OS. If you're using Windows, we think it's easier to use the "setup executable" they link to in the comments.

Once this is done, when you plug in and power on your FAST controller, you should see some kind of notification that new hardware has been detected. What exactly you see will depend on which FAST controller you're using and what OS you have. For example, here's what happens when you plug a FAST WPC controller into Windows 10 for the first time (after you've installed the FTDI driver):



(This is just a progress bar which shows Windows configuring the drivers. You don't have to click anything to get it started, and it should only take 5-10 seconds. It will only happen the first time you plug in the hardware.)

2. Configure your hardware platform for FAST

To use MPF with a FAST, you need to configure your platform as `fast` in your machine-wide config file, like this:

```
hardware:
  platform: fast

fast:
  driverboards: fast
```

You also need to configure the *driverboards:* entry for what kind of driver boards you're controlling.

Use `driverboards: fast` if you're using FAST I/O boards (like the 3208, 0804, etc.), or use `driverboards: wpc` if you're using an existing WPC or Snux System 11 driver board.

3. Find the FAST COM ports

Even though the FAST controllers are USB devices, they use "virtual" COM ports to communicate with the host computer running MPF. On your computer, if you look at your list of ports and then connect and power on your FAST controller, you will see 4 new ports appear. The exact names and numbers of these ports will vary depending on your computer, what other devices you have, and which port you plug the FAST controller into, but the order of which ports do what is the same everywhere:

- First (lowest numbered) port: **DMD Processor**

- Second: **NET processor** (the main processor)
- Third: **RGB LED processor**
- Fourth: **Unused** (available for your own custom use!)

Note that the FAST Nano controller does not have a DMD processor, so on that device, both the first and fourth ports are unused.

You need to tell MPF which ports are used for the FAST Controller, and the first step to doing that is to figure out what the port names are on your system:

Finding the COM ports on Windows

On Windows, it's easiest to use the Device Manager. Right-click on the Start button (or whatever it's called now) and choose "Device Manager" from the popup menu.

Then expand the "Ports (COM & LPT)" menu section to see which ports the FAST Controller is using. The easiest way to do this is to open the Device Manager to that section, then plug your FAST Controller in (or power it on) and just see which four port names appear.

The port names will start with "COM" and then be a number, and there will be four consecutive numbers to represent the four FAST ports.

Finding the COM ports on Max or Linux

On Mac or Linux, it's easiest to find the port numbers via the terminal window (or console window). To do that, open a new window and run the following command:

```
ls /dev/tty.*
```

This will list all the devices whose names begin with "tty".

The four FAST ports will have the name that starts with "tty.usbserial-", then a number, then a letter A-D. (The number will be different on every system.) The port ending with the "A" is the first port, the "B" is the second, etc.

For example, the four FAST ports might be something like on MAC:

```
/dev/tty.usbserial-141A
/dev/tty.usbserial-141B
/dev/tty.usbserial-141C
/dev/tty.usbserial-141D
```

On linux it would look like this:

```
/dev/ttyUSB0
/dev/ttyUSB1
/dev/ttyUSB2
/dev/ttyUSB3
```

If you have multiple FAST devices they will enumerate more or less randomly dependent on the order they are plugged in. Unfortunately, the USB devices do not contain any serial number. However, we can pin them based on the USB port they are plugged into. On linux this can be achieved using a UDEV rules such as this:

```
SUBSYSTEM=="tty", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6011", ENV{ID_PATH_TAG}=="pci-0000_00_14_
↪0-usb-0_12_1_0", SYMLINK+="ttyDMD1"
```

The device will then be available as /dev/ttyDMD1. You can run the following command while plugging in the device to get the relevand ID_PATH_TAG (and also idVendor and idProduct in case they changed with other revisions):

```
udevadm monitor --property
```

4. Add the ports to your config file

Next you need to add the ports to your machine config file. To do this, create a new section called `fast:`, and then add a `ports:` setting under it.

Then if you have a FAST Core or WPC controller, enter the names of the first three ports. If you have a FAST Nano controller, enter the names of the middle two ports (the second and third, since the first isn't used on a Nano).

So an example for Windows might look like this:

```
fast:
    ports: com3, com4, com5
```

And an example for Mac or Linux might look like this:

```
fast:
   ports: /dev/tty.usbserial-141B, /dev/tty.usbserial-141C
```

Note that if you have a FAST Core controller but you're not actually using the hardware DMD, then you don't have to enter the first port in your config. (Same is true if you're not using the LED controller.) MPF queries each port in this list to find out what's actually on the other end and then sets itself up appropriately.

Note that if you're using a version of Windows before Windows 10 and you have COM port numbers greater than 9, you will have to enter the port names like this: `\\.\COM10`, `\\.\COM11`, `\\.\COM12`, etc. (It's a Windows thing. Google it for details.)

There are more settings in the *fast:* section of the machine config that we have not covered here, but the ports are the bare minimum you need to get up and running.

5. Configure your watch dog timeout

FAST Pinball controllers have the ability to use a *watch dog* timer. This is enabled by default with a timeout of 1 second. If you would like to disable this, or you'd like to change the timeout, you can do so in the `fast:` section of your machine-wide config.

```
fast:
   ports: com3, com4, com5  # or whatever your ports are
   watchdog: 1000
```

The `watchdog:` setting is the timeout in milliseconds. Use 0 to disable it.

Note that at this time, FAST Pinball controllers only use the watch dog for the NET processor (which controls stuff on the IO boards, like coils). The watch dog is not used for the DMD or LEDs.

What if it did not work?

Have a look at our *FAST troubleshooting guide*.

How to configure switches (FAST Pinball)

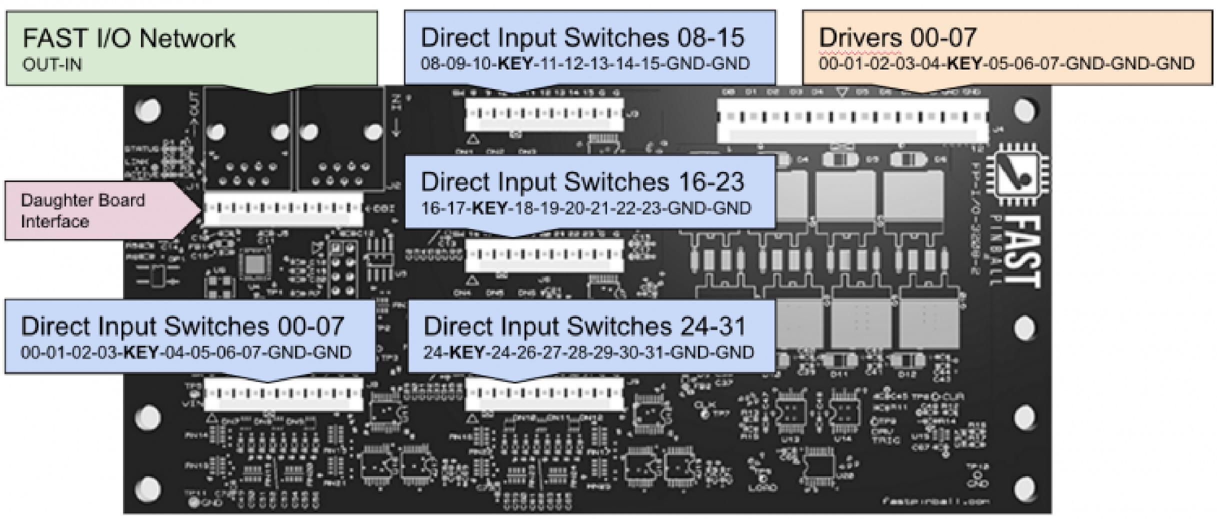
Related Config File Sections
fast:
switches:

To configure switches with FAST Pinball hardware, you can follow the guides and instructions in the *Switches* docs.

However there are a few things to know and some additional options you get with FAST hardware that is discussed here.

number:

When you're using FAST IO boards, switches plug into individual IO boards. Then the IO boards are connected together in a loop.



The `number:` setting for each switch is its board's position number in the chain, then the dash, then the switch input number. Note that the position number starts with zero, so the first IO board in the chain is 0, the second is 1, etc.

```
switches:
  my_switch:
    number: 0-0    # first board, switch 0
  some_other_switch:
    number: 2-24    # third board, switch 24
```

Notes:

- The first board in the chain is board 0.
- The boards are counted in the direction of the "out" connector on the controller board.
- Different models of IO boards have different numbers of switches, and MPF will make sure that the numbers work for each type of board. (e.g. a switch number 10 isn't valid on an 0804 board since that board only has 8 switches numbered 0-7).

Also note that prior versions of MPF just numbered all the switches in one continuous sequence from the first board through the last, but that was confusing. You can still do that if you want (in integer format), but we feel the board-input format is much easier to understand.

Debounce options

FAST controllers have advanced capabilities when it comes to debouncing switches. (More on what that is *here*).

Since FAST switches are directly connected (e.g. there is no switch matrix), and since every FAST IO board has its own processor and firmware, the states of switches are checked often (every 1ms). You can specify the exact debounce time that a switch must consistently be in a new state in both the open and close directions.

Specifying default debounce settings

By default, MPF provides two debounce profiles for switches ("normal" and "quick"). When using FAST pinball controllers, the "normal" debounce profile is `4ms` for both the debounce open and debounce closed times, and the "quick" debounce profile is `2ms` for both debounce open and closed times.

You can change any of these in the `fast:` section of your machine-wide config, like this:

```
fast:
  default_quick_debounce_open: 2ms
  default_quick_debounce_close: 2ms
  default_normal_debounce_open: 4ms
  default_normal_debounce_close: 4ms
```

(Note that other settings from the `fast:` section of your config have not been included here for clarity.)

Per-switch debounce settings

When using FAST Pinball controllers, you can also specify the debounce open and debounce closed settings on a per-switch basis. To do that, just add a `debounce_open:` and/or `debounce_close:` setting to an individual switch, like this:

```
switches:
  my_switch:
    number: 1-0
    platform_settings:
      debounce_open: 5ms
      debounce_close: 20ms
```

(continues on next page)

(continued from previous page)

```
some_other_switch:
  number: 3-24
```

Valid values are 1 to 255 ms.

What if it did not work?

Have a look at our *FAST troubleshooting guide*.

How to configure coils/drivers/magnets (FAST Pinball)

Related Config File Sections
fast:
coils:

To configure coils, drivers, motors, and/or magnets (basically anything connected to an IO board's driver outputs) with FAST Pinball hardware, you can follow the guides and instructions in the *Coils (Solenoids)* docs.

Warning: Please ensure that you have established *common ground between logic and coil power* before turning on high voltage on your coils (especially on homebrew machines). Ignoring this might lock on your coils, overheat them, burn down your house or kill you. We are serious, floating grounds are dangerous. If you are not an electrical engineer read the *guide about voltages and power*.

In a nutshell: You need to connect your logic ground (5V/12V) and your high voltage ground (48V or 80V). A *power entry or power filter board* is a convenient solution to solve this (and more) issues.

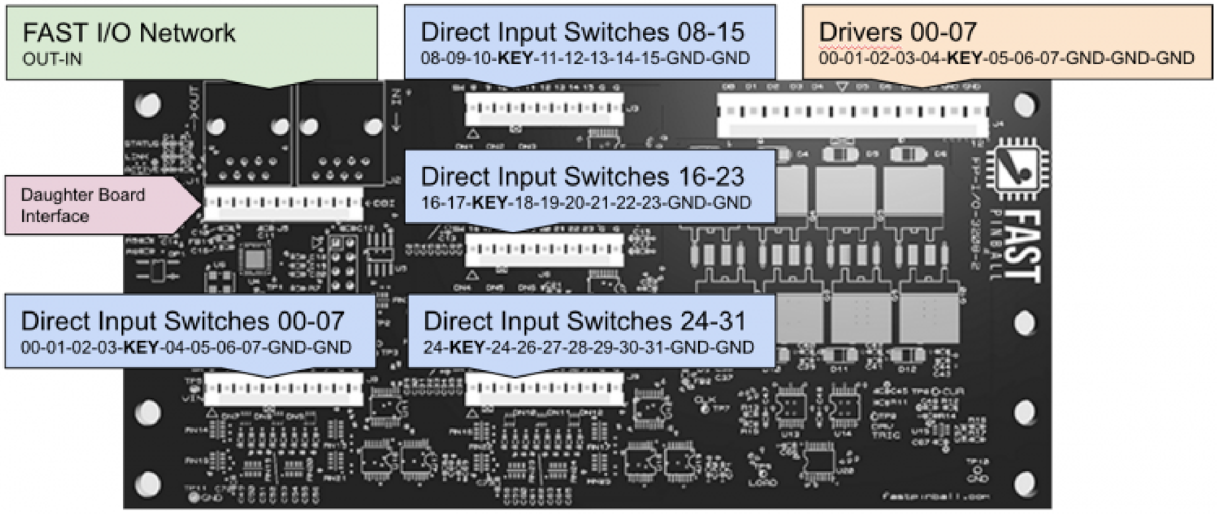
Always turn all PSUs off when connecting power or you might fry all boards at once. This is generally a good idea but even more important when connecting more than one power supply to a board.

IF YOU DID NOT UNDERSTAND WHAT THIS WARNING MEANS STOP NOW AND TRY TO UNDERSTAND IT. OTHERWISE YOUR HARDWARE WILL LIKELY BURST INTO FLAMES AND YOU NEED TO WAIT A FEW DAYS FOR A REPLACEMENT OR EVEN WORSE IT MIGHT KILL YOU. IGNORING THIS IS THE MOST COMMON CAUSE FOR BROKEN DRIVER BOARDS.

However there are a few things to know and some additional options you get with FAST hardware that are discussed here.

number:

When you're using FAST IO boards, drivers plug into individual IO boards. Then the IO boards are connected together in a loop.



The `number:` setting for each driver is its board's position number in the chain, then the dash, then the driver output number. Note that the position number starts with zero, so the first IO board in the chain is 0, the second is 1, etc.

```
coils:
  my_coil:
    number: 0-0    # first board, driver 0
  some_other_coil:
    number: 2-14    # third board, driver 14
```

Notes:

- The first board in the chain is board 0.
- The boards are counted in the direction of the "out" connector on the controller board.
- Different models of IO boards have different numbers of drivers, and MPF will make sure that the numbers work for each type of board. (e.g. a driver number 10 isn't valid on an 0804 board since that board only has 4 drivers numbered 0-3).

Also note that prior versions of MPF just numbered all the drivers in one continuous sequence from the first board through the last, but that was confusing. You can still do that if you want (in integer format), but we feel the board-input format is much easier to understand.

Pulse Power

In the *Coils (Solenoids)* section of the documentation, we talked about *how adjusting a coil's pulse time can affect its strength*. Adjusting the coil's pulse times still assumes that 100% power will be applied to that coil during that pulse time.

However, FAST Pinball controllers allow you to specify the power that's applied to the coil during the initial pulse time. This is similar to the *Adjust coil hold power*, except it applies to the initial pulse time instead of the extended hold time.

You can configure the pulse power by adding a `default_pulse_power:` setting to a coil definition and then specifying the power value from 0-1. (Like default_hold power, 0% to 100%)

For example, consider the following configuration:

```
coils:
  some_coil:
    number: 1-3
    default_pulse_ms: 30
    default_pulse_power: 0.5
```

When MPF sends this coil a pulse command, the coil will be fired for 30ms at 50% power. You can even combine default_pulse_power and default_hold_power, like this:

```
coils:
  some_coil:
    number: 1-3
    default_pulse_ms: 30
    default_pulse_power: 0.5
    default_hold_power: 0.25
```

In this case, if MPF enables this coil, the coil will be fired at 50% power for 30ms, then drop down to 25% power for the remainder of the time that it's on.

Setting Recycle Times

FAST Pinball controllers allow you to precisely control the *recycle time* for coils or drivers.

A coil's `recycle:` setting is a boolean (True/False), which is set to `False` by default. When using FAST Pinball hardware, if you set `recycle: true`, then the recycle time is automatically set to twice the coil's `default_pulse_ms:` setting. (e.g. a coil with a `default_pulse_ms: 30` and `recycle: true` will have a 60ms recycle time).

However, with FAST Pinball hardware, you can manually set a coil's recycle time by adding a `recycle_ms:` setting, like this:

```
coils:
  slingshot_r:
    number: 1-4
    default_pulse_ms: 30
    platform_settings:
      recycle_ms: 100
```

If you manually specify a recycle_ms value, then that's the value that's used and the coil's `recycle:` (true/false) setting is ignored.

Replacing FETs on FAST Driver Boards

In case you burned one of your FETs on a FAST board those can be replaced. Usually, FETs will turn on permanently when burned. As a result your coils will be stuck on and your fuse should burn (if not your coil will). If you output does not activate at all a burned FET is rather unlikely the culprit.

Consult the FAST support for an official repair. Alternatively, you can buy `IRL540NSTRLPBF` FETs from your electronics supplier and replace them yourself. Replacing SMD FETs is possible with a decent soldering iron and some practise.

What if it did not work?

Have a look at our *FAST troubleshooting guide*.

Related How To guides

- *Coil Resistance and Hardware Details*
- *Wiring Dual Wound Coils*
- *Dual-Wound versus Single-Wound coils*
- *Adjust coil hold power*
- *Adjust coil strength (pulse times)*
- *Recycle / "Cool Down" Time*
- *Details About Flippers*
- *How to configure single-wound flippers*
- *How to configure dual-wound flippers*
- *Flipper end-of-stroke (EOS) switches*

How to configure Flippers, Slingshots, Pop Bumpers, and other "quick response" devices (FAST Pinball)

MPF uses some special tricks to ensure that "quick response" devices like flippers, slingshots, and pop bumpers are able to respond to switch changes as fast as possible. (Read more about that *here*.)

When using FAST Pinball hardware, there are a few things you should know about these hardware rules.

First, remember that FAST IO boards contain both switch inputs and driver outputs.

For best performance, either:

1. Make sure switches & drivers for hardware rules are on the same IO board, *or*
2. Make sure switches are the first 8 switches on the first IO board

In other words, if you have a pop bumper or slingshot, make sure that the activation switch for that device and the coil for that device are plugged into the same IO board. That shouldn't be too hard, since you'll have multiple IO boards underneath your playfield.

For flippers, however, that's probably not possible, so FAST Pinball controllers use the concept of "priority" switches which are the first 8 switches plugged into the first board in the chain. (These will be the switches numbered `0-0` through `0-7`.)

These priority switches are sent across the FAST loop network immediately which means they can be used with hardware rules to trigger drivers (coils) on any IO board in the network.

If you only have two flippers in your machine, this is probably nothing you'll ever need to worry about since it will be easy to connect the flipper switches and coils to the same IO board (and of course the same will be true for all the other quick response devices in your machine).

But if you have more than two flippers, there's a good chance that the additional flippers will be somewhere far away from the flipper buttons and the main flippers. In that case, no problem, but make sure the IO board that has your flipper buttons connected to it is the first one in the chain, and make sure your flipper buttons are connected to one of the 0-7 positions on that IO board, and then everything will be fine.

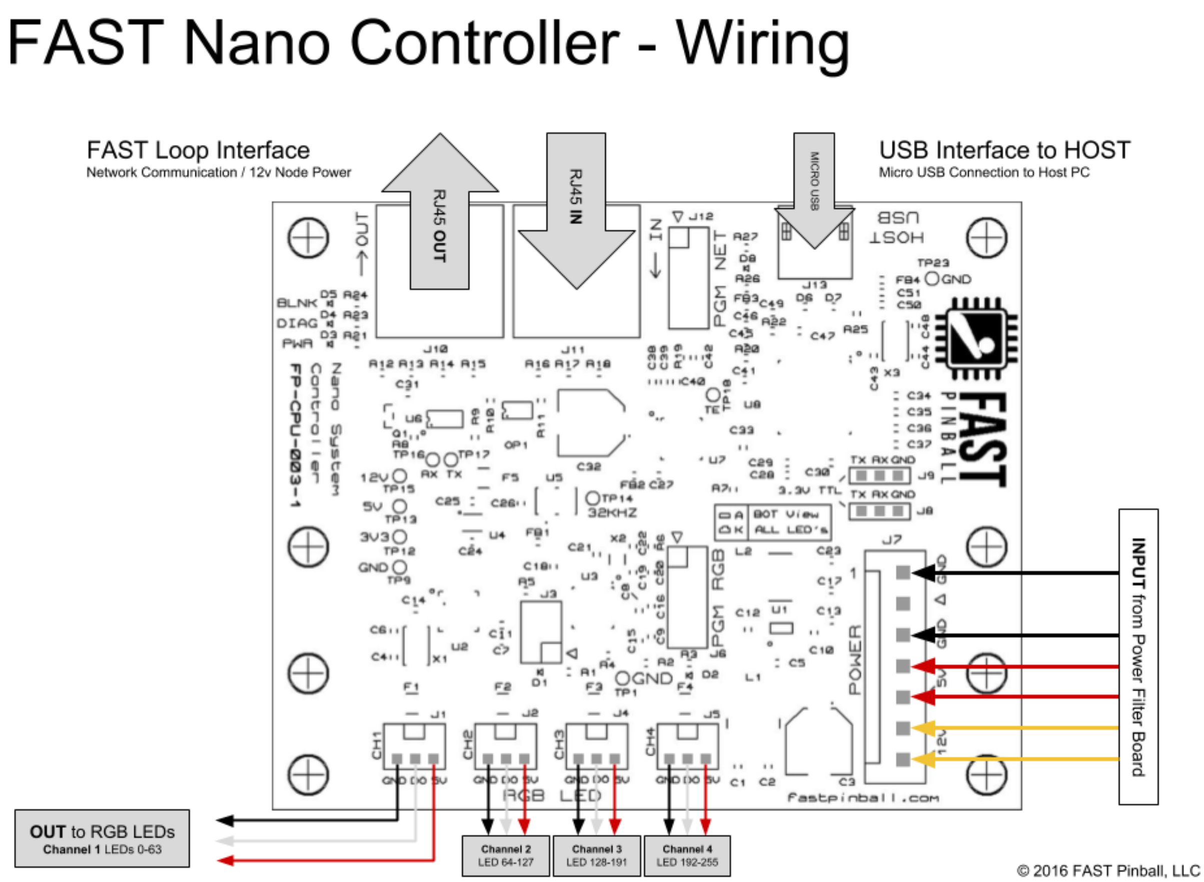
What if it did not work?

Have a look at our *FAST troubleshooting guide*.

How to configure LEDs (FAST Pinball)

Related Config File Sections
leds:
fast:

Each FAST Pinball Controller has a built-in 4-channel RGB LED controller which can drive up to 64 RGB LEDs per channel. This controller uses serially-controlled LEDs (where each LED element has a little serial protocol decoder chip in it), allowing you to drive dozens of LEDs from a single data wire. These LEDs are generally known as “WS2812” (or similar). You can buy them from many different companies, and they’re what’s sold as the “NeoPixel” brand of products from Adafruit. (They have all different shapes and sizes.)



Most of the settings in the *Lights* documentation apply to LEDs connected to FAST Pinball controllers, however there are a few FAST-specific things to know.

Overview video about serial LEDs:

https://youtu.be/Q9BG9T7Kj4A

Channel and Number Syntax

In MPF *lights* abstract a light source which emits arbitrary colors. However, this is not true for all real lights. Some support only white (GIs), others only a single-color (i.e. red inserts) and others support full RGB. For that reason MPF knows *light numbers and channel numbers*. Internally, a light consists of one or multiple channels. For instance, a single-color GI will contain a single white channel. While a RGB light will control a red, green and a blue channel. A white light behind a red insert should be a single red channel (because it cannot emit other colors through the red insert). You can configure those channels using the `channels` setting or use `start_channel` and `type` to define the channels. See *Lights* for details.

However, in most cases a platform supports one type of lights (per `subtype`) this would be overly verbose and we added the `number` setting for configuring lights in the common platform way. For instance a platform for GIs will configure single channel white lights or a serial LED controller will configure RGB lights with three channels.

FAST assumes RGB lights by default. For everything else (i.e. RGBW) you have to use channels.

The FAST Nano supports 256 LEDs across four chains (listed as "CH 1" - "CH 4" on the Nano). LEDs 0-63 are on chain 1, 64-127 on chain 2, 128-191 on chain 3 and 192-255 on chain 4 (please note that FAST diagrams label these 4 headers as "channels" but we are using the term "chains" to avoid confusion with the non-RGB "Channels" section below).

Light Numbers

FAST numbers use the format: `number`

This is as easy as it gets. Just provide the number of you LED in the chain. Internally, FAST assumes three channels per LED (RGB/GRB *WS2811/WS2812 LEDs*).

Channels

FAST channels use the format: `number-index`

`number` is the same as above and `index` is a an index from 0 to 2. This is because serial LEDs are traditionally RGB (or GRB) LEDs with exactly three channels. However, this is not true for RGBW or similar LEDs which do not work with this style of numbering. Luckily, you can chain them instead and have MPF calculate the internal channels for you:

```
lights:
  led_0:
    start_channel: 0      # you could also use number: 0
    subtype: led
    type: rgb    # will use red: 0-0, green: 0-1, blue: 0-2
  led_1:
```

(continues on next page)

(continued from previous page)

```
    previous: led_0
    subtype: led
    type: rgbw   # will use red: 1-0, green: 1-2, blue: 1-3, white: 2-0
  led_2:
    previous: led_1
    subtype: led
    type: rgbw   # will use red: 2-1, green: 2-2, blue: 3-0, white: 3-1
```

See *WS2811 and WS2812 LEDs in Pinball* for details.

RGB LED buffering

Most computers have the ability to send LED updates to the FAST Pinball controller faster than the controller can process them. If this happens, then the LED command messages can get backlogged and it will appear that you have a "delay" in your LEDs and/or you might get weird colors due to corrupt messages.

To help combat this, there are two settings you can adjust:

```
mpf:
  default_light_hw_update_hz: 50
fast:
  rgb_buffer: 3
```

If you notice that your LEDs seem to be getting behind, you can adjust the `default_led_hw_update_hz:` setting to be lower. (Frankly the 50hz by default is too high and we should lower it to 30.) You can probably drive 128 or so LEDs at 50Hz, but if you have more than that then you might need to start playing with this number.

Hardware LED fading

You can globally set the fade rate for LEDs connected to a FAST Pinball controller via the `fast:hardware_led_fade_time:` setting. (This is `0ms` by default, meaning it's disabled.)

See the *fast:* section of the config file reference for details.

Color Correction

If you are using RGB LEDs, they might not be perfectly white when you turn them on. They might be pinkish or blueish instead depending on the brand of the LED. To a certain extend this is normal/expected and you can compensate for it by configuring *color_correction profiles in light_settings*.

What if it did not work?

Have a look at our *FAST troubleshooting guide*.

How to configure Matrix Lights (FAST Pinball)

Related Config File Sections
lights:

Matrix lights are currently only supported on FAST Pinball via their WPC Controller. Like the other WPC-related settings in MPF, you can enter the numbers right out of your operators manual, so there's nothing FAST-specific you have to do.

What if it did not work?

Have a look at our *FAST troubleshooting guide*.

How to configure mono/traditional DMD (FAST Pinball)

Related Config File Sections
dmds:

The FAST WPC and Core controllers can drive traditional single-color pinball DMDs via the 14-pin DMD connector cable that's been in most pinball machines for the past 25 years, like this:

It makes no difference as to whether you're using an LED or an original plasma gas DMD. (Also it doesn't matter what color it is.)

1. Verify your port settings

In order to use a DMD with a FAST Pinball controller, you need to have the port that's connected to the DMD processor on the FAST board listed in the `ports:` section in the `fast:` section of your machine-wide config.

See the *How to use install drivers & configure COM ports (FAST Pinball)* guide for details.

2. Add a physical DMD device entry

Once you have your hardware and port set, you need to create the actual device entry for the DMD.

You do this in the `dmds:` section of the machine config. This section is like the other common sections (switches, coils, etc.) where you enter the name(s) of your device(s), and then under each one, you enter its settings.

(And yes, in case you're wondering, it's possible to have more than one physical DMD.)

To do this, create a section in your machine-wide config called `dmds:`, and then pick a name for the DMD, like this:

```
dmds:
  my_dmd:
    shades: 16
```

You need to have at least one setting for this to be a valid YAML file, so we usually just pick the `shades` and add that with a value of `16` (which means the DMD runs will convert the display content to 16 mono shades when it displays it).

The "shades" option is how many brightness shades you want. 1990s WPC machines supported 4 shades, and modern Stern DMD machines support 16. The FAST Pinball controllers support 16 shades (even on older 1990s plasma DMDs). Most modern games will probably be 16 shades, but you can do 4 (or even 2) if you want an old school look.

There are lots more options for the physical_dmd: section than just the "shades" option listed here. Check the *dmds:* for a list of all the options.

Note that one option you do NOT have for physical DMDs is the color. That's because the color of the DMD is determined by the DMD itself. You don't actually send it color values, rather, you just send it brightness levels, and the DMD shows those brightness levels with whatever color the DMD is.

3. Set a source display

Now that you have everything configured, the last step is to make sure the DMD knows what content to show. In MPF, you do this by mapping a physical DMD to an *MPF display*.

By default, the DMD will look for a display (in your *displays:* section called "dmd". However you can override this and configure the DMD to use whatever logical display you want by setting a `source_display:` setting. (Just make sure that the width and height of your source display match the physical pixel dimensions of the DMD or else it will be weird.)

A final config you can test

At this point you're all set, and whatever slides and widgets are shown on the DMD's source display in MPF-MC should be shown on the physical DMD.

That said, all these options can be kind of confusing, so we created a quick example config you can use to make sure you have yours set right. (You can actually just save this config to `config.yaml` in a blank machine folder and run it to see it in action which will verify that you've got everything working properly.)

To run this sample config, you can run `mpf both`.

When you run it, do not use the `-x` or `-X` options, because either of those will tell MPF to not use physical hardware which means it won't try to connect to the Teensy.

Note that the *Using a traditional (single color) physical DMD* guide has more details on the window and slide settings used in this machine config.

```
hardware:
  platform: fast
fast:
  ports: com3, com4, com5  # be sure to change this to your actual ports
  driverboards: fast
displays:
  window:  # on screen window
    width: 600
    height: 200
  dmd:  # source display for the DMD
    width: 128
    height: 32
    default: true
window:
  width: 600
  height: 200
  title: Mission Pinball Framework
  source_display: window
dmds:
  my_dmd:
    brightness: 1.0
slides:
  window_slide_1:  # slide we'll show in the on-screen window
    - type: display
      effects:
        - type: dmd
          dot_color: ff5500
      width: 512
      height: 128
    - type: text
      text: MISSION PINBALL FRAMEWORK
      anchor_y: top
      y: top-3
      font_size: 30
    - type: rectangle
      width: 514
      height: 130
      color: 444444
  dmd_slide_1:  # slide we'll show on the physical DMD
    - type: text
      text: IT WORKS!
      font_size: 25
slide_player:
  init_done:
    window_slide_1:
      target: window
    dmd_slide_1:
      target: dmd
```

What if it did not work?

Have a look at our *FAST troubleshooting guide*.

How to configure an RGB DMD (FAST Pinball)

Related Config File Sections
rgb_dmds:

If you would like to use the FAST RGB LED DMD, follow the instructions for the *How to configure a "SmartMatrix" RGB LED DMD*.

You can copy the following example (and replace `com12` with your com port):

```
hardware:
  rgb_dmd: smartmatrix
smartmatrix:
  smartmatrix_1:
    port: com12
    baud: 4000000
    old_cookie: false
```

What if it did not work?

Have a look at our *FAST troubleshooting guide*.

How to configure servos (FAST Pinball)

Related Config File Sections
servos:

You can drive servos from any FAST IO board by adding the FAST Servo Controller daughter board to it. You then configure and use the servos like normal. The only real "FAST-specific" thing is the number.

Overview video about *servos*:

https://youtu.be/wA6KEODwQ5w

number:

The number of the servo requires a bit of math. Each FAST IO board "reserves" six slots for daughter board accessories (regardless of whether there's a daughter board there are not). So the numbers go like this:

- First board in the chain (Board 0), numbers 0, 1, 2, 3, 4, 5
- Second board in the chain (Board 1), numbers 6, 7, 8, 9, 10, 11
- Third board in the chain (Board 2), numbers 12, 13, 14, 15, 16, 17

- Fourth board in the chain (Board 3), numbers 18, 19, 20, 21, 22, 23
- etc.

So to figure out the number for your servo, first figure out which board it's plugged into, then look at which connection on that board it uses, then figure out the number based on the list above.

By default, standalone numbers like this have to be entered in hex format, so once you find your number, enter it as the hex equivalent:

Regular	Hex
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	a
11	b
12	c
13	d
14	e
15	f
16	10
17	11
18	12
19	13
20	14
21	15
22	16
23	17

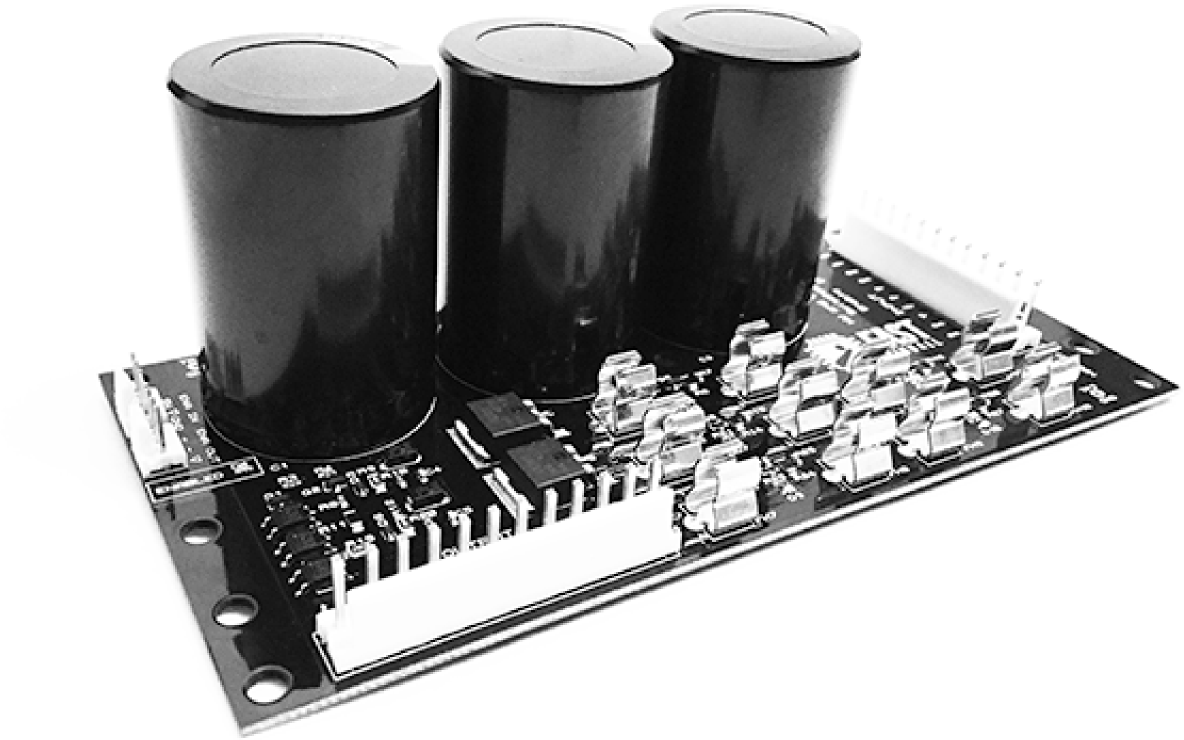
If you don't want to mess with all this hex stuff, you can set the config number format to "int" via the `fast: config_number_format:` setting. See the *fast:* section of the config file reference for details.

What if it did not work?

Have a look at our *FAST troubleshooting guide*.

Power Filter Board

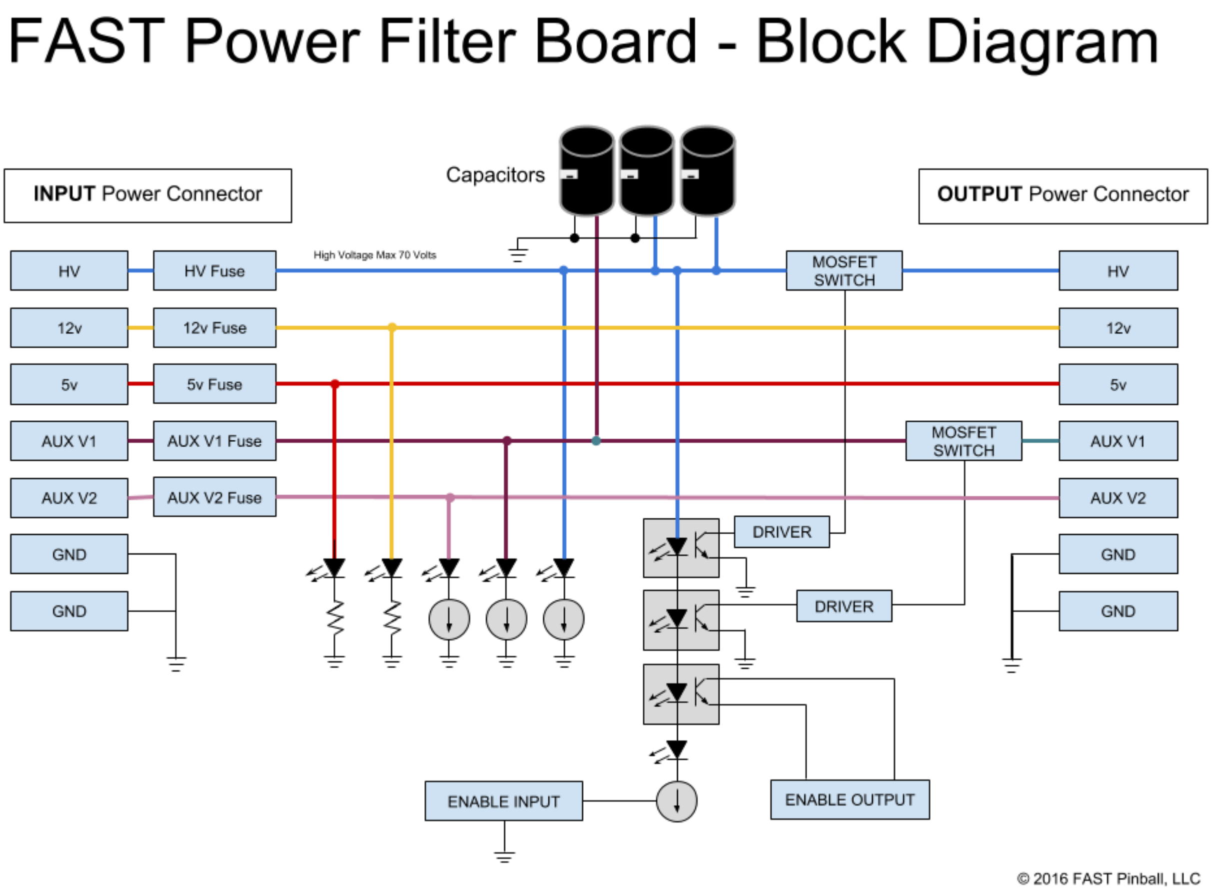
This board can be used to fan out your power rails. See *Voltages and Power in Pinball Machines* for details.



The board supports 5 power rails with one fuse per rail:

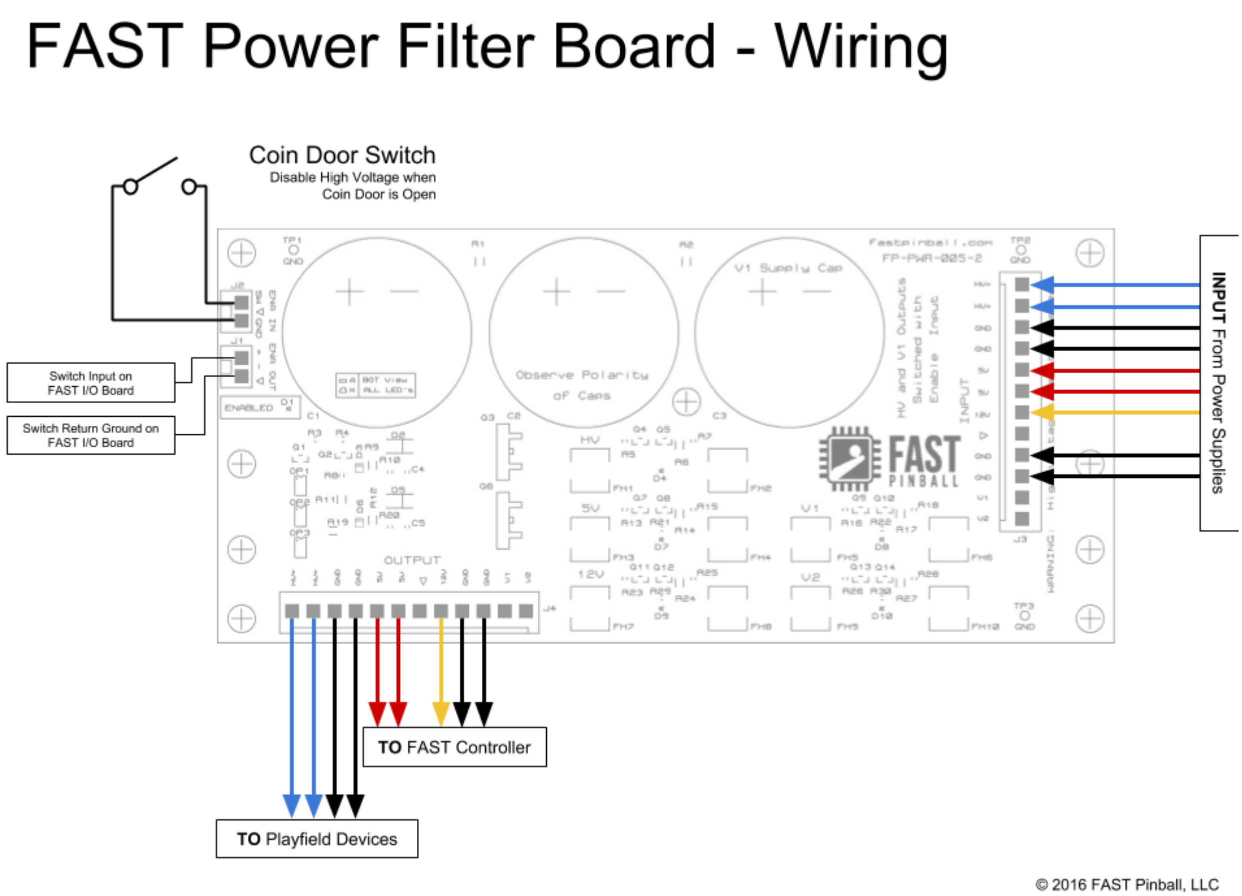
- High Voltage (HV)
- Aux V1
- Aux V2
- 5V
- 12V

There are capacitors on HV and Aux V1. This is the theory of operations:



Additionally, there is a high voltage enable switch on the board on J2. You can connect it to your door switch to cut power when the door opens. Make sure to close this switch when you operate the machine or HV will be off. During development you may use a jumper but be careful since HV will be always on.

Connect all your PSUs to J3 and the playfield and controller to J4. This is how FAST envisions the wiring of the board:



Troubleshooting FAST

If you got problems with your hardware platform we first recommend to read our *troubleshooting guide*. Here are some hardware platform specific steps:

Run Hardware Scan

Using `mpf hardware scan` you can find out if your Nano is talking properly to MPF using USB. Additionally, it will show you which node boards are connected:

```
$ mpf hardware scan

NET CPU: NET FP-CPU-002-1 01.03
RGB CPU: RGB FP-CPU-002-1 00.89
DMD CPU: DMD FP-CPU-002-1 00.88

Boards:
Board 0 - Model: FP-I/O-3208-2    Firmware: 01.00 Switches: 32 Drivers: 8
Board 1 - Model: FP-I/O-0804-1    Firmware: 01.00 Switches: 8 Drivers: 4
Board 2 - Model: FP-I/O-1616-2    Firmware: 01.00 Switches: 16 Drivers: 16
Board 3 - Model: FP-I/O-1616-2    Firmware: 01.00 Switches: 16 Drivers: 16
```

If you are missing boards here check your wiring. Also verify that firmware versions match. In the example above the NET CPU has firmware 1.03 but the nodes still run on 1.00 which indicates an issue. See *mpf hardware (command-line utility)* for details about the command.

Stuck on Drivers

See the section about *Replacing FETs on FAST Driver Boards* if you suspect burned FETs.

Permission Denied on Linux

If you see an error such as:

```
serial.serialutil.SerialException: [Errno 13] could not open port /dev/ttyUSB1: [Errno 13] Permission
↪denied: '/dev/ttyUSB1'
```

Your user does not have sufficient permissions to access that port. You could run MPF as root but we do not recommend that. Alternatively, you can create a udev rule or add your user to the dialout group:

```
sudo usermod -a -G dialout $USER
```

After a restart of your PC MPF should be able to access that serial port.

Enable Debugging

If you got problems with your platform try to enable `debug` first. As described in the *general debugging section* of our *troubleshooting guide* this is done by adding `debug: true` to your `fast` config section:

```
fast:
  debug: true
```

This will add a lot more debugging and might slow down MPF a bit. We recommend to disable/remove it after finishing debugging.

Firmware Upgrade

MPF generally works with the latest firmware for FAST. There have been some protocol changes between firmware and we do not usually test our software with older firmware version. Consider upgrading to the latest firmware. You can find out your current firmware version using `mpf hardware scan` (see above).

Coils Are Not Firing

What to do if your coils are not working?

Check if Your Hardware is Working at all

Sounds stupid but this is a good start: Is the hardware working at all? Do you see switch hits in the logs? If not, check our section `Your hardware is not working at all`.

Check the Watchdog

If switches (or other features of the platform) are working but coils are not we have to dig deeper. Most hardware platforms have some kind of watchdog. Often there is some LED which indicates if the watchdog is received. The MPF log might also contain clues (especially if you have enabled `debug` and run MPF with verbose flags `-v -V`). If the watchdog is not received by your platform it will not enable coils.

In most cases watchdog related problems indicate wiring problems. Check if your boards are properly wired.

Test Your Coil Numbers using MPF Service CLI

Hardware is connected and generally working, watchdog is good but still your coils are not working? Maybe something with the numbering is odd. Lets tests that using the *MPF Service CLI*. Alternatively, you can also use *service mode* if you have already configured it. Both ways work similarly.

To use service cli:

1. Open two consoles
2. Start your game (e.g. using `mpf both`)
3. Start the service cli from within your game folder using `mpf service`.
4. Type `list_coils` and press `ENTER` to see a list of coils.
5. Type `coil_pulse your_coil` and press `ENTER` to pulse it.

Does it work? If not check the log and try verify the coil number. If you do not specify `default_pulse_ms` MPF will use `10ms` which might not be enough for some mechs. Try to increase that gently (maybe `20ms` or `30ms`).

Reducing light update rate

If you got a lot of lights you might run into bus contention issues. You can reduce the light update rate in MPF:

```
mpf:
  default_light_hw_update_hz: 30   # defaults to 50
```

If you set this too low fades will be less smooth but otherwise it should not affect your game.

Your hardware is not working at all

If your hardware is not working at all make sure that you removed the options `-X`, `-x` and `--vpx` from your `mpf both` or `mpf game` command line. Those options will overwrite the settings in your `hardware` section and MPF will not even try to connect to your hardware. If you got config errors we suggest you add `-X` to figure things out without interfacing real hardware all the time. Just keep that option in mind.

Another stupid thing to check: Is your hardware connected to your PC? We know it is stupid but a loose USB connector has happened to most of us.

On Linux you might want to run the command `lsusb` which should show both of your micro controllers connected. You should see two lines similar to

```
Bus 002 Device 014: ID 0483:5740 STMicroelectronics Virtual COM Port
Bus 002 Device 015: ID 0483:5740 STMicroelectronics Virtual COM Port
```

If you are unsure about the output, run the command once with your controllers connected and once without. If there is no difference, then for sure the USB device is not properly connected.

Add debugging to related devices

If you got problems with some switches also add `debug: true` to those as it will give to more insights into the intentions of those devices. Same will work for flippers, coils, lights, servos, steppers and more. See *general debugging section* for details.

Run MPF with verbose flag

See *general debugging section* for details. TLDR: run `mpf both -t -v -V`.

Report Your Issue and Ask For Help

If you cannot find the issue yourself please prepare some information about your issue according to our *troubleshooting guide* and ask in our forum.

Consider Improving the Documentation

Did you solve your issue but found that some relevant information in the documentation is missing or should be linked/located elsewhere? Either tell us in the forum or consider *improving the documentation* yourself to save future users some troubles the same way others saved you some troubles by writing this documentation.

How to use MPF with Stern SPIKE / SPIKE 2 machines

Related Config File Sections
hardware:
spike:

If you haven't done so already, be sure to read the *MPF Overview* page to understand how MPF talks to physical pinball machines in general.

Stern pinball machines from early 2015 (Wrestlemania) onwards use a control system called SPIKE (or SPIKE 2 from Batman 66 onwards). The complete list of SPIKE machines is available in IPDB (click here for SPIKE and SPIKE 2 machines).

You can read all about how SPIKE works in the operators manuals for the games, but the important thing to know here is that SPIKE machines essentially have a full linux computer inside them (the "SPIKE CPU Node") which runs the game code from an SD card.

If you want to use MPF to control or power a Stern SPIKE system, you can make some small changes to the SD card to enable external control and then connect the computer running MPF to the CPU Node via USB.

Note: When you use MPF with a Stern SPIKE machine, MPF itself does not run "on" the SPIKE CPU Node. Rather you still run MPF on a host computer (your laptop, a Raspberry Pi, a mini-ATX motherboard in the machine, etc.), and it connects to the SPIKE CPU node via a serial or USB connection to control the machine.

Doing so gives you full control of the machine. You can read the states of switches, fire coils, set LEDs, etc. Then you can use MPF to write your own game code, just like any other platform.

Note that you *cannot* access any of the existing Stern game rules, code, or assets (videos, images, sounds, etc.) All of that is compiled into the original game code on the SD card and protected by copyright. So if you just want to do a "small tweak" to the rules of a Stern SPIKE machine, then MPF is not the right tool for that. Instead MPF would be used to completely rewrite the game from scratch, either to write a different version for the existing machine or to retheme the machine into something of your own creation.

Note: The MPF to Stern SPIKE bridge & support is new and EXPERIMENTAL. Much of this will change in the next weeks and months as we get more real world experience with it.

Warning: It's possible that using the MPF SPIKE bridge will void your warranty. For example, maybe you build a config or MPF contains a bug that holds a coil on too long and it burns up your machine. Use it at your own risk. It's also possible that you will not void your warranty. We are not lawyers and don't know.

Warning: If you break or corrupt your original SD card with your Stern game code on it, you may have to get a new one from Stern support. Again, proceed at your own risk only if you know what you're doing.

Fundamentally, using MPF with a Stern SPIKE system is like putting a P-ROC in a Williams WPC machine. All it does is expose the hardware to a computer which you can then control, and you're on your own in terms of rules and assets and code and everything. The advantage of using a SPIKE machine is you don't have to buy a $325 P-ROC, and you can swap back-and-forth between the original rules and your own code by changing an SD card versus having to unplug and re-plug a bunch of wires to swap out a board.

Stern SPIKE features that work today

- *Coils / drivers*
- *Switches*
- *LEDs & GIs*
- *Backbox LEDs*

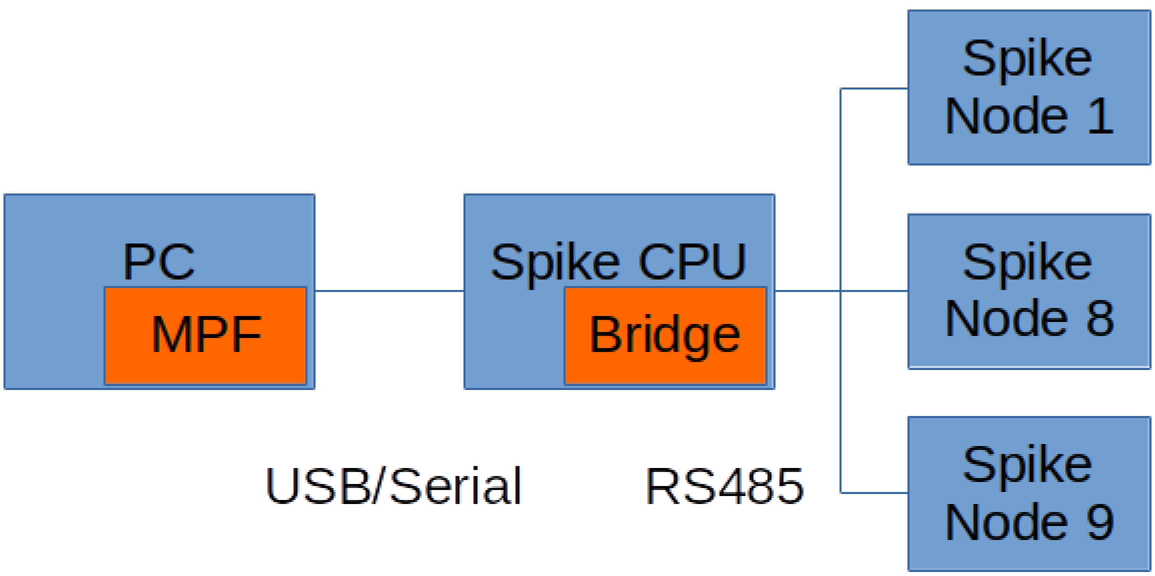
- Hardware Rules (flippers, pop bumpers, slingshots, etc.)
- *DMDs*
- *Steppers*

How does the MPF SPIKE interface work?

Here's a more technical overview of how MPF talks to a Stern SPIKE machine. You don't have to read this section if you don't care.

Stern SPIKE hardware is a series of node boards that are connected via Cat-5 cables which is known as the SPIKE node bus. The CPU running the game code from the SD card on the CPU node sends commands to individual node boards to actuate drivers and set LEDs and stuff like that, and it receives switch state updates from node boards with switches attached.

When you use a Stern SPIKE machine with MPF, you install a piece of software called the "MPF SPIKE Bridge" on the SD card (ideally you first make a copy of your existing SD card and keep the original in a safe place), and then when the machine powers on, instead of running the existing game code from the SD card, the CPU runs the MPF SPIKE bridge software.



The MPF SPIKE bridge is fairly simple. Essentially all it does is relay messages from the SPIKE node bus to the debug port on the CPU node, and it also accepts commands sent via the debug port and retransmits them to the node bus.

So in order to connect a computer running MPF to the Stern SPIKE machine, you buy a small USB-to-serial adapter (Amazon.com has them for under $10) and connect one end of it to the CPU node's debug header, and you plug the other end into your computer which is running MPF. (That can be Windows, Mac, Linux, Raspberry Pi, etc. Just a regular computer running the regular version of MPF.)

From there you just configure MPF like regular. You set the platform to "spike", you set the port that your USB-to-serial adapter is using, and you set all your coils, switches, and LEDs based on their node board & IDs from the operator's manual.

If you ever want to go back to the original game code from Stern, then just swap out the SD card with the MPF SPIKE Bridge on it and replace it with the original card from Stern and you're all set.

Stern SPIKE features that do not work (yet)!

Sound

Currently if you want to use sound (which of course you do), the way to do it is to use the sound card in the computer running MPF and speakers connected there.

The SPIKE system has sound capabilities, and it would be nice to be able to use it along with its existing speakers and amps, but the way MPF connects via the debug port does not allow for enough bandwidth for us to do sound this way.

This is something that might change in the future, or perhaps we can find an easy way to connect the sound output from the computer to the SPIKE amp.

Servos

Once we get access to a SPIKE machine with servos, we'll get support for them added.

Small LCD from WWE

WWE LEs have a small playfield LCD which is controlled via the SPIKE node bus. MPF does not yet support this, though of course you could use any HDMI display connected to the machine running MPF.

How to modify a Stern SPIKE SD card & install the MPF SPIKE bridge

1. Backup the existing SD card

When you download firmware updates from Stern's websites to a USB stick, the updates only contain the specific parts of the code that have changed since the original version.

In other words, if you break or somehow screw up the SD card with the SPIKE game code on it, you will not be able to fix it by re-downloading the latest firmware. (You'll have to call Stern and get a new SD card with the software already on it.)

So be very careful here.

Our recommendation is to create an image of the original SD card, and then put the original in a safe place and then copy the image to a new SD card. That way you're always working with a copy and the original SD card is never touched.

Note that we do not yet know which cards are best or will be fully compatible, so our recommendation is to get a card that's around the same size as the current one. Let us know what you find in terms of what works and what doesn't!

Known SD Cards that work: SanDisk Ultra Plus 16GB purchased from Best Buy

One tool you can use to backup an image of your SD Card is HDD Raw Copy. This tool will back up a copy to your local drive and you can restore it to the new SD card. For a tutorial on backing up your Stern SD card using HDD Copy check out the following video (starting from second 35):

https://youtu.be/KlKw8raWixI

Note: Save a copy of your SD card image in case you need to restore your SD card. If, at one point, your SD memory card becomes corrupted, restoring from the backed up image fixes the issue.

2. Mount the SD card

You need to mount the Linux root partition (which is probably #3).

On Windows you need an additional tools to mount ext3. We got a report that "Paragon ExtFS for Windows" works fine for this.

On Mac OS X, the tool "FUSE-ext2" is an option. You will most likely need to use sudo, and depending on your configuration the appropriate disk device may vary. In the following example, the Linux root is on partition 3 of the SD card, which is disk2:

```
sudo fuse-ext2 /dev/disk2s3 /Volumes/SD -o force
```

3. Edit /etc/inittab

Last line needs to be changed to enable login without a password:

```
S0:2345:respawn:/sbin/getty 115200 ttyS0 -n -l /bin/sh
```

Furthermore, you might want to add this line to allow USB login (e.g. if your board does not have DBGU populated).

```
USB0:2345:respawn:/sbin/getty 115200 ttyUSB0 -n -l /bin/sh
```

If your USB to serial adapter has a "RTS" and "CTS" pin or if you are using a null-modem cabel you can enable hardware flow control. In that case use the following line (notice that we added `-h`):

```
USB0:2345:respawn:/sbin/getty 115200 ttyUSB0 -h -n -l /bin/sh
```

4. Edit /etc/rc2.d/S95game

Add the following two lines as the new second and third lines in this file:

```
/usr/local/bin/avrisp /usr/local/spike/netbridge.hex /usr/local/spike/netbridge.fuses
exit 1
```

This causes this script to exit instead of running the original Stern game code. (You can remove this line again if you want to run the original game again.)

5. Install the spike bridge

Add mpf-spike-bridge to /bin/bridge and mark it as executable.

On Linux this can be done with *chmod +x bridge* from within the folder.

Get the bridge from https://github.com/missionpinball/mpf-spike

Note that we have a precompiled binary in there (as well as the Rust source code).

Note: It might be hard to mark the bridge binary as executable on Windows (but should be possible). If you cannot do this proceed to the next step and afterwards do the following:

1. Download PuTTY from www.putty.org. PuTTY is a free telnet app that allows you to remotely connect to the Linux OS running on the SPIKE system. PuTTY was also useful for verifying the connection from your Windows machine to the Linux OS running on SPIKE.
2. In PuTTY, select the "Serial" buttont, change to correct COM (COM1, COM3, COM4, etc) port and set speed to 115200 baud. If you are unsure of which COM port Windows used when you plugged in your cable, open the Device Manager in the Control Panel. Click open the PORTS drop down to find which COM port is in use.
3. Power up spike
4. Press enter and you should get a command promt (if not, your serial connection is probably not working).
5. Type the following:

```
mount -o remount,rw /
chmod +x /bin/bridge
mount -o remount,ro /
```

Note: On OS X with fuse-ext2, overwriting files can fail without a message. When updating mpf-spike-bridge, you may want to remove the old bridge file before copying the new one.

:: rm <sd_mount>/bin/bridge cp <your_path>/mpf-spike-bridge/bridge <sd_mount>/bin/bridge
chmod 755 <sd_mount>/bin/bridge

---

6. Unmount the SD card. Put it back in your spike system

Unmount the card. Really! Do that! Spike will not boot from a corrupted filesystem. SD cards may need a while to write everything. Give them those extra 10s. This is particularly important on Windows. If the red LED in the middle of the Stern CPU board is not blinking your SD card may be corrupt.

Note: The SD card can become corrupted when removing the card without ejecting it properly. You can fix this by restoring your backup from above.

Now when you power up the pinball machine, instead of running the original game code, it will run the spike bridge which will listen for commands from the CN2 connector and will send out information about the state of the machine via that connector.

What if it did not work?

Have a look at our *SPIKE troubleshooting guide*.

Connecting your computer to the Stern SPIKE CPU node

Related Config File Sections
hardware:
spike:

There are at least 3 options to connect a computer running MPF to the SPIKE CPU via a serial connection.

1. USB to USB Null Modem Cable
2. USB to Serial Adapter
3. Using two USB to Serial Adapters

OPTION 1: USB to USB Null Modem Cable

Probably the cleanest and easiest method is to purchase the USB to USB Null Modem Cable. With this cable, you can plug one end into the USB port on your computer and the other end into one of the two USB ports on the SPIKE board. On a Windows computer, use the Device Manager to determine which COM port the cable has been assigned by Windows. Update you machine configuration with the correct COM port (example, COM5).

```
spike:
  port: COM5
```

Null modem cables used to be a common way to connect two computers together. This is the most expensive solution at about $50 USD. However it looks just like a USB cable. The only vendor that has the USB to USB Null Modem Cable is the FDTI company.

https://ftdichip.com/products/usb-nmc-2-5m/

This particular cable also provides faster data transfer rates (up to 3 MBaud) than Options 2 and 3.

OPTION 2: USB to Serial Adapter

The second method is to purchase a USB-to-serial adapter and connect it to the DBGU header (CN2) on the SPIKE CPU node. The problem you may have is that not all SPIKE boards have the header soldered onto the board. A header is essentially a 6 pin socket that the adapter can plug into. If you do have the header at location CN2, great! Read on.

Ok, you have a header on the SPIKE board. Simply purchase an inexpensive USB to serial adapter and plug it in. There are lots of them, most for less than $10, and they're all pretty much the same.

Some examples that should work (though we don't guarantee it and we're happy to hear feedback or recommendations):

https://www.amazon.com/FICBOX-CP2102-Serial-Downloader-Arduino/dp/B01CU12324/
https://www.amazon.com/HiLetgo-CP2102-Module-Serial-Converter/dp/B00LODGRV8
https://www.amazon.com/HiLetgo-Ft232rl-Serial-Adapter-Arduino/dp/B00IJXZQ7C
https://www.adafruit.com/products/3309 https://www.sparkfun.com/products/12731
https://www.sparkfun.com/products/13830

Make sure you have a 3.3v adapter (or that your adapter can be set for 3.3v).

Note: If you're using a Raspberry Pi, you can use its built-in serial pins and don't need a USB-to-serial adapter.

Connecting using DBGU

Connect the USB serial adapter to the DBGU header (CN2) on the SPIKE CPU node.

Pins are marked GND, RX, TX. You do not need more than these.

Todo: Add a photo and more detailed pinout instructions (*Help us to write it*).

Unfortunately, this header seems to be missing on some revisions of Spike. You can solder it in though. However, it does not contain any flow-control pins to it will not work at higher baud rates (up to 400k roughly).

OPTION 3: Connect using two USB-Serial Adapters

Newer versions of the SPIKE CPU node do not have a connector attached to the CN2/DBGU header. The newer board is the same, but you see a blank spot instead of the plug-in connector attached. If you do not want to solder a header onto the SPIKE board then you need to go back to Option 1 or use this option. Soldering on the SPIKE board is risky if you lack experience with a solder iron and will likely void your warranty.

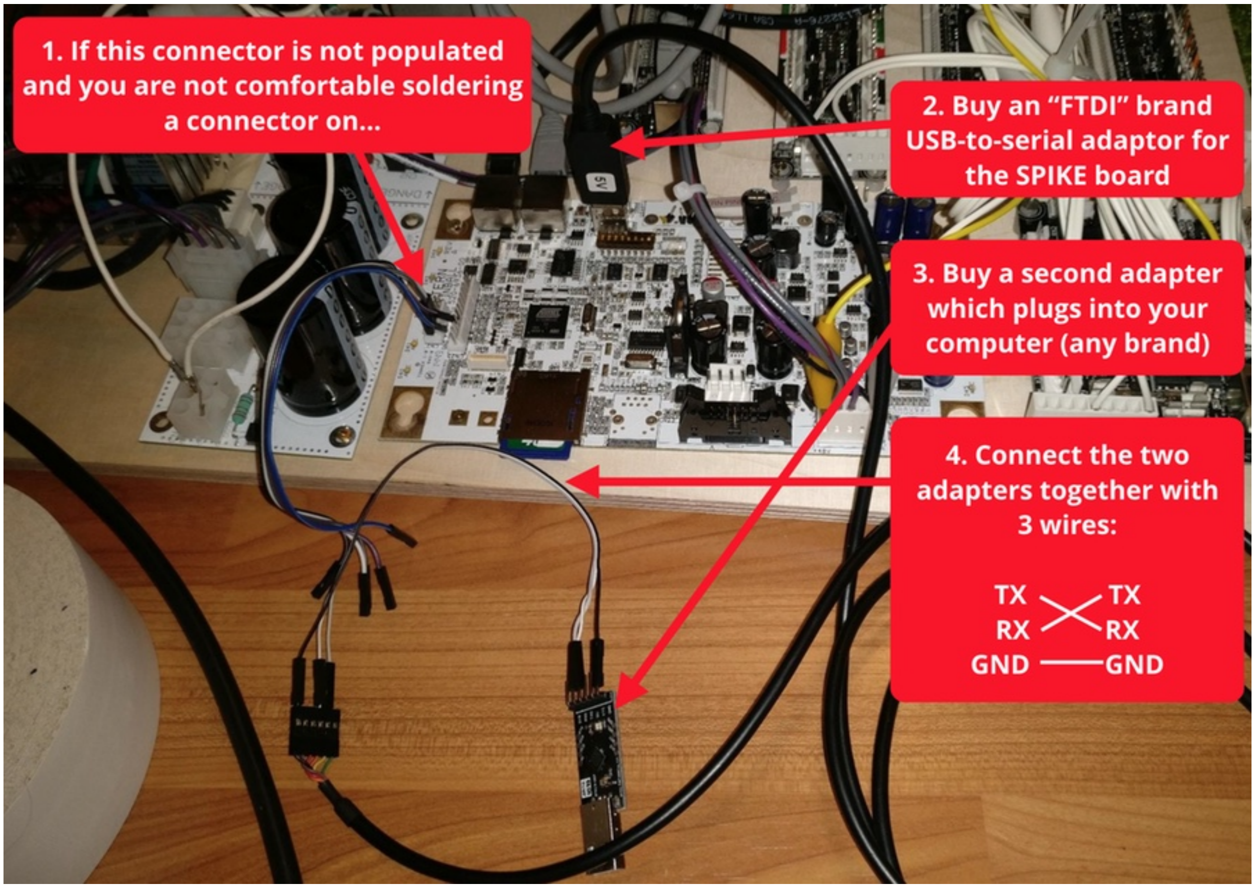
For this option, you can buy two USB serial adapters and then use the USB connection on the SPIKE CPU node.

The one you connect to the SPIKE CPU node needs to have an actual FTDI brand chip because the FTDI drivers are included in the code on the SPIKE board. The second adapter for your computer can be any brand since it's easy to install whatever drivers it needs on your computer. Whatever serial port appears on your computer when you plug in this adapter is the port name you'll use in your machine config.

These two adapters will have connectors or headers on them that you need to connect together. Connect the "RX" (receive) from one to the "TX" (transmit) on the other and vice-versa. Also connect the grounds (possible labeled "GND") together. It's probably a good idea to twist the wires together to reduce interference, especially if your wires are more than a few inches long.

In addition to above you should also "CTS" to "DTS" and "DTS" to "CTS". This will allow you to enable hardware flow control which is essential at higher baud rates (up to 3M).

The following diagram illustrates how everything fits together:



You've essentially created a null modem cable as described in Option 1. This option may be a little cheaper but the solution is far less elegant and stable.

What if it did not work?

Have a look at our *SPIKE troubleshooting guide*.

How to configure MPF for Stern SPIKE hardware

Related Config File Sections
hardware:
spike:

This guide explains how to configure MPF to work with Stern SPIKE pinball machines. It applies to SPIKE and SPIKE 2 systems.

1. Install the drivers for your USB-to-serial adapter

Before you proceed, make sure that you have the drivers properly installed for your USB-to-serial adapter and that when you plug it in, you see the serial port.

2. Configure your hardware platform for SPIKE

To use MPF with a SPIKE hardware, you need to configure your platform as `spike` in your machine-wide config file. You'll also need to add a "spike:" section with some additional settings:

```
hardware:
  platform: spike
spike:
  port: /dev/ttyUSB0
  baud: 115200
  flow_control: false
  debug: false
  nodes: 0, 1, 8, 9, 10, 11
  runtime_baud: 115200
```

Some notes on the settings:

port: Use the port of your USB-serial adapter or of the internal serial on your computer. On Windows, this will have a name like "COM5".

baud: This needs to match the value from Step 3 in the *MPF SPIKE bridge instructions*. It is used to initialise the connection to SPIKE only. Afterwards, the bridge will switch to `runtime_baud`.

flow_control: If your hardware supports flow control and you connected "RTS" and "CTS" in the previous steps set this to True. It will make the connection much more stable at higher speeds. It can be `False` for a first test.

runtime_baud: Note that since only control and switch information is sent across this bus, 115k baud is plenty fast enough if you choose not to use a DMD. However, if you want to use a DMD you need more speed (see below for details).

debug: Set this to true for print more details in the log.

nodes: This is a list of the node board addresses that your system has. You can get this from the manual. Here's an example from Wrestlemania Pro:

Node Address	Description	Location	Part Number
0	SPIKE CPU Node	Backbox	520-6936-00
1	Cabinet Node	Cabinet	520-5319-00
8	Lower Playfield 48V 8-Driver Node	Lower playfield	520-6935-00
8a	Lower Playfield Serial LED Extension	Lower playfield	520-6950-00
9	Mid Playfield 48V 4-Driver Node	Mid playfield	520-5329-00
9a	Trough Serial Opto Receiver Extension	Lower playfield ball trough	520-5345-00
10	Upper Mini Playfield 4-Driver 48V Node	Upper playfield	520-5329-00
11	Mid Playfield I/O Node	Mid playfield	520-5322-00

Only map the node boards and ignore the extension boards because those are transparent to MPF. Just consider 8 and 8a/8b to be the same node.

Once you got your game running you can increase the speed using `runtime_baud`:

```
hardware:
  platform: spike
spike:
  port: /dev/ttyUSB0
  baud: 115200
  runtime_baud: 2000000
  flow_control: true
  debug: false
  nodes: 0, 1, 8, 9, 10, 11
```

This will increase the baudrate after the start of the mpf-spike-bridge. You do not have to change anything to use this setting. The following baudrate are supported:

- 230400
- 460800
- 576000
- 1000000
- 1152000
- 2000000
- 2500000
- 3000000
- 3500000

- 4000000

Depending on your hardware setup they might or might not work. Most setups communicate reliably up to something beween 1Mbaud and 2.5Mbaud. To stream full 30fps to your DMD you need about 2Mbaud. You need flow_control at rates higher than about 0.5MBaud.

What if it did not work?

Have a look at our *SPIKE troubleshooting guide*.

How to configure coils & drivers (Stern SPIKE)

Related Config File Sections
spike:
coils:

To configure coils, drivers, motors, and/or magnets (basically anything connected to an node's driver outputs) for Stern SPIKE machines, you can follow the guides and instructions in the *Coils (Solenoids)* docs.

> **Warning:** Please ensure that you have established *common ground between logic and coil power* before turning on high voltage on your coils (especially on homebrew machines). Ignoring this might lock on your coils, overheat them, burn down your house or kill you. We are serious, floating grounds are dangerous. If you are not an electrical engineer read the *guide about voltages and power*.
>
> In a nutshell: You need to connect your logic ground (5V/12V) and your high voltage ground (48V or 80V). A *power entry or power filter board* is a convenient solution to solve this (and more) issues.
>
> Always turn all PSUs off when connecting power or you might fry all boards at once. This is generally a good idea but even more important when connecting more than one power supply to a board.
>
> IF YOU DID NOT UNDERSTAND WHAT THIS WARNING MEANS STOP NOW AND TRY TO UNDERSTAND IT. OTHERWISE YOUR HARDWARE WILL LIKELY BURST INTO FLAMES AND YOU NEED TO WAIT A FEW DAYS FOR A REPLACEMENT OR EVEN WORSE IT MIGHT KILL YOU. IGNORING THIS IS THE MOST COMMON CAUSE FOR BROKEN DRIVER BOARDS.

However there are a few things to know and some additional options you get with SPIKE hardware that are discussed here.

number:

The `number:` setting for each driver is a combination of the node it's connected to and its address from the manual. For example, here's the driver reference table from Page 11 of the Wrestlemania Pro manual:

DRIVER REFERENCE TABLE

ID	Name	Node	Con-nector	Ret. Pin	Ret. Wire	Volt-age	V+ Pin	V+ Color	Location	Type	Address	Part Number
1	Trough	8	CN8	3	ORG-GRY	48V	4	GRY-ORG	Playfield	Coil - 27-1500	8-DR-5	090-5004-ND
2	Auto Plunger	8	CN6	5	ORG-WHT	48V	6	GRY-ORG	Playfield	Coil - 23-800	8-DR-4	090-5001-ND
3	Center 3-Bank Drop Target Reset	9	CN1	3	YEL-WHT	48V	4	GRY-BLK	Playfield	Coil - 25-1240	9-DR-3	090-5034-ND
7	Ramp Control Gate Right	10	CN11	3	BLU-RED	48V	4	GRY-BRN	Back Panel	Coil - 32-1250	10-DR-3	090-5060-01-ND
8	Shaker Motor	1	CN2	1-2	RED	48V	3-5	BLU	Cabinet	Motor	1-DR-0	041-5029-04
9	Left Pop Bumper	9	CN5	3	ORG-RED	48V	4	GRY-BLK	Playfield	Coil - 26-1200	9-DR-0	090-5044-ND
10	Right Pop Bumper	9	CN7	3	ORG-BRN	48V	4	GRY-BLK	Playfield	Coil - 26-1200	9-DR-1	090-5044-ND
11	Bottom Pop Bumper	9	CN9	3	ORG-BLK	48V	4	GRY-BLK	Playfield	Coil - 26-1200	9-DR-2	090-5044-ND
12	Left Ring Slingshot	10	CN5	3	YEL-BLK	48V	4	GRY-BRN	Mini-PF	Coil - 26-1200	10-DR-0	090-5044-ND
13	Right Ring Slingshot	10	CN9	3	YEL-RED	48V	4	GRY-BRN	Mini-PF	Coil - 26-1200	10-DR-2	090-5044-ND
14	Ring Eject	10	CN7	3	YEL-BRN	48V	4	GRY-BRN	Mini-PF	Coil - 27-1500	10-DR-1	090-5004-ND
15	Left Flipper	8	CN5	3	ORG-YEL	48V	4	GRY-ORG	Playfield	Coil - 24-850	8-DR-0	090-5083-03-ND
16	Right Flipper	8	CN7	3	ORG-GRN	48V	4	GRY-ORG	Playfield	Coil - 24-850	8-DR-1	090-5083-03-ND
17	Left Slingshot	8	CN12	5	ORG-BLU	48V	6	GRY-ORG	Playfield	Coil - 26-1200	8-DR-7	090-5044-ND
18	Right Slingshot	8	CN10	5	ORG-VIO	48V	6	GRY-ORG	Playfield	Coil - 26-1200	8-DR-6	090-5044-ND
20	Left Flipper Hold	8	CN9	3	YEL-ORG	48V	4	GRY-ORG	Playfield	Coil - 31-3500	8-DR-2	090-5083-03-ND
21	Right Flipper Hold	8	CN11	3	YEL-GRN	48V	4	GRY-ORG	Playfield	Coil - 31-3500	8-DR-3	090-5083-03-ND
30	Meter 1	1	CN3	2	BLK	12V	1	RED	Cabinet	Digital Out	1-DR-2	500-9946-00
31	Meter 2	1	CN4	2	BLK	12V	1	RED	Cabinet	Digital Out	1-DR-3	500-9946-00
32	Ticket Dispenser	1	CN11	3		12V	1		Cabinet	Digital Out	1-DR-4	

The address for each driver is in the highlighted column. To enter the number for the driver into MPF, remove the middle "DR" letters so you just have the node number and address number (with a dash between them). For example, the driver for the left flipper coil with the address `8-DR-0` would be entered into the MPF config as `8-0`, etc.

```
coils:
  c_shaker:
    number: 1-10    # Node 1, coil 10
    default_pulse_ms: 100
    allow_enable: true
  c_flipper:
    number: 8-1    # Node 8, coil 1
```

What if it did not work?

Have a look at our *SPIKE troubleshooting guide*.

Related How To guides

- *Coil Resistance and Hardware Details*
- *Wiring Dual Wound Coils*
- *Dual-Wound versus Single-Wound coils*
- *Adjust coil hold power*
- *Adjust coil strength (pulse times)*
- *Recycle / "Cool Down" Time*

- *Details About Flippers*
- *How to configure single-wound flippers*
- *How to configure dual-wound flippers*
- *Flipper end-of-stroke (EOS) switches*

How to configure LEDs & GI (Stern SPIKE)

Related Config File Sections
lights:
spike:

Stern SPIKE machines have replaced all incandescent lights with LEDs. Instead of a lamp matrix, individual LEDs are connected to node boards and can be controlled with 256 levels of brightness.

GI (general illumination) are regular LEDs, and so are flashers, and so are the white backlight LEDs in the backbox. So pretty much everything is an LED.

Many LEDs are single element, single color, with colored insers in front of them. This means that you cannot control the color of the LED, rather, you just control the brightness and the color is what it is.

Most machines also have RGB LEDs that can be set to any color. In those cases the individual red, green, and blue channels each have their own addresses, and then you can group them together into a single, logical RGB LED that you can set to whatever color you want.

Finally, in SPIKE machines, you'll sometimes see several LEDs connected to a single output, meaning that when you set the brightness of that output, you're setting the brightness for all those LEDs.

MPF uses the `lights:` section of the machine config to define LEDs. Most of the settings in the *Lights* documentation apply to LEDs in Stern SPIKE machines, though there are a few SPIKE-specific things to know.

number:

The main thing you need to know about configuring LEDs (besides the fact that you add them to the `lights:` section of your config) is how the hardware numbering works.

Pretty much you just look up the number in the manual for your machine and then enter it without any letters. For example, here is (part of) the lighting chart from Wrestlemania Pro:

3.2 LIGHTING REFERENCE

ID	Name	Node	Node Ext.	Conn.	Ret. Pin	Ret. Wire	Src. Pin	Src. Wire	Location	Type	Light Color	Address	Part Number
1	Start Button	1	-	CN6	3	YEL-BRN	1	REDv	Cabinet	Feature	White	1-LP-2	112-5033-08
2	Tournament Start Button	1	-	CN6	4	YEL-RED	1	RED	Cabinet	Feature	White	1-LP-3	112-5033-08
3	Shoot Again	8	8a	8a-CN1	3	RED-GRY	5	RED	Playfield	Feature	White	8a-LP-47	520-5307-00
4	100 Thousand	8	8a	8a-CN1	2	GRN-GRY	5	RED	Playfield	Feature	White	8a-LP-46	520-5307-00
5	200 Thousand	8	8a	8a-CN1	1	BLU-GRY	5	RED	Playfield	Feature	White	8a-LP-45	520-5307-00
6	300 Thousand	8	8a	CN14	D7	-	-	-	Playfield	Feature	White	8a-LP-20	520-6950-00
7	400 Thousand	8	8a	CN14	D6	-	-	-	Playfield	Feature	White	8a-LP-19	520-6950-00
8	500 Thousand	8	8a	CN14	D5	-	-	-	Playfield	Feature	White	8a-LP-18	520-6950-00
9	600 Thousand	8	8a	CN14	D4	-	-	-	Playfield	Feature	White	8a-LP-17	520-6950-00
10	700 Thousand	8	8a	CN14	D18	-	-	-	Playfield	Feature	White	8a-LP-30	520-6950-00
11	800 Thousand	8	8a	CN14	D17	-	-	-	Playfield	Feature	White	8a-LP-29	520-6950-00
12	900 Thousand	8	8a	CN14	D12	-	-	-	Playfield	Feature	White	8a-LP-25	520-6950-00
13	1 Million	8	8a	CN14	D3	-	-	-	Playfield	Feature	White	8a-LP-16	520-6950-00
14	2 Million	8	8a	CN14	D11	-	-	-	Playfield	Feature	White	8a-LP-24	520-6950-00
15	3 Million	8	8a	CN14	D16	-	-	-	Playfield	Feature	White	8a-LP-28	520-6950-00
16	4 Million	8	8a	CN14	D10	-	-	-	Playfield	Feature	White	8a-LP-23	520-6950-00

Use the address column (highlighted in yellow) to get the numbers for each LED. Remove the "LP" letters, and also remove any lowercase letters (like the "a") from the node. What you're left with is the node address and LED number.

For example, the Shoot Again light with the address `8a-LP-47` would be entered as `number: 8-47`.

```
lights:
  backlight:
    number: 0-0    # 0-0 is the special address for the backlight
  start_button:
    number: 1-2
  tourney_start_button:
    number: 1-3
  shoot_again:
    number: 8-47
```

The backbox backlight Stern SPIKE systems have controllable brightness for the white lights in the backbox that illuminate the translight. All of those LEDs are tied together and controlled as one with the address `0-0`.

GI (General Illumination) GI in Stern SPIKE systems are just regular LEDs. You can tag them with the tag `gi` and then turn them on in the attract mode and/or use them in shows for special effects. Really there's nothing special about them. They're just lights. (Just remember they're controlled and defined as "lights", not as "GIs".)

Flashers Flashers in Stern SPIKE systems are also controlled just like normal lights. They just happen to be super bright, but other than that, use them like any other LED. (Just remember they're controlled and defined as "lights", not as "flashers".)

RGB LEDs

You'll notice in the operator's manual that RGB LEDs are actually three separate LEDs with a separate address for the red, green, and blue channel. Since MPF deals with RGB LEDs as single objects you can set to any color, you need to group the three individual channels of RGB LEDs into single RGB objects.

Here's an example from the Wrestlemania Pro manual:

29	Left Lane Arrow(Red)	11	-	CN2	7	RED-VIO	2	RED	Playfield	Feature	RGB	11-LP-3	520-5333-00
30	Left Lane Arrow(Green)	11	-	CN2	8	GRN-VIO	2	RED	Playfield	Feature	RGB	11-LP-4	520-5333-00
31	Left Lane Arrow(Blue)	11	-	CN2	9	BLU-VIO	2	RED	Playfield	Feature	RGB	11-LP-5	520-5333-00

You could enter the three channels as three separate lights in the `lights:` section of your machine config. However, that would complicate your light shows and lights would not show up nicely in the MPF monitor.

Therefore, you can define a RGB light with multiple channels. What this does is create a new virtual RGB LED which is a grouping of the three LED channels into the RGB LED. Then you can use it like any light.

```
lights:
  left_lane_arrow_rgb:
    channels:
      red:
        number: 1-10
      green:
        number: 1-11
      blue:
        number: 1-12
```

What if it did not work?

Have a look at our *SPIKE troubleshooting guide*.

How to configure DMDs (Stern SPIKE)

Related Config File Sections
spike:
dmds:
displays:

Stern Spike 1 machines support a monochrome DMD. MPF can drive the DMD over serial but you have to make sure that your serial is fast enough to provide sufficient throughput (at least 1.5Mbaud). This can be configured using `runtime_baud` (as described in *How to configure MPF for Stern SPIKE hardware*):

```
hardware:
  platform: spike
spike:
  port: /dev/ttyUSB0
  baud: 115200
  runtime_baud: 2000000       # play with this setting
  nodes: 0, 1, 8, 9, 10, 11
```

Then configure your dmd like in this example:

```
displays:
  window:  # on screen window
    width: 600
```

(continues on next page)

(continued from previous page)

```
    height: 200
  dmd:  # source display for the DMD
    width: 128
    height: 32
    default: true
dmds:
  my_dmd:
    platform: spike
    fps: 30
# some default slides (you don't need those but they are a nice start)
slides:
  window_slide_1:  # slide we'll show in the on-screen window
    - type: display
      width: 512
      height: 128
      effects:
        - type: dmd
  dmd_slide_1:  # slide we'll show on the physical DMD
    - type: text
      text: MPF
      font_size: 30
      color: red
      x: 0
      animations:
        add_to_slide:
          - property: x
            value: 250
            duration: 30
            relative: true
slide_player:
  init_done:
    window_slide_1:
      target: window
    dmd_slide_1:
      target: dmd
```

Note that the *Using a traditional (single color) physical DMD* guide has more details on the window and slide settings used in this machine config.

What if it did not work?

Have a look at our *SPIKE troubleshooting guide*.

How to configure switches (Stern SPIKE)

Related Config File Sections
spike:
switches:

To configure switches on Stern SPIKE machines, you can follow the guides and instructions in the *Switches* docs.

The only special thing to know is how the number works.

number:

The number of a switch on a Stern SPIKE machine is a combination of the address of the node its plugged into, and then its individual ID.

You can find the switch numbers are in the manual. Omit the "SW" and letters for extension boards. Here's an example from Wrestlemania Pro:

3.3 SWITCH REFERENCE

ID	Name	Node	Node Ext	Conn.	Input Pin	Input Wire	GND Pin	Ground Wire	Location	Type	Address	Part Number
1	Left Return Lane	11	-	CN7	2	LGN-YEL	10	BLK-RED	Playfield	Rollover	11-SW-0	500-9935-04
2	Right Return Lane	11	-	CN5	12	PNK-BLK	10	BLK-GRY	Playfield	Rollover	11-SW-8	500-9935-04
3	Left Outlane	11	-	CN7	3	LGN-BLU	10	BLK-RED	Playfield	Rollover	11-SW-1	500-9935-04
4	Right Outlane	11	-	CN5	3	PNK-BRN	10	BLK-GRY	Playfield	Rollover	11-SW-9	500-9935-04
5	Left Slingshot	8	-	CN12	4	GRY-BLU	3	BLK-GRN	Playfield	Leaf	8-SW-7	180-5231-00
6	Right Slingshot	8	-	CN10	4	GRY-VIO	3	BLK-GRN	Playfield	Leaf	8-SW-6	180-5231-00
7	Center 3-Bank Drop Target Right	9	-	CN15	4	WHT-ORG	14	BLK-BLU	Playfield	Opto	9-SW-6	520-5252-13
8	Center 3-Bank Drop Target Center	9	-	CN15	3	WHT-RED	14	BLK-BLU	Playfield	Opto	9-SW-5	520-5252-13
9	Center 3-Bank Drop Target Left	9	-	CN15	2	WHT-BRN	14	BLK-BLU	Playfield	Opto	9-SW-4	520-5252-13
10	Left Flipper Button	8	-	CN9	2	GRY-BRN	1	BLK-GRN	Cabinet	Leaf	8-SW-2	180-5164-01
11	Right Flipper Button	8	-	CN11	2	GRY-RED	1	BLK-GRN	Cabinet	Leaf	8-SW-3	180-5164-01
12	Left Lane	11	-	CN7	5	LGN-GRY	10	BLK-RED	Playfield	Rollover	11-SW-3	500-9935-04
14	Left Orbit	9	-	CN15	9	WHT-GRY	14	BLK-BLU	Playfield	Rollover	9-SW-11	180-5230-01
15	Tournament Start Button	1	-	CN6	9	GRY-WHT	5	BLK-WHT	Cabinet	Micro	1-SW-12	180-5174-00
16	Trough 6	9	9a	CN14	-	-	-	-	Playfield	Opto	9a-SW-17	520-5344-00 tx 520-5345-00 rx
17	Trough 5	9	9a	CN14	-	-	-	-	Playfield	Opto	9a-SW-18	520-5344-00 tx 520-5345-00 rx

This would result in the following switch entries:

```
switches:
  s_left_inlane:
    number: 11-0
  s_right_inlane:
    number: 11-8
  s_left_outlane:
    number: 11-1
  s_right_outlane:
    number: 11-9
  s_left_sling:
    number: 8-7
  s_right_sling:
    number: 8-6
  s_center_drops_right:
    number: 9-6
    type: false
  s_center_drops_middle:
    number: 9-5
    type: false
  s_center_drops_left:
```

(continues on next page)

(continued from previous page)

```
    number: 9-4
    type: false
  s_left_flipper:
    number: 8-2
  s_right_flipper:
    number: 8-3
  s_left_lane:
    number: 11-3
  s_left_orbit:
    number: 9-11
  s_tourney_start:
    number: 1-12
  s_trough_6:
    number: 9-17
    type: false
  s_trough_5:
    number: 9-18
    type: false
```

Note that optos (highlighted in green) need to have the `type: NO` added to them.

What if it did not work?

Have a look at our *SPIKE troubleshooting guide*.

How to configure steppers (Stern SPIKE)

Related Config File Sections
spike:
switches:
steppers:

Node board on Spike support up to four steppers. Steppers connect to light outputs on the board and a homing switch (which may be on another board). We guess that they are hardware-wise similar to the *StepStick* (but that does not matter if you are using an existing machine).

To configure a stepper in Spike you can use the following example:

```
switches:
  s_stepper_home:
    number: 11-4
steppers:
  stepper0:
    number: 10-0
    homing_mode: switch
    homing_switch: s_stepper_home
    platform_settings:
      speed: 20
      light_number: 10-10
    named_positions:
```

(continues on next page)

(continued from previous page)

```
    100: test_1
    200: test_2
    500: test_3
```

This will configure Stepper `0` on node `10`. You can choose a number from `0` to `3`. Which ones does not matter but you can only use every number once.

Then you need to look up the motor reference number in your manual. This is an example from Game of Thrones LE:

3.5 MOTOR REFERENCE

ID	Name	Node	Node Ext.	Conn.	Ret. Pin	Ret. Wire	Src. Pin	Src. Wire	Location	Type	Light Color	Address	Part Number
1	Dragon Motor 1	10	-	CN16	11	BRN YEL	1	RED	Playfield	Stepper		10-LP-10	
2	Dragon Motor 2	10	-	CN16	12	BRN GRN	1	RED	Playfield	Stepper		10-LP-11	

We are interested in `10-LP-10`. This is used as `light_number` above. `10-LP-11` is not used and we guess that Spike automatically uses the next output as well.

The home switch is called `Dragon Home` in GoT and has the number `11-SW-4` according to the manual. We configure is as `s_stepper_home` in this example.

You can change `speed` and `homing_speed` to configure how fast the stepper will move. See *Stepper Motors* for more details about steppers.

What if it did not work?

Have a look at our *SPIKE troubleshooting guide*.

Troubleshooting Spike

If you got problems with your hardware platform we first recommend to read our *troubleshooting guide*. Here are some hardware platform specific steps:

Enable Debugging

If you got problems with your platform try to enable `debug` first. As described in the *general debugging section* of our *troubleshooting guide* this is done by adding `debug: true` to your `spike` config section:

```
spike:
  debug: true
```

This will add a lot more debugging and might slow down MPF a bit. We recommend to disable/remove it after finishing debugging.

Debugging the MPF-Spike Bridge

To debug the bridge you can enable more logging to a USB drive. First open a shell to your serial port (the one connected to your Spike). Stick some USB drive to a USB port on Spike and mount it to /mnt/.

If you USB drive contains a partition use:

```
mount /dev/sda1 /mnt
```

Alternatively use:

```
mount /dev/sda /mnt
```

If you did not get an error your operation succeeded. You can have a look at the content of your stick using:

```
ls /mnt
```

Afterwards, add the following options to your spike config:

```
spike:
  debug: true
  bridge_debug: true
  bridge_debug_log: /mnt/spike.log
```

Now close your shell and start MPF. MPF will instruct the bridge to create a log file on your USB drive with more debug information about the nodebus and other things. This will be helpful to find issues with incorrect commands or responses.

To safely unmount your drive stop MPF, open the console again and type:

```
umount /mnt
sync
```

You can now safely remove the USB drive and download the file on your PC.

Capturing the Bus Traffic of Your Game Using Interceptty

To understand what the game does it is sometimes helpful to capture what it sends and receives on netbus. Unfortunately, we don't know how to enable debugging or verbose mode in the game binary. (Please let us know if you find out.)

Instead, we redirect the serial in Linux and capture the bus this way. Unfortunately, this is not perfect and at least on Spike 1 causes timing issues. Nevertheless, this shows us how things work and also sometimes teaches us how error recovery works in Spike.

Get our interceptty binary and put it on your USB drive. Mount the USB drive as above and run the following command:

Spike 1

```
cd /mnt && chmod +x interceptty-arm
mv /dev/ttyS4 /dev/ttyS4_real; interceptty-arm -s 'ispeed 460800 ospeed 460800' -l /dev/ttyS4_real /dev/
↪ttyS4 > /mnt/serial_dump &
```

(continues on next page)

(continued from previous page)

```
```

Spike 2

```
cd /mnt && chmod +x interceptty-arm
mv /dev/ttymxc1 /dev/ttymxc1_real; interceptty-arm -s 'ispeed 460800 ospeed 460800' -l /dev/ttymxc1_
↪real /dev/ttymxc1 > /mnt/serial_dump &
```

This command should return instantly and run in the background. Now start the game binary in the foreground:

```
/games/game
```

Some versions of some games give you a nice service CLI here. Play the game and make sure you activate all relevant features. Flippers might not work some times. Just try again as this unfortunately sometimes messes up timings.

When you are done after a while stop the game using `ctrl+c`. Then type `fg` to get interceptty in the foreground and stop it using `ctrl+c`.

Restore the serial:

Spike 1

```
mv /dev/ttyS4_real /dev/ttyS4
```

Spike 2

```
mv /dev/ttymxc1_real /dev/ttymxc1
```

Now unmount the USB drive as above and you are done. Please share the capture on the MPF user forum.

Coils Are Not Firing

What to do if your coils are not working?

Check if Your Hardware is Working at all

Sounds stupid but this is a good start: Is the hardware working at all? Do you see switch hits in the logs? If not, check our section `Your hardware is not working at all`.

Check the Watchdog

If switches (or other features of the platform) are working but coils are not we have to dig deeper. Most hardware platforms have some kind of watchdog. Often there is some LED which indicates if the watchdog is received. The MPF log might also contain clues (especially if you have enabled `debug` and run MPF with verbose flags `-v -V`). If the watchdog is not received by your platform it will not enable coils.

In most cases watchdog related problems indicate wiring problems. Check if your boards are properly wired.

Test Your Coil Numbers using MPF Service CLI

Hardware is connected and generally working, watchdog is good but still your coils are not working? Maybe something with the numbering is odd. Lets tests that using the *MPF Service CLI*. Alternatively, you can also use *service mode* if you have already configured it. Both ways work similarly.

To use service cli:

1. Open two consoles
2. Start your game (e.g. using `mpf both`)
3. Start the service cli from within your game folder using `mpf service`.
4. Type `list_coils` and press `ENTER` to see a list of coils.
5. Type `coil_pulse your_coil` and press `ENTER` to pulse it.

Does it work? If not check the log and try verify the coil number. If you do not specify `default_pulse_ms` MPF will use `10ms` which might not be enough for some mechs. Try to increase that gently (maybe `20ms` or `30ms`).

Reducing light update rate

If you got a lot of lights you might run into bus contention issues. You can reduce the light update rate in MPF:

```
mpf:
  default_light_hw_update_hz: 30   # defaults to 50
```

If you set this too low fades will be less smooth but otherwise it should not affect your game.

Your hardware is not working at all

If your hardware is not working at all make sure that you removed the options `-X`, `-x` and `--vpx` from your `mpf both` or `mpf game` command line. Those options will overwrite the settings in your `hardware` section and MPF will not even try to connect to your hardware. If you got config errors we suggest you add `-X` to figure things out without interfacing real hardware all the time. Just keep that option in mind.

Another stupid thing to check: Is your hardware connected to your PC? We know it is stupid but a loose USB connector has happened to most of us.

On Linux you might want to run the command `lsusb` which should show both of your micro controllers connected. You should see two lines similar to

```
Bus 002 Device 014: ID 0483:5740 STMicroelectronics Virtual COM Port
Bus 002 Device 015: ID 0483:5740 STMicroelectronics Virtual COM Port
```

If you are unsure about the output, run the command once with your controllers connected and once without. If there is no difference, then for sure the USB device is not properly connected.

Add debugging to related devices

If you got problems with some switches also add `debug: true` to those as it will give to more insights into the intentions of those devices. Same will work for flippers, coils, lights, servos, steppers and more. See *general debugging section* for details.

Run MPF with verbose flag

See *general debugging section* for details. TLDR: run `mpf both -t -v -V`.

Report Your Issue and Ask For Help

If you cannot find the issue yourself please prepare some information about your issue according to our *troubleshooting guide* and ask in our forum.

Consider Improving the Documentation

Did you solve your issue but found that some relevant information in the documentation is missing or should be linked/located elsewhere? Either tell us in the forum or consider *improving the documentation* yourself to save future users some troubles the same way others saved you some troubles by writing this documentation.

How to configure MPF for Penny K Pinball PKONE hardware

Here's a list of all the How To guides which explain how to use MPF with Penny K Pinball PKONE hardware. These guides include the numbering format (how you map specific entries in your config files to board and connector locations) as well as overall settings that affect how your PKONE hardware performs. (Watch dogs, update speeds, etc.).

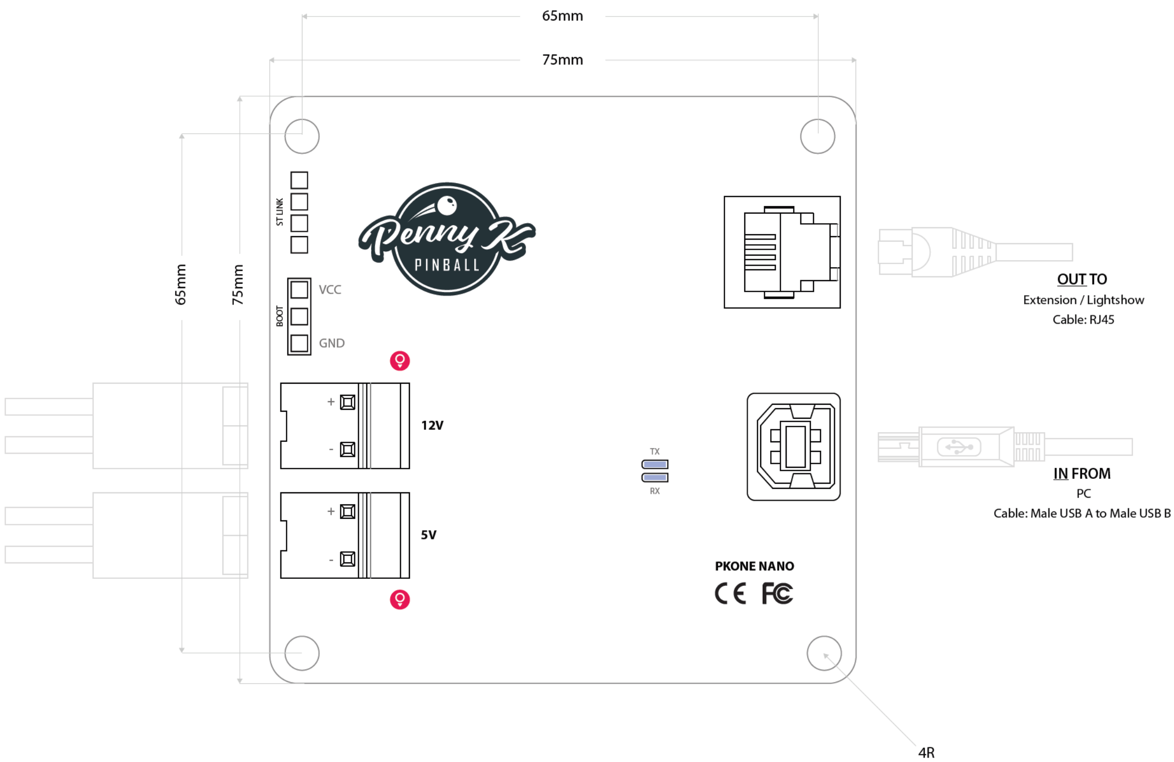
For additional information, please visit the Penny K Pinball website.

Connecting PKONE to your Computer

This page is about connecting the PKONE system to your computer. It roughly covers connecting the bus between the boards.

PKONE Nano

Connect your PKONE NANO controller to your PC using USB.



Then connect the OUT port of your NANO to the IN port of your first board (Extension or Lightshow). Consequently, connect the OUT port of the first board to the IN port of your second board (etc.). Be sure each Extension board or Lightshow board has a unique Address ID set using the Address ID switches on each board. Finally, be sure the last board in the chain has the CANBUS Protocol Termination Jumper set to properly terminate the bus.

Notes:

- Address ID values are numbered starting with zero (Extension boards have addresses 0 to 7 while Lightshow boards have addresses 0 to 3).
- An Extension board cannot have the same Address ID number as a Lightshow board (all connected boards must have unique Address ID values).
- You do not have to chain the boards in the same order as their Address ID numbers.

How to use install drivers & configure COM ports (Penny K Pinball PKONE)

Related Config File Sections
hardware:
pkone:

This guide explains how to configure MPF to work with a Penny K Pinball controller (PKONE NANO) and add-on boards.

1. Install the USB driver

PKONE Pinball controllers use a USB chip from STM. On most operating systems the driver is built-in (Windows 10, Linux, MacOS), so there is no need to download and install the STM driver.

Once this is done, when you plug in and power on your PKONE controller, you should see some kind of notification that new hardware has been detected. What exactly you see will depend on what OS you have.

TODO: Finish this section

2. Configure your hardware platform for PKONE

To use MPF with a PKONE controller system, you need to configure your platform as `pkone` in your machine-wide config file, like this:

```
hardware:
  platform: pkone
```

3. Find the PKONE COM port

Even though the PKONE controllers are USB devices, they use "virtual" COM ports to communicate with the host computer running MPF. On your computer, if you look at your list of ports and then connect and power on your PKONE controller, you should see a new port appear. The exact name and number of this port will vary depending on your computer, what other devices you have, and which port you plug the PKONE controller into.

You need to tell MPF which port is used for the PKONE Controller, and the first step to doing that is to figure out what the port names are on your system:

Finding the COM ports on Windows

On Windows, it's easiest to use the Device Manager. Right-click on the Start button (or whatever it's called now) and choose "Device Manager" from the popup menu.

Then expand the "Ports (COM & LPT)" menu section to see which ports the FAST Controller is using. The easiest way to do this is to open the Device Manager to that section, then plug your PKONE Controller in (or power it on) and just see which port name appears.

The port name will start with "COM" and then be a number.

Finding the COM ports on Max or Linux

On Mac or Linux, it's easiest to find the port numbers via the terminal window (or console window). To do that, open a new window and run the following command:

```
ls /dev/tty.*
```

This will list all the devices whose names begin with "tty".

The PKONE port will have the name that starts with "/dev/cu.usbmodem", then a number. (The number will be different on every system.)

For example, the PKONE port might be something like on MAC:

```
/dev/cu.usbmodem (141)
```

On linux it would look like this:

```
/dev/ttyASM0
```

4. Add the port to your config file

Next you need to add the port to your machine config file. To do this, create a new section called `pkone:`, and then add a `port:` setting under it.

Then if you have a PKONE Nano controller, enter the name of the port.

So an example for Windows might look like this:

```
pkone:
    port: com3
```

And an example for Mac or Linux might look like this:

```
pkone:
   port: /dev/cu.usbmodem
```

Note that if you're using a version of Windows before Windows 10 and you have COM port numbers greater than 9, you will have to enter the port names like this: `\\.\COM10`, `\\.\COM11`, `\\.\COM12`, etc. (It's a Windows thing. Google it for details.)

There are more settings in the *pkone:* section of the machine config that we have not covered here, but the port is the bare minimum you need to get up and running.

What if it did not work?

Have a look at our *PKONE troubleshooting guide*.

How to configure switches (Penny K Pinball PKONE)

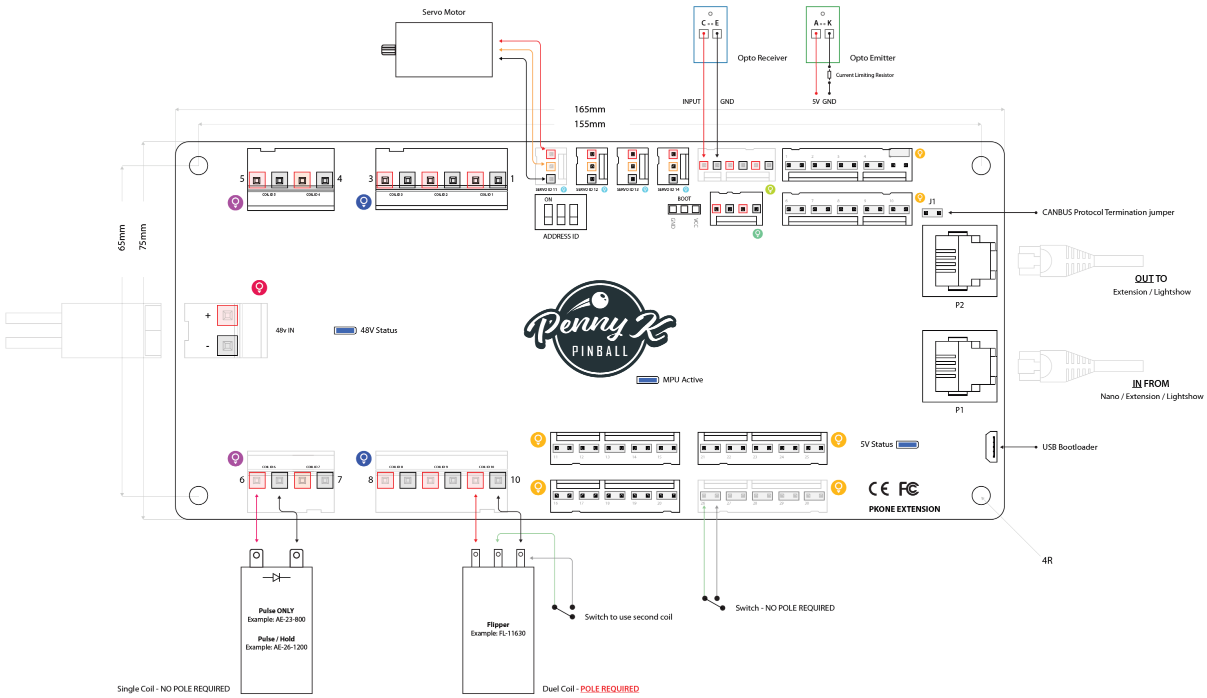
Related Config File Sections
pkone:
switches:

To configure switches with Penny K Pinball PKONE hardware, you can follow the guides and instructions in the *Switches* docs.

However there are a few things to know and some additional options you get with Penny K Pinball PKONE hardware that is discussed here.

number:

When you're using PKONE Extension boards, switches plug into individual Extension boards. Then the Extension boards are connected together in a chain.



The `number:` setting for each switch is its board's Address ID number in the PKONE chain, then the dash, then the switch input number (1-35).

```
switches:
  my_switch:
    number: 0-0    # Extension board at address 0, switch 0
  some_other_switch:
    number: 2-24    # Extension board at address 2, switch 24
```

Notes:

- The PKONE Extension board Address ID switches can be set from 0 to 7.
- Switches 31-35 are setup in the hardware to support optos and other normally closed (NC) switches. Do not list them as NC switches in your configuration as the hardware already inverts the values before sending them to MPF.

What if it did not work?

Have a look at our *PKONE troubleshooting guide*.

How to configure coils/drivers/magnets (Penny K Pinball PKONE)

Related Config File Sections
pkone:
coils:

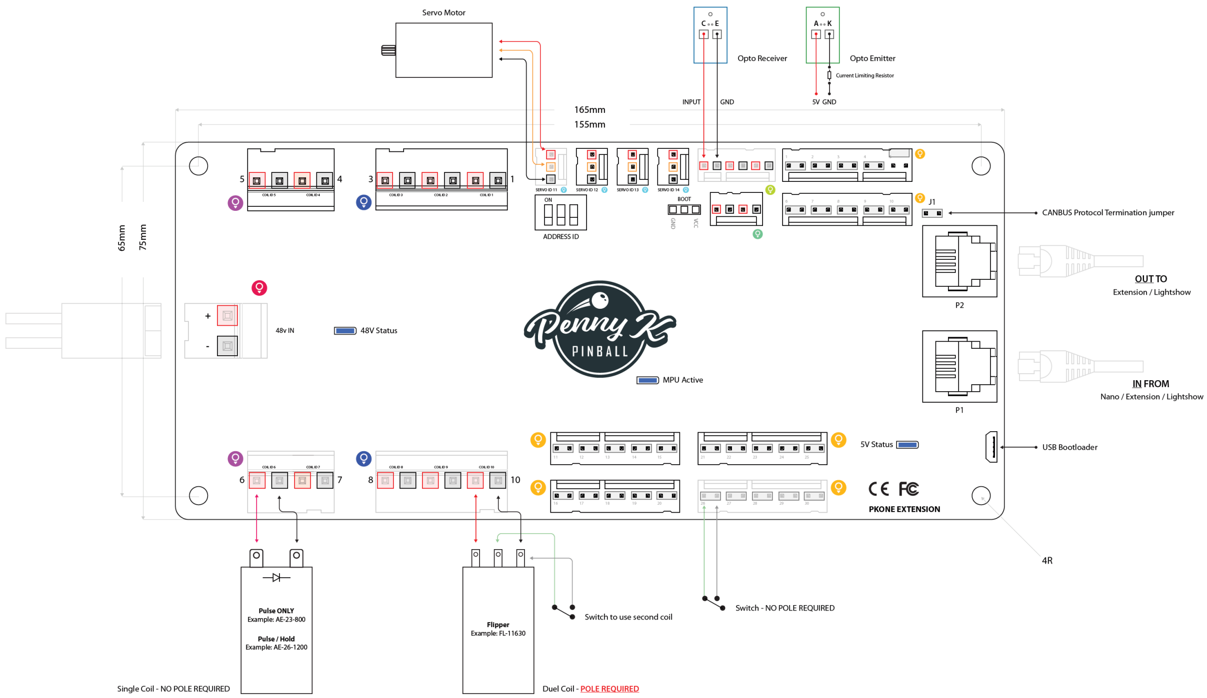
To configure coils, drivers, motors, and/or magnets (basically anything connected to a PKONE Extension board's driver outputs) with Penny K Pinball hardware, you can follow the guides and instructions in the *Coils (Solenoids)* docs.

> **Warning:** Please ensure that you have established *common ground between logic and coil power* before turning on high voltage on your coils (especially on homebrew machines). Ignoring this might lock on your coils, overheat them, burn down your house or kill you. We are serious, floating grounds are dangerous. If you are not an electrical engineer read the *guide about voltages and power*.
>
> In a nutshell: You need to connect your logic ground (5V/12V) and your high voltage ground (48V or 80V). A *power entry or power filter board* is a convenient solution to solve this (and more) issues.
>
> Always turn all PSUs off when connecting power or you might fry all boards at once. This is generally a good idea but even more important when connecting more than one power supply to a board.
>
> IF YOU DID NOT UNDERSTAND WHAT THIS WARNING MEANS STOP NOW AND TRY TO UNDERSTAND IT. OTHERWISE YOUR HARDWARE WILL LIKELY BURST INTO FLAMES AND YOU NEED TO WAIT A FEW DAYS FOR A REPLACEMENT OR EVEN WORSE IT MIGHT KILL YOU. IGNORING THIS IS THE MOST COMMON CAUSE FOR BROKEN DRIVER BOARDS.

There are a few things to know about controlling drivers and coils with PKONE hardware that are discussed here.

number:

When you're using PKONE Extension boards, drivers plug into individual Extension boards. Then the Extension boards are connected together in a chain to the controller.



The `number:` setting for each coil/driver is its board's Address ID number in the PKONE chain, then the dash, then the coil/driver output number (1-10).

```
coils:
  my_coil:
    number: 0-1    # Extension board with Address ID 0, coil/driver 1
  some_other_coil:
    number: 2-10    # Extension board with Address ID 2, coil/driver 10
```

Notes:

- The PKONE Extension board Address ID switches can be set from 0 to 7.

Pulse Power

In the *Coils (Solenoids)* section of the documentation, we talked about *how adjusting a coil's pulse time can affect its strength*. Adjusting the coil's pulse times still assumes that 100% power will be applied to that coil during that pulse time.

Penny K Pinball PKONE controllers allow you to specify the power that's applied to the coil during the initial pulse time. This is similar to the *Adjust coil hold power*, except it applies to the initial pulse time instead of the extended hold time.

You can configure the pulse power by adding a `default_pulse_power:` setting to a coil definition and then specifying the power value from 0-1. (Like default_hold power, 0% to 100%)

For example, consider the following configuration:

```
coils:
  some_coil:
```

(continues on next page)

(continued from previous page)

```
    number: 1-3
    default_pulse_ms: 30
    default_pulse_power: 0.5
```

When MPF sends this coil a pulse command, the coil will be fired for 30ms at 50% power. You can even combine default_pulse_power and default_hold_power, like this:

```
coils:
  some_coil:
    number: 1-3
    default_pulse_ms: 30
    default_pulse_power: 0.5
    default_hold_power: 0.25
```

In this case, if MPF enables this coil, the coil will be fired at 50% power for 30ms, then drop down to 25% power for the remainder of the time that it's on.

Setting Recycle Times

Penny K Pinball controllers allow you to precisely control the *recycle time* for coils or drivers.

A coil's `recycle:` setting is a boolean (True/False), which is set to `False` by default. When using Penny K Pinball hardware, if you set `recycle: true`, then the recycle time is automatically set to twice the coil's `default_pulse_ms:` setting. (e.g. a coil with a `default_pulse_ms: 30` and `recycle: true` will have a 60ms recycle time).

With Penny K Pinball hardware, you can manually set a coil's recycle time by adding a `recycle_ms:` setting, like this:

```
coils:
  slingshot_r:
    number: 1-4
    default_pulse_ms: 30
    platform_settings:
      recycle_ms: 100
```

If you manually specify a recycle_ms value, then that's the value that's used and the coil's `recycle:` (true/false) setting is ignored.

What if it did not work?

Have a look at our *PKONE troubleshooting guide*.

Related How To guides

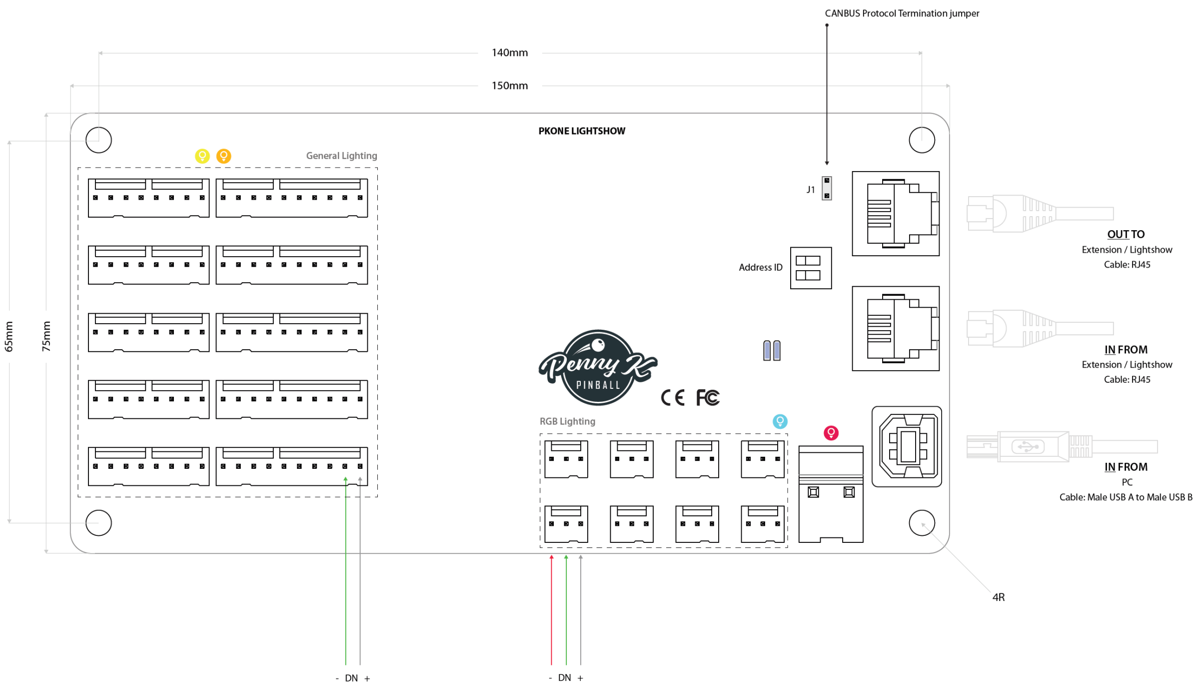
- *Coil Resistance and Hardware Details*
- *Wiring Dual Wound Coils*
- *Dual-Wound versus Single-Wound coils*
- *Adjust coil hold power*

- *Adjust coil strength (pulse times)*
- *Recycle / "Cool Down" Time*
- *Details About Flippers*
- *How to configure single-wound flippers*
- *How to configure dual-wound flippers*
- *Flipper end-of-stroke (EOS) switches*

How to configure WS281XLEDs (Penny K Pinball)

Related Config File Sections
leds:
pkone:

Each PKONE Lightshow add-on board has a built-in 8-group RGB or RGBW LED controller (depending upon which firmware is loaded on the Lightshow) which can drive up to 64 RGB or RGBW LEDs per group (a total of up to 512 LEDs). This controller uses serially-controlled LEDs (where each LED element has a little serial protocol decoder chip in it), allowing you to drive dozens of LEDs from a single data wire. These LEDs are generally known as "WS2812" (or similar). You can buy them from many different companies, and they're what's sold as the "NeoPixel" brand of products from Adafruit. (They have all different shapes and sizes.)



Most of the settings in the *Lights* documentation apply to LEDs connected to PKONE Lightshow boards, however there are a few PKONE-specific things to know.

Overview video about serial LEDs:

https://youtu.be/Q9BG9T7Kj4A

Channel and Number Syntax

In MPF *lights* abstract a light source which emits arbitrary colors. However, this is not true for all real lights. Some support only white (GIs), others only a single-color (i.e. red inserts) and others support full RGB. For that reason MPF knows *light numbers and channel numbers*. Internally, a light consists of one or multiple channels. For instance, a single-color GI will contain a single white channel. While a RGB light will control a red, green and a blue channel. A white light behind a red insert should be a single red channel (because it cannot emit other colors through the red insert). You can configure those channels using the `channels` setting or use `start_channel` and `type` to define the channels. See *Lights* for details.

However, in most cases a platform supports one type of lights (per `subtype`) this would be overly verbose and we added the `number` setting for configuring lights in the common platform way. For instance a platform for GIs will configure single channel white lights or a serial LED controller will configure RGB lights with three channels.

PKONE assumes RGB or RGBW lights by default (depending upon which firmware your Lightshow board is running). For everything else (i.e. RGBW) you have to use channels.

The PKONE Lightshow supports 512 LEDs on eight groups (64 in each group).

Light Numbers

The `number:` setting for each LED is its board's Address ID number in the PKONE chain, a dash, the LED output group number (1-8), another dash, then finally the LED output number in the group chain (1-64) (`address id-group-number`). Internally, PKONE assumes three channels per LED (RGB/GRB) when running RGB firmware and four channels per LED (RGBW) when running RGBW firmware. While assigning numbers manually will work, it is recommended you use the newer chaining syntax referenced below in the Channels section.

Channels

PKONE channels use the format: `address id-group-index`

`address id` and `group` are the same as above and `index` is a an index from 0 to 191 for RGB firmware and 0 to 255 for RGBW firmware. The channel syntax makes it easy to mix LEDs of various types in the same group chain (as long as they are WS281X compatible). The easiest, and recommended, method of numbering is to chain the LEDs in your configuration file and have MPF calculate the internal channel numbers for you (please note the `type` setting is required when using `start_channel/previous` settings):

```
lights:
  led_0:
    start_channel: 0-1-0
    subtype: led
    type: rgb    # will use red: 0-1-0, green: 0-1-1, blue: 0-1-2
  led_1:
    previous: led_0
    subtype: led
    type: rgbw   # will use red: 0-1-3, green: 0-1-4, blue: 0-1-5, white: 0-1-6
  led_2:
    previous: led_1
```

(continues on next page)

(continued from previous page)

```
    subtype: led
    type: rgbw   # will use red: 0-1-7, green: 0-1-8, blue: 0-1-9, white: 0-1-10
```

This method of chaining your LEDs works exactly the same way whether your Lightshow board is running RGB or RGBW firmware.

See *WS2811 and WS2812 LEDs in Pinball* for additional details.

Color Correction

If you are using RGB LEDs, they might not be perfectly white when you turn them on. They might be pinkish or blueish instead depending on the brand of the LED. To a certain extend this is normal/expected and you can compensate for it by configuring *color_correction profiles in light_settings*.

subtype:

Single value, type: `string`. Defaults to empty.

This value is used to distinguish between simple LEDs and WS281X RGB LEDs in the PKONE hardware system. This value must be set to `led` or left empty when setting up WS281X RGB/RGBW LEDs.

What if it did not work?

Have a look at our *PKONE troubleshooting guide*.

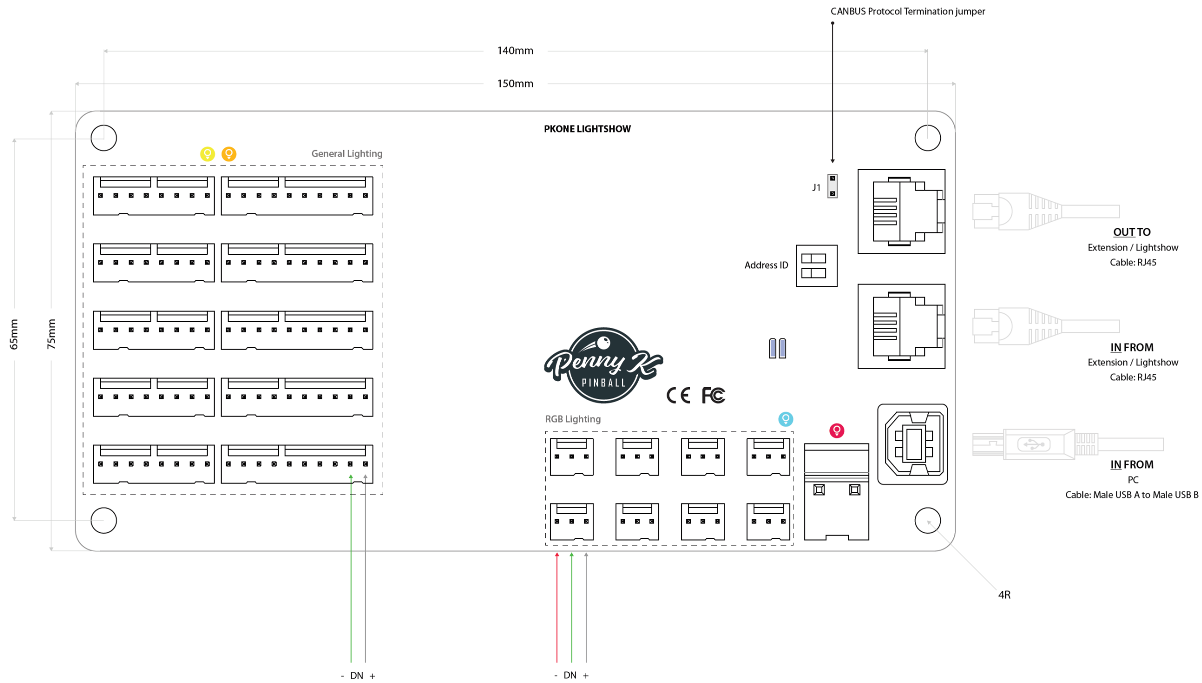
How to configure Simple LEDs (Penny K Pinball)

Related Config File Sections
lights:

Up to 45 Simple LED lights are supported on Penny K Pinball PKONE Lightshow boards. Simple LED lights are single channel monochromatic LEDs most frequently used under colored inserts in the playfield or behind colored artwork behind a backglass.

number:

When you're using PKONE Lightshow boards, simple LEDs plug into individual Lightshow boards. Then the Lightshow boards are connected together in a chain with other add-on boards (such as PKONE Extension boards) to the controller.



The `number:` setting for each simple LED is its board's Address ID number in the PKONE chain, then the dash, then the simple LED output number.

```
lights:
  special_light:
    number: 0-1    # Lightshow board with Address ID 0, simple LED 1
    subtype: simple
  some_other_light:
    number: 2-10    # Lightshow board with Address ID 2, simple LED 10
    subtype: simple
```

Notes:

- The PKONE Lightshow board Address ID switches can be set from 0 to 3.

subtype:

Single value, type: `string`. Defaults to empty.

This value is used to distinguish between simple LEDs and WS281X RGB LEDs in the PKONE hardware system. This value must be set to `simple` when setting up simple LEDs (WS281X RGB LEDs use `led` as the `subtype:` value).

What if it did not work?

Have a look at our *PKONE troubleshooting guide*.

How to configure servos (Penny K Pinball PKONE)

Related Config File Sections
servos:

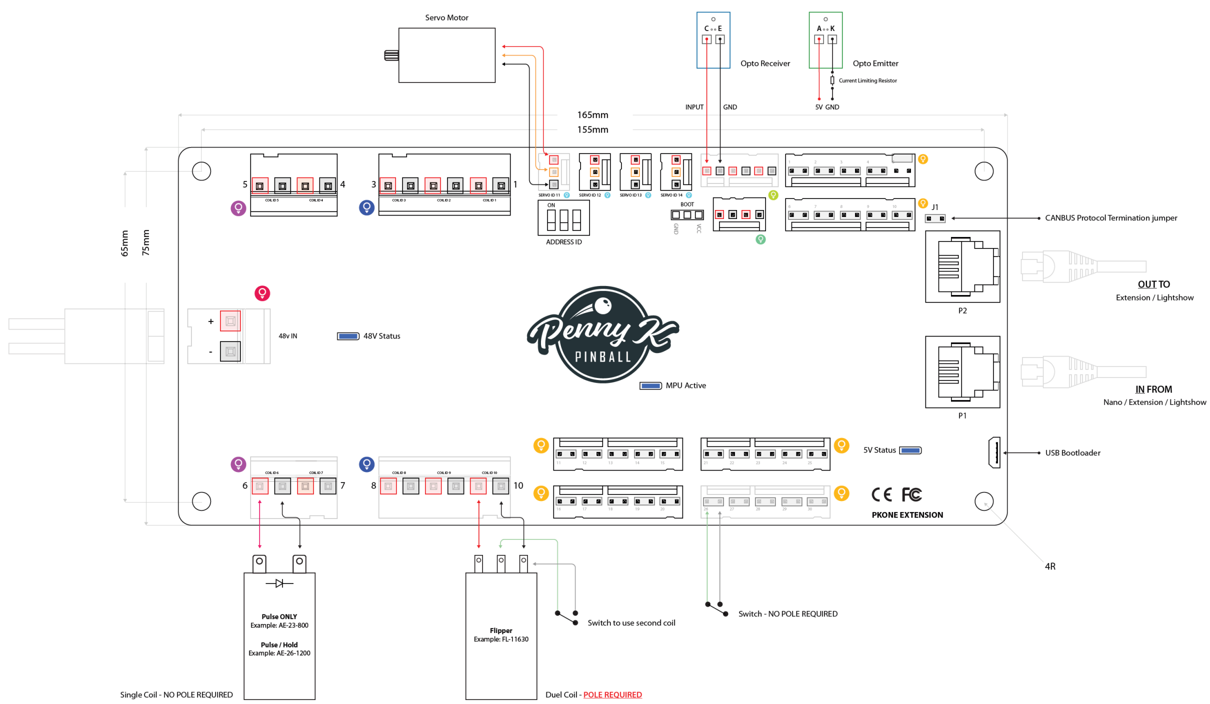
You can drive up to four servos from any PKONE Extension board.

Overview video about *servos*:

https://youtu.be/wA6KEODwQ5w

number:

When you're using PKONE Extension boards, coils plug into individual Extension boards. Then the Extension boards are connected together in a chain to the controller.



The `number:` setting for each servo is its board's Address ID number in the PKONE chain, then the dash, then the servo output number (11-14).

```
servos:
  servo_1:
    number: 0-11    # Extension board with Address ID 0, servo 11 (the first one)
  some_other_servo:
    number: 2-14    # Extension board with Address ID 2, servo 14
```

Notes:

- The PKONE Extension board Address ID switches can be set from 0 to 7.
- Servos are numbered from 11 to 14 on the PKONE Extension board and not from 1 to 4.

All the servo config options are explained in-depth in the *servos: section* of the config file reference.

What if it did not work?

Have a look at our *PKONE troubleshooting guide*.

Troubleshooting Penny K Pinball PKONE Hardware

If you got problems with your hardware platform we first recommend to read our *troubleshooting guide*. Here are some hardware platform specific steps:

Run Hardware Scan

Using `mpf hardware scan` you can find out if your PKONE boards are talking properly to MPF using USB:

```
$ mpf hardware scan

Penny K Pinball Hardware
------------------------

- Connected Controllers:
  -> PKONE Nano - Port: com3 at 115200 baud (firmware v1.1, hardware rev 2)

- Extension boards:
  -> Address ID: 0 (firmware v1.1, hardware rev 2)
  -> Address ID: 1 (firmware v1.1, hardware rev 2)

- Lightshow boards:
  -> Address ID: 2 (RGB firmware v1.0, hardware rev 1)
  -> Address ID: 3 (RGBW firmware v1.0, hardware rev 1)
```

See *mpf hardware (command-line utility)* for details.

Enable Debugging

If you got problems with your platform try to enable debug first. As described in the *general debugging section* of our *troubleshooting guide* this is done by adding `debug: true` to your opp config section:

```
pkone:
  debug: true
```

This will add a lot more debugging and might slow down MPF a bit. We recommend to disable/remove it after finishing debugging.

Snux System 11 Driver Board

Related Config File Sections
hardware:
snux:
system11:
switches:
coils:

MPF can be used with Williams System 11 machines. (Also since Data East's system was a clone of Williams System 11, everything here also applies to those machines.) This How To guide walks you through the process of buying the hardware you need and configuring MPF to work with it.

(A) Understand the challenges of System 11 hardware

The original System 11 Williams/Bally hardware (and the Data East clone) was created in a time when computing resources were scarce and hardware was expensive. It's sort of a "crossover" between the early solid state machines of the '80s and the more modern WPC machines. Because of this, there are a lot of, umm. . . "quirks" to the design which were necessary at the time but which may seem a bit strange in today's world. Even though we tend to lump all "System 11" machines into a single category, there were actually four different generations of System 11 machines, called System 11, System 11A, System 11B, and System 11C. (And just to make things even more fun, some changes were made part way through System 11B.) So technically-speaking there are actually five different types of System 11 machines out there!

Flippers

On modern WPC pinball machines, flipper buttons are just regular switches that send their inputs to the CPU, and flipper coils are just regular coils that are controlled by the CPU. Typical flippers in MPF are configured via the flippers: section of the config file, and when flippers are enabled, hardware rules are written to the pinball controller to allow them to be fired "instantly" when the flipper buttons are hit. Back in the days of System 11, the CPUs in those machines didn't have enough horsepower to constantly poll the status of the flipper buttons and to drive the flippers in software while also doing everything else the CPU needed to do to run the game. So instant, System 11 machines had the flipper buttons directly connected to the flipper coils, meaning that hitting the flipper button would activate the flipper coil directly without any intervention of the CPU. Of course the machine still needed a way to enable or disable the flippers, since the flippers needed to be disabled when a game was not going on and when the player tilted. To do this, System 11 machines used a "flipper enable" relay. This was a mechanical relay connected to a driver output on the driver board. When that driver was enabled, the relay was energized and the flippers worked. When that relay was disabled, the relay de-energized, the electrical connection to the flipper buttons was broken, and the flippers stopped working. While this meant that the CPU didn't have to directly control the flippers, it also meant that many modern conveniences are not available on that hardware. For example, on modern machines you can control the strength of the flipper by adjusting the pulse times of the flipper coils with millisecond-level accuracy. But these older machines gave full power to the flipper until the flipper bat hit the end-of-stroke (EOS) switch, and that switch mechanically cut off power to the high-power winding (while keeping power enabled on the low-power hold winding). So in those days, changing the strength of a flipper was done by physically swapping out the flipper coil with a stronger or weaker one.

"Special" Solenoids

Flippers are not the only types of devices that require instant response in pinball machines. They also need instant response action for slingshots, pop bumpers, and (sometimes) diverters. In many System 11 machines, these types of devices were also controlled by the flipper enable relay. So when that relay was enabled, it enabled not just the flippers but also the pop bumpers and slingshots. Of course pop bumpers and slingshots are a bit different than flippers:

- The CPU needs to know when pop bumpers and slingshots are hit so it can assign points, flash lights, play sounds, etc.
- The CPU needs to be able to manually fire pop bumpers and slingshots for things like ball search and the coil test options in the operators menu.

In other words, it seems that pop bumpers and slingshots really need to be controlled the "new" way since the CPU needs to know when they're hit and the CPU needs to be able to manually fire them. But of course firing a pop bumper or slingshot when their switch is hit needs to happen instantly, and as we just discussed, that was not possible in the System 11 days. So how did they get around it? System 11 machines call these types of solenoids *special solenoids* (that is literally what they're called in the manual) because they're actually controllable via two different ways:

- When the flipper enable relay is enabled, a hit to these devices' switches creates a direct electrical connection to their coils which fires them.
- These devices' coils also have a second (additional) control input which lets the CPU fire them from the service test menu or for ball search.

Furthermore you'll also notice that there are switches in the switch matrix for many of these devices which are used to let the CPU know that these devices have been hit to assign points and to do effects. At this point you might think, "Great! So these devices have CPU- controlled coils, and they have switches in the switch matrix, so I can just set them up like regular devices since I'm using modern hardware!" Not so fast. In many System 11 machines, the switches in the switch matrix which tell the CPU that a pop bumper or slingshot has been hit are not the same switches that fire the coil! For example, the switch attached to the skirt of the pop bumper that the ball hits is a high-voltage switch that is physically connected to the pop bumper's coil. The CPU does not see that switch at all. When that switch is hit (if the flipper enable relay is active), then it grounds the connection to the coil and the coil fires. When the coil fires, its shaft hits a second switch underneath, and that's the switch that is connected to the switch matrix and the CPU. (And actually there's a third switch under there too which is the EOS switch which cuts power to the coil after it's been fired.) So in reality, yeah, you may see a switch in the switch matrix for a pop bumper, but that switch is not, "Hey the pop bumper skirt switch was hit, so fire the pop bumper now," rather, that switch is, "Hey the pop bumper just fired. Just FYI." The exact details of how these special solenoids work depends on the specific machine and which version of System 11 it is. For example, some devices (like pop bumpers and slingshots) should always be on whenever the flippers are enabled, so the flipper enable relay enables them too. Other devices (like diverters) should only be active sometimes, so they have their own enable driver (which is like the flipper enable relay, but separate from it) so they can be controlled individually.

The A/C Relay & Switched Solenoids

But wait! There's more! System 11 machines also have this concept of the A/C relay. This is not A/C in the terms of alternating current. It has nothing to do with that. It's actually used to control things called A-side and C-side devices. The basic concept is that since the driver circuitry was expensive, Williams decided they could get double their "bang for their buck" by connecting two devices so a single output. So you might see on a schematic that a single driver output is connected to both a ball kickout coil and a flasher. Then there was a relay (called the A/C relay, or sometimes the C-select relay) connected in there too. If the A/C relay was in the A position, then firing that driver would fire the coil connected to the A side of that output, and if the A/C relay was in the C position, then firing that driver would fire the device connected to the C side of that output. This worked because they had a single A/C relay that was connected to an entire bank of 8 drivers. So they could actually control 16 different devices (8 drivers with two devices each) from just 9 driver outputs (8 drivers plus 1 for the A/C relay). They were also smart about what types of devices they connected to each side of the relay. System 11 machines put the "important" devices on the A side (things that interact with the ball on the playfield, like diverters, kickout holes, motors, etc.), and they put the "less important" things on the C side (flashers and the knocker coil). So this means they will constantly enable and disable the A/C relay to do different effects, but if two things need to happen at exactly the same time, they can service the A-side first (since those are the important ones) and then flip the relay to the C-side and pick those up after a few hundred milliseconds of delay.

Controlled Solenoids

In addition to switched, controlled, and flipper solenoids, System 11 machines also included what they called "controlled" solenoids which was their name for normal, modern-style solenoids. So in addition to all the craziness of the other control schemes, some solenoids were regular. No special switches. No special handling. Just regular solenoids.

GI (General Illumination)

In WPC machines, GI strings are controlled via separate GI drivers (which are alternating current and which may or may not be dimmable). In System 11, GI strings were regular driver outputs, just like any solenoid. The catch is that most (maybe all?) GI strings on System 11 machines are "backwards" in the sense that the GI is on when the driver is disabled, and you enable the driver to turn off the GI. This was done because the GI is almost always on all the time, though there are periods when you might want to turn it off for special effects. So to save on wear of the relays and make things simpler, in System 11 machines, the GI is just always on until the CPU turns it off.

Putting it all together

If you look at the solenoid table in the operators manual of a System 11 machine, you'll see that all the drivers fall into these categories. Some are are switched, some are controlled, some are flippers, and some are special. Check out the solenoid table from PinBot. Note that the first 16 solenoids are the A/C switched solenoids, and there are two coils for each number 1-8 with an "A" and "C" suffix denoting which side they're on. Then the next 8 (numbers 9-16) are controlled solenoids. These are the regular modern-style drivers which also include the GI (remember they're active off) and important flashers they don't want to share with A/C switched drivers. Then you have the next batch 17-22 which are the special solenoids that are enabled when the flipper enable relay is enabled, but they can also be manually controlled for ball search and testing. And finally you have the left and right flipper solenoids which don't have numbers because they're not connected to the driver board. Also notice solenoid 14 is the "Solenoid Select Relay." That's the A/C select which when inactive means that drivers 1-8 are connected to the A-side devices, and when active means drivers 1-8 are connected to the C-side devices.

(B) The Snux board

Okay, so now that you're caught up with the intricacies of System 11 hardware, how do you actually control this via MPF? The usual way you control an existing machine is to remove the original CPU board and to replace it with either a P-ROC controller. The new pinball controller plugs into the backbox and uses the existing driver board. The problem with System 11 is that unlike more modern machines, the System 11 CPU board and driver board were actually combined into one single huge board. So when you take out the CPU board, you also lose the driver board. This means if you put a P-ROC controller into a System 11 machine, you don't have a driver board. :(This is where the Snux board comes in. The Snux board (which is our name for it) is a System 11 driver board created by Mark Sunnucks. (His online handle is Snux which is why we call it the Snux board.) Mark developed this board a few years ago because he wanted to control an F-14 machine with a P-ROC. The Snux board can be thought of kind of like the WPC power driver board except that it's made to work with System 11 machines instead of WPC machines. Since the original System 11 combo CPU board / driver board was so huge, when you remove it from your System 11 machine there's plenty of room to put the Snux board and a P-ROC controller in it's place. The Snux board connects to the P-ROC controller via the standard 34-pin ribbon cable, and then it has all the connectors (in their proper locations) to connect the existing wiring connectors from the System 11 machine to it. So in order to control a System 11 machine with MPF, you need to get a Snux board. Mark has a day job and built this board as a hobby, but he sells them to other folks who are interested in modernizing System 11 machines. Mark lives in the UK, so the exact price you pay depends on the exchange rate, shipping to your country but it's around $180 US (Then you also have to buy a P-ROC to drive it.) You can contact Mark via PM (on Pinside as Snux). In addition to the board there are 3 or 4 cables you'll need, Mark can advise.

Displays

All System 11 machines used various combinations of segment displays and these cannot be directly controlled via the P-ROC. If you do want to use the original segment displays, Jim at mypinballs.com sells an adaptor board that will connect between the P-ROC and the displays. Otherwise you can use the various other display options that MPF provides.

(C) Understand how MPF works with the Snux board

Once you have your P-ROC controller and the Snux board installed in your System 11 machine, you need to build your machine- wide configuration file for your machine. MPF has a *Snux* interface which is actually implemented as a platform overlay. A platform overlay, in MPF, is like a second layer that sits on top of the regular platform interface and modifies the way it works. So since the Snux board works with the P-ROC controller, the main platform interface MPF uses is the P-ROC platform. Then the Snux platform overlay layers on top of it to handle the special cases that arise when using the P-ROC with a Snux board. (For example, automatically controlling the A/C relay to make sure it's in the right position when an A-side or C-side driver is activated, and preventing the activation of C-side drivers when the A/C relay is in the A position and vice-versa.) The Snux driver overlay completely hides the nuances of the System 11 hardware from you. You can freely enable, disable, or pulse any A-side or C-side driver you want, and MPF will automatically control the A/C relay and make sure it's in the proper position. Since A-side drivers are more important in the machine, MPF will always give them priority. If simultaneous requests for an A-side and C-side driver come in at the same time, MPF will service the A-side driver and add the C-side driver to a queue, and then when the A-side driver is done, MPF will flip the relay to the C-side and then service the C-side driver. Similarly if drivers on the C-side are active and an A-side request comes in, MPF will deactivate the C-side drivers, flip the relay, and then service the A-side drivers. The takeaways from this are (1) A-side drivers always have priority, and (2) the handling of the A/C relay is automatic.

(D) System 11-specific MPF configuration

Once you have your hardware setup, there are a few things you need to do in your config file.

1. Configure your hardware interface

The first thing to do is to configure your hardware options in the hardware section of your machine-wide config. You configure the main platform as *p_roc*, but then for *driverboards* you configure it as *snux*, like this:

```
hardware:
  platform: virtual
  driverboards: wpc
  coils: snux
  switches: snux
```

2. Configure snux options

The MPF machine-wide config file contains a few options for the Snux driverboard. These options are set in the default *mpfconfig.yaml* file which means you don't have to add them to your own config file, but we're including them here just for completeness:

```
coils:
  c_diag_led_driver:
    number: c24
    default_hold_power: 1.0

snux:
  diag_led_driver: c_diag_led_driver
```

The Snux board maps driver `c_diag_led_driver` which is driver 24 to the "diag" LED on the board. When you power on your machine, the diag LED is off. Then when MPF connects to the board, this LED turns on solid. Finally when MPF is done loading and it starts the main machine loop, this LED flashes twice per second. If this LED stops flashing, that means MPF crashed. :)

3. Configure system11 options

Next you need to add a system11: section to your machine-wide config and specific some System 11 options. At this point you might be wondering, "Why aren't these options in the snux section?" The reason is that the settings in the snux section apply to the Snux board itself, whereas the settings in this system11 section apply to any System 11 machine that MPF might control. Of course at this point, that's only possible via the Snux board, but they're technically separate settings since the architecture allows for future System 11 boards that may exist at some point. Here's the system11 configuration section from Pin*Bot:

```
system11:
  ac_relay_delay_ms: 75
  ac_relay_driver_number: c14
```

The *ac_relay_delay_ms* is the number of milliseconds MPF waits before and after flipping the A/C select relay to allow for it to fully switch positions. For example, if you have a C-side driver active and you need to activate an A-side driver, MPF cannot simply deactivate the A/C relay and the C-side device and activate the A-side device all at the same time. If it does then power will "leak" from one side to the other as the relay is transitioning. So what actually happens in this scenario is that MPF will deactivate the C-side devices, then wait 75ms for them to really be "off", then deactivate the A/C relay, then wait another 75ms for the relay to flip, then activate the A-side device. We did some experimentation with different delay times. On Pin*Bot, 50ms was definitely too short as we'd see some weak flashes from C-side flashers connected to A-side devices we were activating on the transition. 75ms seems fine, though really this is all faster than humans can perceive (and C-side devices aren't as time sensitive), so even setting this to 100ms is probably fine. 75ms is the default if you don't add this section to your config. The *ac_relay_driver_number* is the driver (with a "C" added to it) from the manual for the A/C select relay. Be sure you check the A/C relay driver number from your manual. It's different in the two System 11 machines we tested. (C14 in *Pin*Bot* and C12 in *Jokerz!*) Also it's labeled differently in different manuals. In the *Jokerz!* manual it's called the "A/C Select Relay," and in the *Pin*Bot* manual it's called the "Solenoid Select Relay."

4. Enable flippers

The Snux board uses driver 23 to enable the flippers:

```
digital_outputs:
  flipper_enable_relay:
    number: c23
    type: driver
    enable_events: ball_started
    disable_events: ball_will_end
```

You can change the events when the flipper should enable and disable. By default we will enable the flippers on ball start and disable them on ball end.

5. Configuring driver numbers

> **Warning:** Please ensure that you have established *common ground between logic and coil power* before turning on high voltage on your coils (especially on homebrew machines). Ignoring this might lock on your coils, overheat them, burn down your house or kill you. We are serious, floating grounds are dangerous. If you are not an electrical engineer read the *guide about voltages and power*.
>
> In a nutshell: You need to connect your logic ground (5V/12V) and your high voltage ground (48V or 80V). A *power entry or power filter board* is a convenient solution to solve this (and more) issues.
>
> Always turn all PSUs off when connecting power or you might fry all boards at once. This is generally a good idea but even more important when connecting more than one power supply to a board.

IF YOU DID NOT UNDERSTAND WHAT THIS WARNING MEANS STOP NOW AND TRY TO UNDERSTAND IT. OTHERWISE YOUR HARDWARE WILL LIKELY BURST INTO FLAMES AND YOU NEED TO WAIT A FEW DAYS FOR A REPLACEMENT OR EVEN WORSE IT MIGHT KILL YOU. IGNORING THIS IS THE MOST COMMON CAUSE FOR BROKEN DRIVER BOARDS.

When you configure coils, flashers, and gis in your MPF hardware config, you can enter the numbers straight out of the operators manual. The only thing to note here is that you must add a "C" to the beginning of the driver number (even for flashers and GI), since that's what triggers MPF to do a WPC-style lookup to convert the driver number to the internal hardware number the platform uses. (It's an WPC-style lookup since the Snux driver board emulates a WPC driver board.) Also for switched solenoids which use the A/C relay, you also need to add an "A" or a "C" to the end of the driver number. Here's a snippet (incomplete) from the *Pin*Bot* machine-wide config file:

```
coils:
  outhole:
    number: c01a
  knocker:
    number: c01c
  trough:
    number: c02a
  visor_motor:
    number: c13
    allow_enable: true
  upper_pf_and_topper_1:
    number: c02c
  left_insert_bottom:
    number: c03c
  right_insert_bottom:
    number: c04c
  lower_pf_and_topper_2:
    number: c05c
  energy:
    number: c06c
  left_playfield:
    number: c07c
  sun:
    number: c08c
  robot_face_insert_bottom:
    number: c09
  topper_3:
    number: c15
  topper_4:
    number: c16
```

Again, don't forgot the "a" or the "c" at the end of the switched solenoids, since that's how MPF knows it needs to use the A/C relay logic for those devices!

6. Configure lamps

Configuring the numbers for matrix lamps is pretty straightforward and something you can also use the manual for. The format for lamp number is the letter "L" followed by the column, then the row. In other words, light number L25 is the light in column 2, row 5. This is a bit confusing because these are not the numbers that the lamps use in the manual! The lights in the lamp matrix table are simply numbered from 1 to 64. So you need to use the chart in the manual to get the column and row positions, not to get the actual light numbers! (When Williams switched to WPC, they switched to lamp numbers based on the column and row. So in WPC machines, the lamps in column 1 are numbers 11-18, the lamps in column 2 are 21-28, etc. System 11 numbers would be 1-8 for column 1, 9-16 for column 2, etc. Basically since System 11 machines have an 8x8 lamp matrix, there should be no numbers 9 or 0 anywhere in your lamp numbers. Here's a snippet of the configuration from Pin*Bot:

```
lights:
  game_over_backbox:
    number: L11
  match_backbox:
    number: L12
  bip_backbox:
    number: L13
  mouth1_backbox:
    number: L14
  mouth2_backbox:
    number: L15
  mouth3_backbox:
    number: L16
  mouth4_backbox:
    number: L17
  mouth5_backbox:
    number: L18
  bonus_2x:
    number: L21
  bonus_3x:
    number: L22
```

Again, don't forget that they should all start with "L", and they're based on the positions in the matrix, not on the numbers from the manual.

7. Configure switches

Switch numbering in System 11 machines is the same as lamp numbering, except the numbers start with "S". Again the numeric portion of the number is based on the column/row, not the switch number in the manual. So even though the manual says that the switch in column 5, row 6 is number 38, you actually enter "L56". Here's another snippet from *Pin*Bot*:

```
switches:
  left_outlane:
    number: S24
    label: Left Outlane
    tags: playfield_active
  left_inlane:
    number: S25
    label: Left Inlane
    tags: playfield_active
  right_inlane:
    number: S26
    label: Right Inlane
    tags: playfield_active
```

(continues on next page)

(continued from previous page)

```
  right_outlane:
    number: S27
    label: Right Outlane
    tags: playfield_active
```

You might have to do some detective work to figure out where the switches are and how they work. For example, remember that switches from slingshots or pop bumpers are most likely activated by the physical action of the device's coil, not by the switch above the playfield. So on Pin*Bot hitting the pop bumper skirt does not activate the pop bumper switch, but manually pushing the pop bumper ring down with your fingers will activate that switch. Also you might see switches with names along the lines of "Right Lane Change." If the lane change in that machine is activated by a slingshot, then most likely the Right Lane Change switch is under the playfield and activated by the physical slingshot arm hitting it. Same for flipper- controlled lane changes. You'll have to hunt to see whether there's a second switch in the flipper EOS stack under the playfield or perhaps a second switch in the stack behind the flipper button.

8. Create your System 11-style trough

Troughs in System 11 machines are not like troughs in modern machines. Rather than a single ball device which acts as the drain as well as the feeder to the plunger lane, System 11 machines have two separate devices with two solenoids. One device is typically called the "outhole" (or "drain") which receives the ball from the playfield, and it kicks the ball over to the trough where the ball is stored. Then the trough has a second coil which kicks the ball into the plunger lane when it needs it. We have a separate How To guide which details how to setup a System 11 1980s- style trough, link below (since many games do this, even ones that aren't System 11), so you can read that for more details. The result though will look something like this:

```
ball_devices:
  outhole:
    ball_switches: outhole
    eject_coil: outhole
    confirm_eject_type: target
    eject_targets: trough
    tags: drain
  trough:
    ball_switches: trough1, trough2
    eject_coil: trough
    eject_targets: plunger_lane
    tags: home, trough
  plunger_lane:
    ball_switches: plunger_lane
    mechanical_eject: true
    eject_timeouts: 3s
```

The key is that you're setting up a "chain" of devices (from *outhole* to *trough* to *plunger lane*), and you're breaking up the special tags so that each device is tagged with it's exact role. (And hey! Now you know why these are all separate tags in MPF instead of a single tag called "trough".)

See *Setting up a System 11 Style Trough* for details.

(E) Final Steps and additional information

MPF's System 11 interface is new, and we haven't yet built a complete game using it. There are most likely things that we haven't thought of yet, so if you're using MPF with a System 11 machine, please post to the forum if you find anything that's weird or that doesn't work as expected.

Snux on Pinside.

This is an example code block with the main Sys11 elements in.

```
hardware:
  platform: virtual
  driverboards: wpc
  coils: snux
  switches: snux

system11:
  ac_relay_delay_ms: 75
  ac_relay_driver: c_ac_relay

snux:
  diag_led_driver: c_diag_led_driver

digital_outputs:
  flipper_enable_relay:
    number: c23
    type: driver
    enable_events: ball_started
    disable_events: ball_will_end

coils:
  c_diag_led_driver:
    number: c24
    default_hold_power: 1.0
  c_ac_relay:
    number: c25
    default_hold_power: 1.0
  c_side_a1:
    number: c11a
  c_side_a2:
    number: c12a
    default_hold_power: 0.5
  c_side_c1:
    number: c11c
  c_side_c2:
    number: c12c
    default_hold_power: 0.5
```

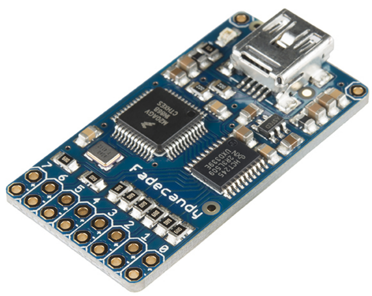
What if it did not work?

Have a look at our *hardware troubleshooting guide*.

How to configure a FadeCandy RGB LED Controller

Related Config File Sections
hardware:
lights:
fadecandy:
open_pixel_control:

MPF allows you to use a FadeCandy LED controller to drive the LEDs in your pinball machine. A FadeCandy is a small, cheap ($25) USB controller which can drive up to 512 serially-controlled RGB LEDs.



You can use the FadeCandy in place of connecting your LEDs to a P-ROC/P3-ROC controller, or you can choose to drive some LEDs via your primary pinball controller and some via the FadeCandy. (This is useful if you want to use more LEDs than what your controller platform supports.)

You can connect up to four FadeCandy boards to drive a total of 2048 LEDs (Which would be insane. And awesome.)

You can read more about the FadeCandy on the main page of the FadeCandy software repository in GitHub or on Adafruit or SparkFun, where you can buy one for $25. The FadeCandy is very advanced, offering advanced light processing capabilities such as dithering and interpolation that are not available if you just control LEDs directly.

If you're not familiar with the FadeCandy, check out this intro video from SparkFun:

https://youtu.be/-4AUBjV7Y-w

Overview video about serial LEDs:

https://youtu.be/Q9BG9T7Kj4A

1. Understanding all the parts and pieces

Before we dig in to setting up a FadeCandy with MPF, let's look at how all the various components will fit together:

- The FadeCandy is a piece of hardware that talks to your host computer via USB. (So if you use it in a pinball machine then you'll have two devices connected via USB—your pinball controller and your FadeCandy.)

- The FadeCandy hardware is driven a FadeCandy server software that you'll run on your host computer along side the MPF game engine and the MPF media controller. The FadeCandy server talks to the FadeCandy hardware via a USB driver.
- The FadeCandy server receives instructions for LEDs connected to the FadeCandy via a protocol called Open Pixel Control (OPC).

Putting it all together, MPF talks to the FadeCandy server via OPC, and the FadeCandy server talks to the FadeCandy hardware via USB.

2. Download the FadeCandy package from GitHub

The first step is to download the FadeCandy package from GitHub. You can unzip it to wherever you want.

3. Install the FadeCandy drivers

When I plugged the FadeCandy hardware into my Windows computer, the driver did not install automatically. Running the fcserver (next step) said it was installing the drivers, but that didn't do anything for me. (It just said "this may take awhile" but I killed it when it didn't seem like it was actually doing anything.)

In my case, I googled and found this procedure to build custom .inf files for Windows. It seems crazy but it wasn't too bad. I had to build two: One for the FadeCandy device and one for the FadeCandy boot loader. Either way, you can follow the docs and the forums around the FadeCandy and get it setup.

4. Setup the fcserver

The FadeCandy download package includes pre-built binaries for Mac and Windows. On Linux you can compile it. Again, the FadeCandy documentation has details about how to do this.

At this point you should be able to run the fcserver and to talk to your FadeCandy LEDs and get them to do things. There are a bunch of sample apps in the FadeCandy package that are kind of cool.

5. Set your LEDs to use the "fadecandy" platform

Next you need to configure your LEDs in MPF to use the `fadecandy` platform. By default, all types of devices are assumed to be using the same platform that you have set in the *hardware:* of your machine config file. So if your platform is set to `fast`, MPF assumes your LEDs are connected to a FAST controller, and if your platform is set to `p_roc` or `p3_roc`, MPF assumes your LEDs are connected to a PD-LED board.

To configure MPF to use FadeCandy LEDs, you can add an entry to the `hardware:` section of your machine config to tell it to override the default platform for your LEDs and to instead use the `fadecandy` platform, like this:

```
hardware:
  platform: p_roc
  driverboards: pdb
  lights: fadecandy
```

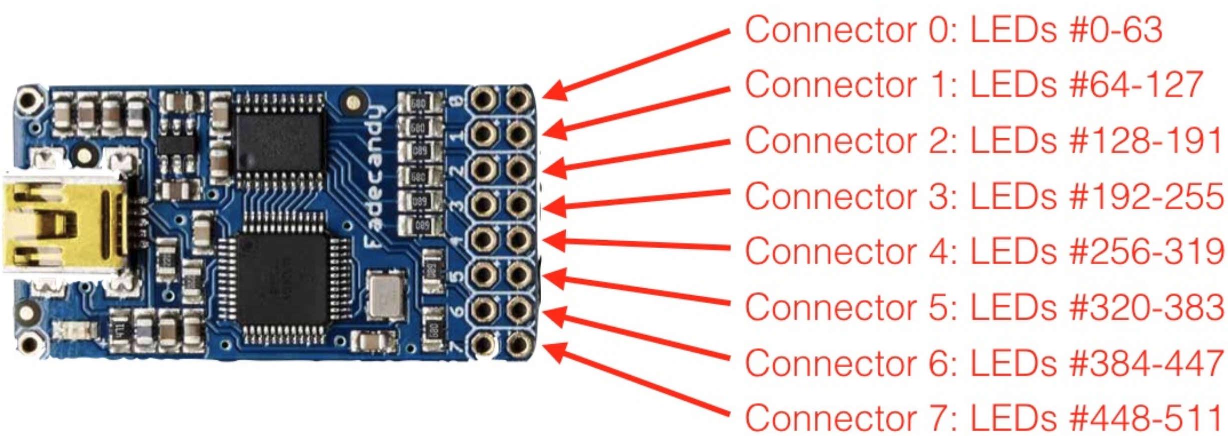
See the *Mixing-and-Matching hardware platforms* guide for more information about setting device-specific default platforms versus overriding the platform for individual devices.

6. Understanding FadeCandy LED numbering

The FadeCandy hardware has 8 connectors for LEDs, each of which can support up to 64 RGB LEDs (for 512 RGB LEDs total). The connectors are numbered 0-7.

The individual LED numbers are sequential across channels. The first LED on Connector 0 is #0, the second is #1, etc., up #63 on Connector 0. Then Connector 1 picks up where Connector 0 leaves off, with the first LED on Connector 2 being #64, and so on. The FadeCandy doesn't actually know how many LEDs are connected to each connector, so the first LED on Connector 1 is always LED #64 even if you have less than 64 LEDs physically connected to Connector 0.

The following diagram explains how the numbering works:



Consider the following config:

```
lights:
  l_led0:
    number: 0    # first LED on connector 0
  l_led1:
    number: 1    # second LED on connector 0
  l_led2:
    number: 128    # first LED on connector 2
```

(If you're familiar with the Open Pixel Control protocol, all of the LEDs on a single FadeCandy board are on the same OPC channel, which is technically what you're specifying with the number before the dash.)

6a. Numbering with multiple channels

You can also assign different OSC channels to your connectors. This has certain performance advantages and allows nicer numbering.

Start your fadecandy server with the following config:

```
{
    "listen": ["127.0.0.1", 7890],
    "verbose": true,
    "color": {
        "gamma": 2.5,
        "whitepoint": [1.0, 1.0, 1.0]
    },
    "devices": [
        {
            "type": "fadecandy",
            "serial": "YOUR_FADECANDY_SERIAL",
            "map": [
                [ 0, 0, 0, 64 ],
                [ 1, 0, 64, 64 ],
                [ 2, 0, 128, 64 ],
                [ 3, 0, 192, 64 ],
                [ 4, 0, 256, 64 ],
                [ 5, 0, 320, 64 ],
                [ 6, 0, 384, 64 ],
                [ 7, 0, 448, 64 ]
            ]
        }
    ]
}
```

Replace *YOUR_FADECANDY_SERIAL* with the serial of your fadecandy. The serial will be shown on the console of *fcserver* when connecting your fadecandy.

Then configure your lights as follows:

```
lights:
  l_led0_0:
    number: 0-0    # first LED on connector 0
  l_led1_0:
    number: 1-0    # first LED on connector 1
  l_led1_1:
    number: 1-1    # second LED on connector 1
  l_led7_20:
    number: 7-20    # twentyth LED on connector 7
```

6b. Numbering with multiple Fadecandy Boards

If you want to use multiple FadeCandy boards we suggest the following config:

```
{
    "listen": ["127.0.0.1", 7890],
    "verbose": true,
    "color": {
        "gamma": 2.5,
        "whitepoint": [1.0, 1.0, 1.0]
    },
    "devices": [
        {
            "type": "fadecandy",
```

(continues on next page)

(continued from previous page)

```
            "serial": "YOUR_FADECANDY_SERIAL1",
            "map": [
                [ 0, 0, 0, 64 ],
                [ 1, 0, 64, 64 ],
                [ 2, 0, 128, 64 ],
                [ 3, 0, 192, 64 ],
                [ 4, 0, 256, 64 ],
                [ 5, 0, 320, 64 ],
                [ 6, 0, 384, 64 ],
                [ 7, 0, 448, 64 ]
            ]
        },
        {
            "type": "fadecandy",
            "serial": "YOUR_FADECANDY_SERIAL2",
            "map": [
                [ 8, 0, 0, 64 ],
                [ 9, 0, 64, 64 ],
                [ 10, 0, 128, 64 ],
                [ 11, 0, 192, 64 ],
                [ 12, 0, 256, 64 ],
                [ 13, 0, 320, 64 ],
                [ 14, 0, 384, 64 ],
                [ 15, 0, 448, 64 ]
            ]
        },
        {
            "type": "fadecandy",
            "serial": "YOUR_FADECANDY_SERIAL3",
            "map": [
                [ 16, 0, 0, 64 ],
                [ 17, 0, 64, 64 ],
                [ 18, 0, 128, 64 ],
                [ 19, 0, 192, 64 ],
                [ 20, 0, 256, 64 ],
                [ 21, 0, 320, 64 ],
                [ 22, 0, 384, 64 ],
                [ 23, 0, 448, 64 ]
            ]
        }
    ]
}
```

Replace *YOUR_FADECANDY_SERIAL1*, *YOUR_FADECANDY_SERIAL2* and *YOUR_FADECANDY_SERIAL3* with the serials of your fadecandy boards (you can use more or less than three). The serial will be shown on the console of *fcserver* when connecting your fadecandy.

Afterwards, configure your lights as follows:

```
lights:
  l_led0_0:
    number: 0-0    # first LED on connector 0 on board 0
  l_led1_0:
    number: 1-0    # first LED on connector 1 on board 0
```

(continues on next page)

(continued from previous page)

```
  l_led1_1:
    number: 1-1    # second LED on connector 1 on board 0
  l_led7_20:
    number: 7-20    # twentyth LED on connector 7 on board 0
  l_led8_0:
    number: 8-0    # first LED on connector 0 on board 1
  l_led8_1:
    number: 8-63    # last LED on connector 1 on board 1
  l_led17_1:
    number: 17-1    # second LED on connector 1 on board 2
```

7. Unterstanding MPF light numbers and channels

In MPF *lights* abstract a light source which emits arbitrary colors. However, this is not true for all real lights. Some support only white (GIs), others only a single-color (i.e. red inserts) and others support full RGB. For that reason MPF knows *light numbers and channel numbers*. Internally, a light consists of one or multiple channels. For instance, a single-color GI will contain a single white channel. While a RGB light will control a red, green and a blue channel. A white light behind a red insert should be a single red channel (because it cannot emit other colors through the red insert). You can configure those channels using the `channels` setting or use `start_channel` and `type` to define the channels. See *Lights* for details.

However, in most cases a platform supports one type of lights (per `subtype`) this would be overly verbose and we added the `number` setting for configuring lights in the common platform way. For instance a platform for GIs will configure single channel white lights or a serial LED controller will configure RGB lights with three channels.

Fadecandy assumes RGB lights by default. For everything else (i.e. RGBW) you have to use channels.

Light Numbers

Fadecandy numbers use the format: `osc_channel-number`

If you mapped OSC channels as described in (6b/c) set them as `osc_channel`. `number` is the index of your light in the chain.

Internally, Fadecandy assumes three channels per LED (RGB/GRB *WS2811/WS2812 LEDs*).

Channels

Fadecandy channels use the format: `osc_channel-channel_index`

`channel_index` is `number * 3`. This is because serial LEDs are traditionally RGB (or GRB) LEDs with exactly three channels. However, this is not true for RGBW or similar LEDs which do not work with this style of numbering. Luckily, you can chain them instead and have MPF calculate the internal channels for you:

```
lights:
  led_0:
    start_channel: 0-0
    subtype: led
```

(continues on next page)

(continued from previous page)

```
    type: rgb    # will use red: 0-0, green: 0-1, blue: 0-2
  led_1:
    previous: led_0
    subtype: led
    type: rgbw   # will use red: 0-3, green: 0-4, blue: 0-5, white: 0-6
  led_2:
    previous: led_1
    subtype: led
    type: rgbw   # will use red: 0-7, green: 0-8, blue: 0-9, white: 0-10
```

See *WS2811 and WS2812 LEDs in Pinball* for details.

8. Launch the fcserver

In order for MPF to communicate with the FadeCandy, the fcserver has to be running. Refer to the FadeCandy documentation for instructions for this. On Windows, for example, it's just called `fcserver.exe`.

There are several command line options you can use with the server, though you don't need any of them with MPF unless you have more than one FadeCandy board connected.

You should launch fcserver in its own window since it will take over the console when it's running. It's also safe to keep it running all the time, or you can add it to a batch file to run it automatically. On my system, the fcserver puts some error message on the screen about not being able to connect to something, but everything still works even with that message continually being written to the console. (I think it's something to do with the P-ROC's FTDI driver? It only comes up when the P-ROC is on.)

9. Additional FadeCandy LED options

The FadeCandy hardware supports some advanced options which are configured in the *fadecandy:* section of your machine configuration file. Specifically, you can set the keyframe interpolation, dithering, gamma, white point, linear slope, and linear cutoff. The defaults should be fine for almost everyone, though you can go nuts if you want.

10. Color Correction

If you are using RGB LEDs, they might not be perfectly white when you turn them on. They might be pinkish or blueish instead depending on the brand of the LED. To a certain extend this is normal/expected and you can compensate for it by configuring *hardware color correction in the fadecandy*. If you need more than one correction profile (e.g. for multiple LED models) you need to fall back to *software color_correction profiles in light_settings*. Hardware correction should be preferred and give you much more dynamic range.

What if it did not work?

Have a look at our *fadecandy hardware troubleshooting guide*.

Troubleshooting Fadecandy

If you got problems with your hardware platform we first recommend to read our *troubleshooting guide*. Here are some hardware platform specific steps:

Enable Debugging

If you got problems with your platform try to enable debug first. As described in the *general debugging section* of our *troubleshooting guide* this is done by adding `debug: true` to your `fadecandy` config section:

```
fadecandy:
  debug: true
```

This will add a lot more debugging and might slow down MPF a bit. We recommend to disable/remove it after finishing debugging.

Flickering Lights after a few Restarts

At some point fadecandy might exhibit erratic behaviour or flickering lights after a few restarts of MPF. This usually can be fixed by power cycling the fadecandy (i.e. unplug it from USB and plug it in again). We created a bug report in the fadecandy repository for this case. We suspect a race which triggers some data corruption in the fadecandy firmware. If you are an embedded engineer or know anybody who could help to fix this issue please let us know. Nevertheless, we have never seen this outside of debugging sessions where we restart MPF frequently so it manageable once you know what it is.

Your hardware is not working at all

If your hardware is not working at all make sure that you removed the options `-X`, `-x` and `--vpx` from your `mpf both` or `mpf game` command line. Those options will overwrite the settings in your `hardware` section and MPF will not even try to connect to your hardware. If you got config errors we suggest you add `-X` to figure things out without interfacing real hardware all the time. Just keep that option in mind.

Another stupid thing to check: Is your hardware connected to your PC? We know it is stupid but a loose USB connector has happened to most of us.

On Linux you might want to run the command `lsusb` which should show both of your micro controllers connected. You should see two lines similar to

```
Bus 002 Device 014: ID 0483:5740 STMicroelectronics Virtual COM Port
Bus 002 Device 015: ID 0483:5740 STMicroelectronics Virtual COM Port
```

If you are unsure about the output, run the command once with your controllers connected and once without. If there is no difference, then for sure the USB device is not properly connected.

Add debugging to related devices

If you got problems with some switches also add `debug: true` to those as it will give to more insights into the intentions of those devices. Same will work for flippers, coils, lights, servos, steppers and more. See *general debugging section* for details.

Run MPF with verbose flag

See *general debugging section* for details. TLDR: run `mpf both -t -v -V`.

Report Your Issue and Ask For Help

If you cannot find the issue yourself please prepare some information about your issue according to our *troubleshooting guide* and ask in our forum.

Consider Improving the Documentation

Did you solve your issue but found that some relevant information in the documentation is missing or should be linked/located elsewhere? Either tell us in the forum or consider *improving the documentation* yourself to save future users some troubles the same way others saved you some troubles by writing this documentation.

I2C Servo Controllers

Related Config File Sections
hardware:
servo_controllers:
servos:

MPF currently supports PCA9685/PCA9635 based servo controllers via I2C. One example for such a controller is the Adafruit 16-Channel 12-bit PWM/Servo Driver. You can use any I2C platform supported by MPF (see *I2C Platforms in MPF*).

Overview video about *servos*:

https://youtu.be/wA6KEODwQ5w

1. Installing I2C Servo Controllers

Connect the controller to the I2C port and add the following config section:

```
hardware:
  servo_controllers: i2c_servo_controller
```

0x40 is actually the default I2C address for this chip but it might be different for some chips.

2. Add your servos

Add your servos to config:

```
servos:
  servo1:
    number: 0  # first servo on controller
```

All these config options are explained in-depth in the *servos: section* of the config file reference.

You can also provide an I2C address per servo:

```
servos:
  servo_on_controller_63_0:
    number: 63-0  # first servo on board with ID 0x3F / 63
  servo_on_controller_63_1:
    number: 63-1  # second servo on board with ID 0x3F / 63
```

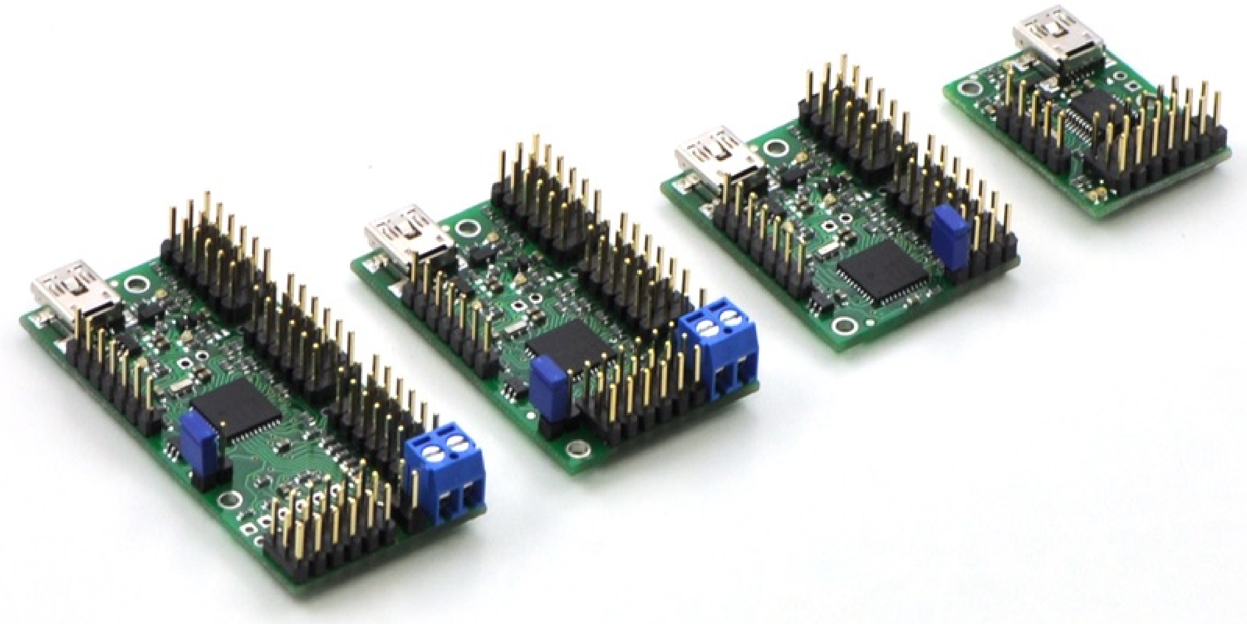
What if it did not work?

Have a look at our *hardware troubleshooting guide*.

Pololu Maestro Servo Controller

Related Config File Sections
pololu_maestro:
servos:

MPF supports servos connected to Pololu Maestro servo controllers. Each Maestro can control multiple servos, with models that control 6, 12, 18, or 24 servos.



Here is an explanation video by the pinball amigos on how to setup a pololu maestro (and more):

https://youtu.be/1QOOJNtsGxw

Overview video about *servos*:

https://youtu.be/wA6KEODwQ5w

1. Install the Pololu Maestro drivers

Just like any hardware device you connect to a computer, you need to install the drivers so your computer can see it. It is easier to do the initial hardware configuration on a Windows PC. Follow the "Getting Started" section of the Pololu Maestro Servo Controller User's Guide. You will need to set Maestro's serial mode to USB Dual Port on the Serial Settings tab of the Maestro Control Center.

2. Configure your hardware platform section

Next, you need to tell MPF that you want to use the `pololu_maestro` platform for servos. (MPF supports several different models of servo controllers.)

To do this, add `servo_controllers: pololu_maestro` to the `hardware:` section of your machine-wide config file, like this:

```
hardware:
  servo_controllers: pololu_maestro
```

This tells MPF that you want the default servo platform to be `pololu_maestro`. If you happen to be using multiple different types of servo controllers, you can override the default by adding a `platform:` entry to individual servo devices (just like any device in MPF that can have its platform overwritten in the device config).

3. Configure the serial port

Next, you need to tell MPF what port the Maestro is connected to. (Note that when you plug in the Maestro, you'll see two serial ports appear. You want to use the first one (the lower number).

Add a section to your machine-wide config like this:

```
pololu_maestro:
  port: COM5
```

On Linux or Mac, it will probably look like this:

```
pololu_maestro:
  port: /dev/ttyACM0
```

4. Add your servo devices

Now that all your hardware is configured, you can add the actual servos to your machine config. In MPF, servos are just like any other device (light, LEDs, coils, etc.) You add a `servos:` section to your config, and then create sub entries in there for each servo you have.

For example:

```
servos:
  servo1:
    servo_min: 0.2
    servo_max: 0.8
    positions:
      0.1: servo1_down
      0.9: servo1_up
    reset_position: 0.5
    reset_events: reset_servo1
    speed_limit: 0.5
    acceleration_limit: 0.5
    number: 1
  servo2:
    positions:
      0.2: servo2_left
      1.0: servo2_home
    reset_position: 1.0
    reset_events: reset_servo2
    number: 2
```

Okay, there's a lot going on in there. Let's break it down.

First, all these config options are explained in-depth in the *servos: section* of the config file reference. But let's point out a few Maestro-specific things here.

The `number:` of the servo is simply which channel on the Maestro board each servo is connected to. These numbers start with 0, so a Micro Maestro 6 supports six servos via numbers 0-5, the Mini Maestro 12 supports twelve servos numbered 0-11, etc.

All servo positioning in MPF is controlled via a floating point value from 0.0 to 1.0. In other words, if you tell a servo to go to position 0.0, that will be one end of its motion, and position 1.0 will be the other end. A value of 0.4 will tell the servo to move to a position that's 40% along from the start limit to the stop limit, etc.

So that's universal, 0.0 - 1.0, throughout MPF.

The way servos actually move to a position is that the servo controller sends a series of microsecond-level pulses which the servo reads and can then translate into a certain position. The actual value of these pulses varies depending on the servo controller and servos you actually have.

You may also set servo_min and servo_max if the servo is trying to move beyond its (hardware) limits when setting it to position 0.0 or 1.0. Those two values will be applied to all positions. For instance, if you move it to 0.0 it will actually move to servo_min (0.2 in the example) and to servo_max for 1.0 (0.8 in the example). Everything in between will be interpolated.

The Pololu Maestro servo controllers can accept speed and acceleration settings which specify how fast the servo moves to the new position, and how (or whether) it accelerates and decelerates when starting and stopping. If you want to use these add the `speed_limit:` and `acceleration_limit:` settings to your config.

5. Using the servo in your game

The servo's `position:` setting contains a list of numerical servo values mapped to MPF events. So to move a servo in your game, just add the position you want to the list and then post that event.

Again, see the *servos: section* of the config file reference for details.

6. Future enhancements

Multiple Pololu Maestro controllers can be chained together (via a single USB port). We don't have support for that yet. (It requires adding and additional address setting to the servo config.) If you want that, let us know and we'll add it.

What if it did not work?

Have a look at our *hardware troubleshooting guide*.

How to use Pololu Tic in MPF

Related Config File Sections
hardware:
pololu_tic:
tic_stepper_settings:
switches:
steppers:

The Pololu Tic is a stepper controller which can control one stepper via USB. Multiple versions with different power rating exist but they all work the same from the perspective of MPF.

TODO: Add a picture of a Pololu Tic

Overview video about *steppers*:

https://youtu.be/YaRNBU0OHGc

Installation

To use the Pololu Tic you need to install `ticcmd` from Pololu. Follow their Installation instructions for ticcmd.

Connecting your stepper

Connect your stepper according to the Pololu manual.

Configuring your stepper

Afterwards, you can use steppers from MPF. This is an example:

```
#config_version=5
hardware:
  stepper_controllers: pololu_tic

switches:
  s_home:
    number: 1
```

(continues on next page)

(continued from previous page)

```
steppers:
  stepper1:
    number: 1
    homing_mode: switch
    homing_switch: s_home
    named_positions:
      10: test_00
      20: test_01
      50: test_10
    platform_settings:
      max_acceleration: 20000
```

You can set certain pololu-specific settings in `platform_settings`. See *tic_stepper_settings:* for details.

What if it did not work?

Have a look at our *hardware troubleshooting guide*.

How to configure a "SmartMatrix" RGB LED DMD

Related Config File Sections
hardware:
rgb_dmds:
smartmatrix:
displays:

This guide explains how to connect a SmartMatrix RGB LED DMD to a pinball machine running MPF.

A SmartMatrix is a cheap ($20) board that you attach to a Teensy ($25) microcontroller which lets you connect an RGB DMD matrix display to the computer running MPF. It's a standalone solution which you can use to add an RGB DMD to a pinball machine that's using FAST Pinball, P-ROC/P3-ROC, or OPP controller hardware.

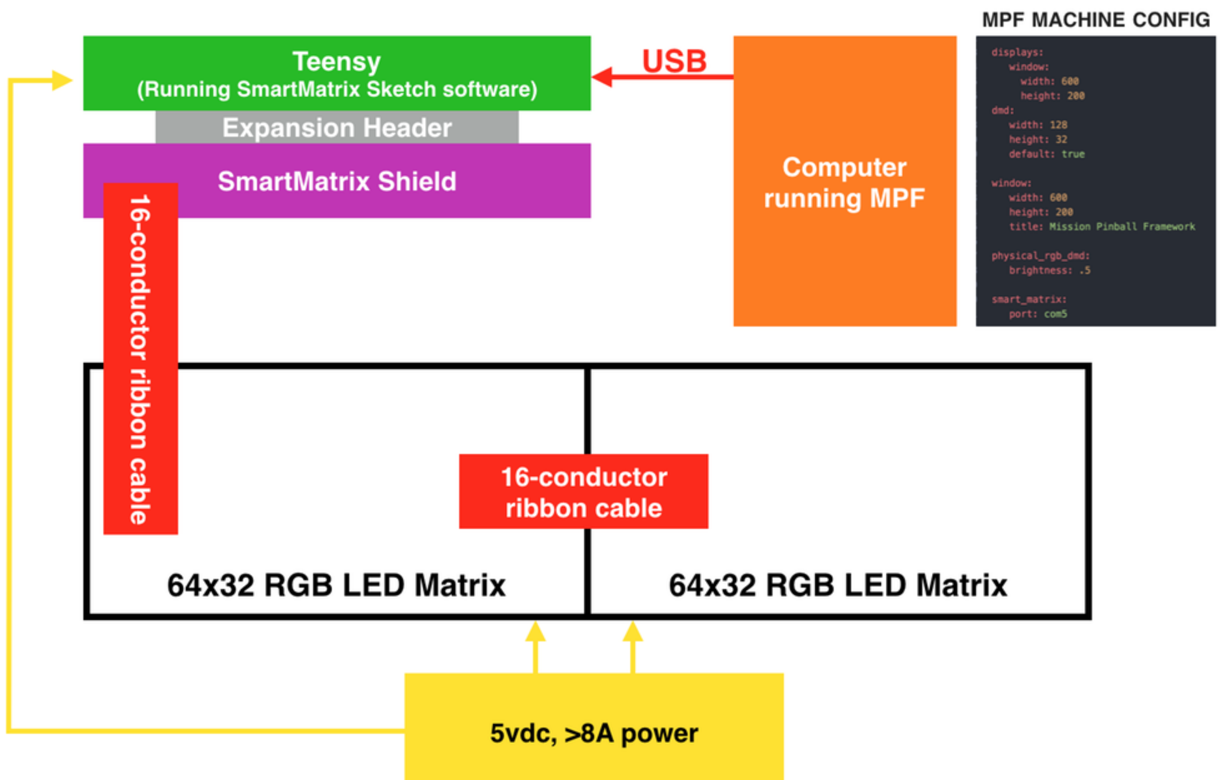
MPF supports several different types of RGB DMDs, and the SmartMatrix is just one of the options. More information about this type of display and other options that MPF supports is available in the *Using an RGB full-color LED DMD* documentation.

Here's an image of the SmartMatrix RGB DMD in action:

And a video which explains it all:

https://youtu.be/zbZQCByeXOU

The following diagram shows how all the components fit together:



1. Buy all the parts you need

This solution is very much a "home brew" solution that will require you to buy a lot of parts from various sources.

Alternatively, FAST pinball also offers a *RGB DMD* which contains controller, panels and mounting brackets (ask them directly since it is not currently listed on their website). If you go with this solution skip steps 1 to 3. You still need a power supply (step 4).

(1) The Panels

We originally had to buy the panels directly from China via AliExpress, but now FAST Pinball sells a kit. The FAST Pinball option is nice because the price is great and they also include a mounting bracket that fits a standard DMD cutout (ask them directly since it is not currently listed on their website).

If you buy the panels yourself on AliExpress, you'll pay about the same price for just the panels, you won't have a mounting bracket, and you'll have to deal with customer support from China. Also FAST tests the panels to make sure all the pixels work—a problem people were running into when buying from AliExpress.

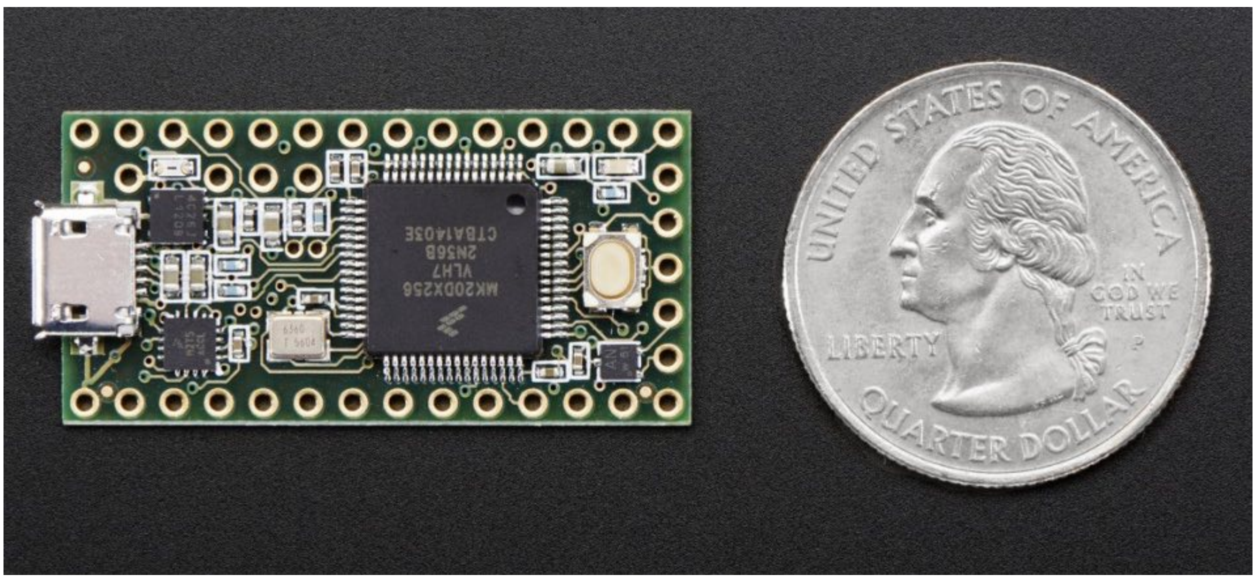
(2) The Teensy

Once you have your panel, you need a way to talk to them via a computer. The panels use some kind of 16-pin signalling system which is some kind of standard in the gigantic advertising display industry.

The solution for MPF is to use a Teensy 3.2 or 3.5 (which is kind of like an Arduino). The Teensy is available from multiple sources for about $20. Here's the link to the website of the guy who actually built it, and you can also get it from Adafruit which is nice because you also need the shield (from the next step) which is also available from them.

The Teensy runs the same software sketches as Arduinos, though it has a slightly different processor architecture which is needed for the rapid bit-shifting of data needed to control these panels.

Here's a Teensy:

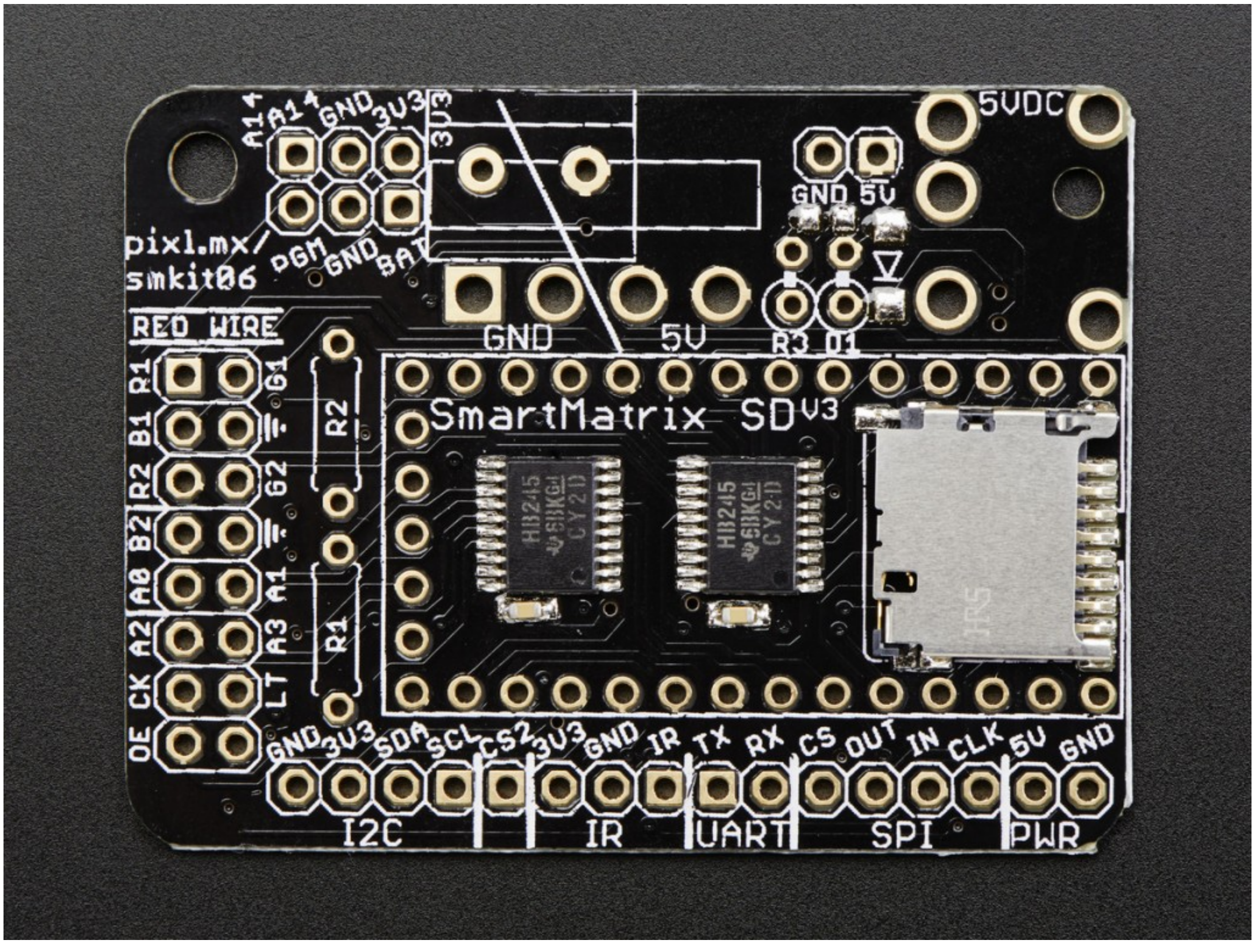


The software to run the Teensy is open source (more on that in Step 3) and the Teensy has a USB port which you connect to your computer which MPF uses to send the display data to the panels.

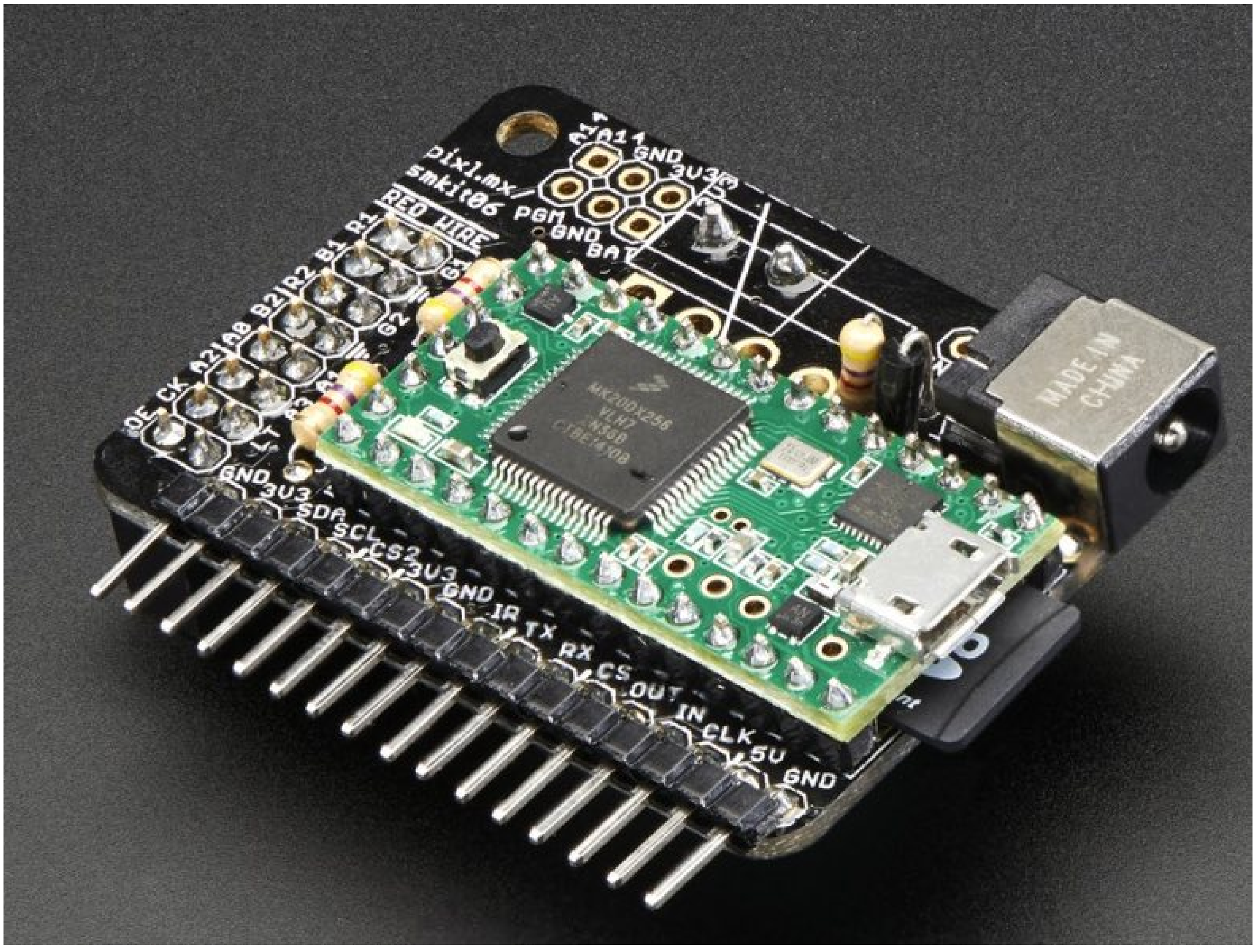
(3) The SmartMatrix Shield

Next you need a way for the Teensy to connect to the displays. That can be done with the SmartMatrix shield (V4 of the shield is $20 at Adafruit).

The SmartMatrix shield is a “dumb” device that basically just connects the Teensy’s GPIO pins to the 16-pin ribbon cable that drives the displays.



The Teensy mounts onto the SmartMatrix shield, creating a single unit which accepts data via USB on one end and spits out the 16-pin signal for the display panels on the other.



(4) The Power Supply

These RGB LED displays require 5vdc for power. At first you might think, "Cool! I have 5v elsewhere in my machine, so I'll just tap into that!" Not so fast. These displays require *a lot* of power. After all, each pixel is actually three separate LEDs (one each for red, green, and blue), and a 128x32 display means that you have 4,096 pixels. So that's 12,228 LEDs you need to power!

If you're ordering your RGB LED display panels from FAST Pinball, you can also order a 5v, 10A power supply from them for $19.

An ATX computer power supply will probably have a decent amount of amps also, so that could be an option too, just check the specs. Any other 5V supply with decent power should also work.

One thing about these RGB LED-based displays is they are bright. Like, really, really bright. (We're talking "burn your retinas if you stare straight at them" kind of bright.)

So even though you can do the math and read that if every pixel is on, full white, 100%, that might take more power than you have, there is no way you're going to run these things at full brightness.

Even at 50% brightness, (which would draw only 50% power) most people find these panels to be too bright. One user runs his at 25%, another at 18%. So it's possible that you might be fine with 5-7 amps of power.

You'll need to connect the power supply up to both panels (the 128x32 display is made up of two 64x32 panels), and while you're at it you can also use it to power your Teensy.

There's a trace you have to cut on the Teensy to control whether it's powered externally or by USB. Don't hook it up to external power if you haven't cut that trace!

2. Load the SmartMatrix code onto the Teensy

Once your hardware's built, you need to load the code onto the Teensy which receives the display data via USB and converts and sends it to the pins connected to the SmartMatrix controller. The people who make the SmartMatrix controller have code sample code available. We just took their sample code, removed all the clutter we don't need, and made it available in the tools folder in the MPF download package. (Here's a direct link to the code which you can use since you probably installed MPF via pip and don't have the download package available.

Also, here's the original sample code we based our code on.

If you are using V4 of the shield, you need to insert this line of code in the first line:

```
#include <SmartLEDShieldV4.h> // this line must be first
```

The V4 shield's library uses more RAM which can causes the Teensy 3.2 to crash during animations or video playback. Using a Teensy 3.5 or 3.6 solves this issue as they have more RAM.

Note that the width and height of your display is set in lines 11 & 12. You can change that if you want to use a different size display.

Mark Sunnucks was able to run a 128x64 display by setting the height there and also by changing the DMAs from 4 to 2 in line 14.

Also note that you can set the brightness of the display in this code too. You can control the brightness in MPF as well, but if you know for sure (maybe due to power limitations) that you never want the brightness to go over a certain amount, then you can set it here and it will be "hard coded" into your Teensy. (You can change this and re-flash your Teensy at any time.)

Here's a quick overview of how to install this code onto the Teensy. Full instructions are here.

- Install the Arduino IDE v1.8.5
- Install the Teensyduino add-in which adds support for the Teensy
- Load the smart_matrix_dmd_teensy_code.ino sketch from the mpf/tools folder or this link
- Push the button on the Teensy to put it into programming mode
- Compile & load the code onto the Teensy from the Arduino IDE

3. Configure your SmartMatrix hardware settings

Once you have your hardware all set, you need to add a `smartmatrix:` section to your machine-wide config and which tells MPF how to talk to RGB DMDs that use the SmartMatrix platform.

The main thing you have to figure out is the port that the Teensy uses. On Windows, you can just open Device Manager and see which port appears when you plug in the Teensy.

On Mac or Linux, open up the terminal window and type the following command: `ls /dev/tty.*` The output of this command will look something like this on Mac:

```
/dev/tty.Bluetooth-Incoming-Port
/dev/tty.usbmodem1448891
```

Or this on linux:

```
/dev/ttyUSB0
/dev/ttyACM0
```

The port will be the one that has “usbmodem” in the name on Mac. On Linux it will probably be ttyUSBx or ttyACMx. (The actual number will likely be different on your system.) You can run this command with the Teensy unplugged, then plug it in, then run the command again, and see which port appears.

So on Windows, you’ll end up with something like:

```
hardware:
  rgb_dmd: smartmatrix
smartmatrix:
  smartmatrix_1:
    port: com12
    baud: 2500000
    old_cookie: false
```

And on Mac or Linux, it will look something like:

```
hardware:
  rgb_dmd: smartmatrix
smartmatrix:
  smartmatrix_1:
    port: "/dev/tty.usbmodem1448891"
    baud: 2500000
    old_cookie: false
```

Just enter the `baud:` and `old_cookie:` settings like they are in the example above. These are the settings that are needed for the SmartMatrix. If you are using the FAST DMD board set baud to *3000000*.

3. Add a physical RGB DMD device entry

Once you have your SmartMatrix hardware platform set, you need to create the actual device entry for the RGB DMD and map it back to the SmartMatrix platform.

You do this in the `rgb_dmds:` section of the machine config. This section is like the other common sections (switches, coils, etc.) where you enter the name(s) of your device(s), and then under each one, you enter its settings.

(And yes, in case you’re wondering, it’s possible to have more than one physical DMD.)

To do this, create a section in your machine-wide config called `rgb_dmds:`, and then pick a name for the DMD, like this:

```
rgb_dmds:
  smartmatrix_1:
    hardware_brightness: .17
    source_display: dmd
```

There are several settings you can enter here. (See the *rgb_dmds:* for details.)

You'll probably also want to configure the brightness, which is a multiplier from 0.0 to 1.0 that's applied to every pixel that's sent to the DMD. In other words, the example of `brightness: .17` means that each pixel will be shown at 17% brightness. (These things are crazy bright!)

Note: If you set the brightness multiplier in the sketch code .INO file you loaded onto the Teensy, then that will multiply the brightness after MPF sends it. In other words, if you set .5 in the config file and .5 in the sketch, then the final brightness will be 25%. You might want to set the absolute max brightness in the .INO file once and then fine-tune it via the config file later.

---

4. Set a source display

Now that you have everything configured, the last step is to make sure the DMD knows what content to show. In MPF, you do this by mapping a physical DMD to an *MPF display*.

By default, the DMD will look for a display (in your *displays:* section called "dmd". However you can override this and configure the DMD to use whatever logical display you want by setting a `source_display:` setting. (Just make sure that the width and height of your source display match the physical pixel dimensions of the DMD or else it will be weird.)

A final config you can test

At this point you're all set, and whatever slides and widgets are shown on the DMD's source display in MPF-MC should be shown on the physical RGB DMD.

That said, all these options can be kind of confusing, so we created a quick example config you can use to make sure you have yours set right. (You can actually just save this config to `config.yaml` in a blank machine folder and run it to see it in action which will verify that you've got everything working properly.)

Note: Be sure to change the `smartmatrix:port:` setting in this example config to match whatever port your Teensy is connected to.

To run this sample config, you can either run `mpf both`.

When you run it, do not use the `-x` or `-X` options, because either of those will tell MPF to not use physical hardware which means it won't try to connect to the Teensy.

Note that the *Using an RGB full-color LED DMD* guide has more details on the window and slide settings used in this machine config.

```
hardware:
  rgb_dmd: smartmatrix
displays:
  window:  # on screen window
    width: 600
    height: 200
  dmd:  # source display for the DMD
    width: 128
    height: 32
    default: true
```

(continues on next page)

(continued from previous page)

```
    round_anchor_x: left
window:
  width: 600
  height: 200
  title: Mission Pinball Framework
smartmatrix:
  smartmatrix_1:
    port: com5  # this will most likely be a different port for you
    baud: 2500000
    old_cookie: false
rgb_dmds:
  smartmatrix_1:
    brightness: .2
slides:
  window_slide_1:  # slide we'll show in the on-screen window
    - type: display  # this widget shows the DMD content in this slide too
      effects:
        - type: color_dmd
      width: 512
      height: 128
    - type: text
      text: MISSION PINBALL FRAMEWORK
      anchor_y: top
      y: top-3
      font_size: 30
      color: white
    - type: rectangle
      width: 514
      height: 130
      color: 444444
  dmd_slide_1:  # slide we'll show on the physical DMD
    - type: text
      text: IT WORKS!
      font_size: 30
      color: red
slide_player:
  init_done:
    window_slide_1:
      target: window
    dmd_slide_1:
      target: dmd
```

What if it did not work?

Have a look at our *hardware troubleshooting guide*.

RGB.DMD Controller

Related Config File Sections
hardware:
rgb_dmds:
smartmatrix:
displays:

The RGB.DMD controller was designed as a variant to the SmartMatrix that would be capable of both controlling RGB LED panels and accepting and decoding the DMD signal from an existing commercial pinball machine. As soon as RGB LED panels with spacing matching that of a DMD became available in September of 2015, Eli worked with the MPF developers to modify the SmartMatrix software and make it possible to stream color DMD images from MPF to SmartMatrix and RGB.DMD displays so that MPF machines can have color displays in the traditional 32x128 DMD form factor.

It's likely that no one would be using RGB LED DMDs if it wasn't for the efforts of Eli Curtz. Eli first posted about these types of panels in the P-ROC forum (now defunct) in 2014. At that time we could only find panels with 3mm spacing between pixels which was a bit larger than traditional pinball DMDs, but that's what kicked off the conversation about, "Whoa, maybe we could use these for 'real' color DMDs some day." Then in September 2015, Eli posted again telling us that we could now get panels with 2.5mm spacing which is the perfect size we need. Eli also showed us how to connect them and what software we needed to make everything work. So really everything here is because of Eli. All we did is take everything he showed us and write it down. (Well, that and we also created the interface for MPF, but that was the easy part.) So thanks Eli!

The Eagle files are available along with the code for those who'd like to build their own RGB.DMD board. Connect your panels and you are good to go.

In MPF, RGB.DMD works just like *How to configure a "SmartMatrix" RGB LED DMD* (go there for details). Can copy the following example (and replace `com12` with your com port):

```
hardware:
  rgb_dmd: smartmatrix
smartmatrix:
  smartmatrix_1:
    port: com12
    baud: 3000000
    old_cookie: false
rgb_dmds:
  smartmatrix_1:
    platform: smartmatrix
    source_display: dmd
```

What if it did not work?

Have a look at our *hardware troubleshooting guide*.

How to configure a PIN2DMD RGB LED DMD

Related Config File Sections
hardware:
rgb_dmds:
pin2dmd:
displays:

This guide explains how to connect a PIN2DMD RGB LED DMD to a pinball machine running MPF.

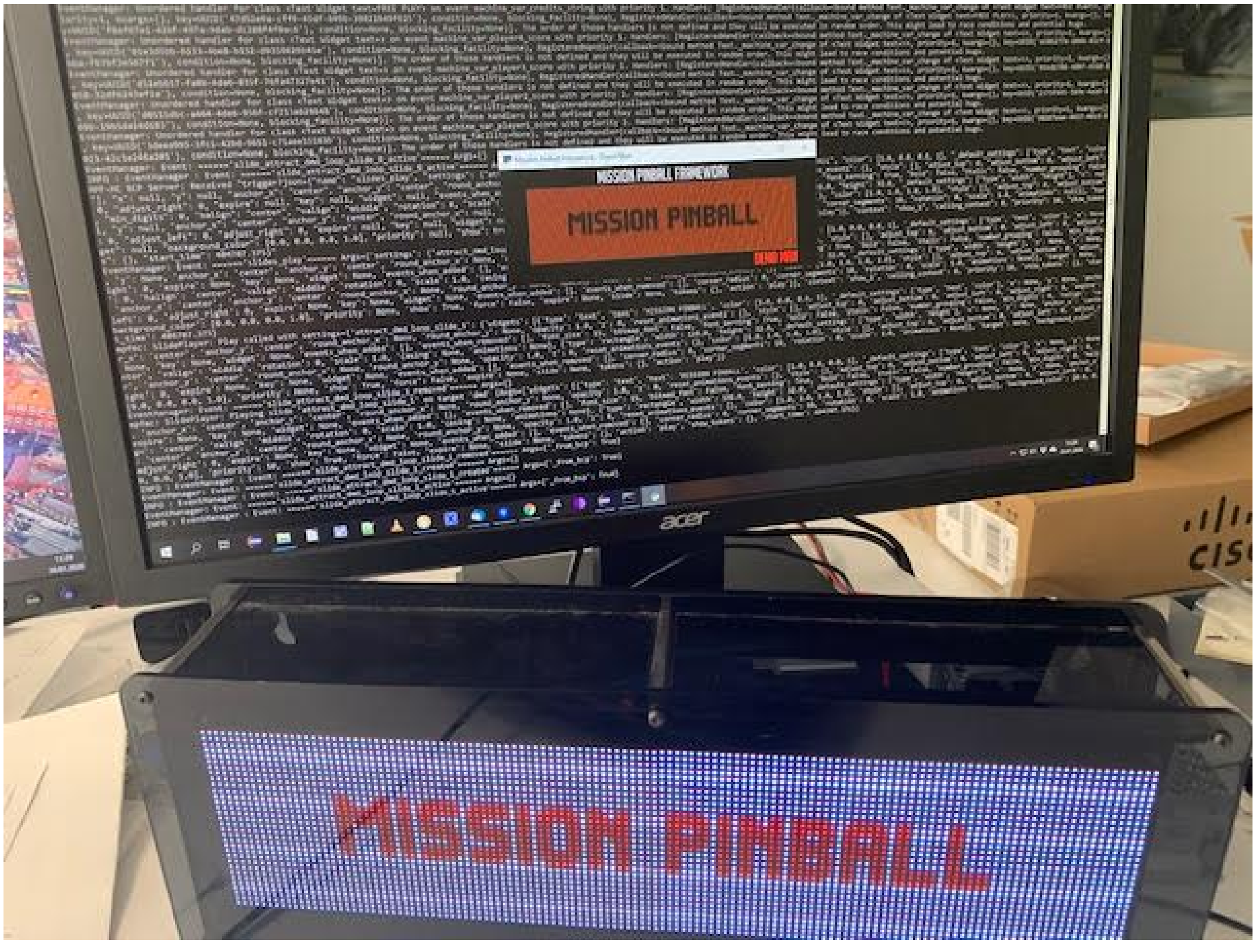
PIN2DMD is a RGB DMD display which connects to a PC via USB. It exists in a 128x32 (traditional pinball) and 192x64 pixel (large SEGA/Data East displays) version.

MPF supports several different types of RGB DMDs, and the PIN2DMD is just one of the options. More information about this type of display and other options that MPF supports is available in the *Using an RGB full-color LED DMD* documentation.

Overview video about Pin2DMD:

https://youtu.be/Q5fH-Q2umV4

This is how a 128x32 pixel PIN2DMD looks in action:



And this is how a 192x64 pixel PIN2DMD looks:

1. Buy and Assemble your PIN2DMD

To use PIN2DMD in MPF you should first install your panel as described on the PIN2DMD homepage (steps hardware and firmware).

2. Install pyusb

MPF uses libusb via pyusb to drive your PIN2DMD. To use your PIN2DMD you need to install pyusb using pip:

```
pip3 install pyusb
```

3. Configure your PIN2DMD hardware settings

Once you have your hardware all set, you need to add a `smartmatrix:` section to your machine-wide config and which tells MPF how to talk to RGB DMDs that use the SmartMatrix platform.

1. Add *pin2dmd* to your hardware section:

```
hardware:
  rgb_dmd: pin2dmd
pin2dmd:
   # debug: True           # uncomment this if you experience any issues and need debug output
  resolution: 128x32       # or 192x64 depending on your panel
  panel: rgb               # or rbg if colors are swapped
```

3. Add a physical RGB DMD device entry

Once you have your SmartMatrix hardware platform set, you need to create the actual device entry for the RGB DMD and map it back to the SmartMatrix platform.

You do this in the `rgb_dmds:` section of the machine config. This section is like the other common sections (switches, coils, etc.) where you enter the name(s) of your device(s), and then under each one, you enter its settings.

(And yes, in case you're wondering, it's possible to have more than one physical DMD.)

To do this, create a section in your machine-wide config called `rgb_dmds:`, and then pick a name for the DMD, like this:

```
rgb_dmds:
  default:  # your DMD
    hardware_brightness: .5      # adjust the brightness of your display if it is too bright
    fps: 30
```

There are several settings you can enter here. (See the *rgb_dmds:* for details.). For PIN2DMD the display currently has to be named `default` because there can be only one PIN2DMD connected.

You'll probably also want to configure the brightness, which is a multiplier from 0.0 to 1.0 that's applied to every pixel that's sent to the DMD. In other words, the example of `hardware_brightness: .2` means that each pixel will be shown at 20% brightness. (These things are crazy bright!)

4. Set a source display

Now that you have everything configured, the last step is to make sure the DMD knows what content to show. In MPF, you do this by mapping a physical DMD to an *MPF display*.

```
displays:
  window:  # on screen window - useful for debugging without real hardware
    width: 600
    height: 200
  dmd:  # source display for the DMD
    width: 128      # 192 if you got a 192x64 pin2dmd panel
    height: 32      # 64 if you got a 192x64 pin2dmd panel
    round_anchor_x: left
    default: true
```

By default, the DMD will look for a display (in your *displays:* section called "dmd". However you can override this and configure the DMD to use whatever logical display you want by setting a `source_display:` setting. (Just make sure that the width and height of your source display match the physical pixel dimensions of the DMD or else it will be weird.)

A final config you can test

At this point you're all set, and whatever slides and widgets are shown on the DMD's source display in MPF-MC should be shown on the physical RGB DMD.

That said, all these options can be kind of confusing, so we created a quick example config you can use to make sure you have yours set right. (You can actually just save this config to `config.yaml` in a blank machine folder and run it to see it in action which will verify that you've got everything working properly.)

To run this sample config, you can either run `mpf both`.

When you run it, do not use the `-x` or `-X` options, because either of those will tell MPF to not use physical hardware which means it won't try to connect to the Teensy.

Note that the *Using an RGB full-color LED DMD* guide has more details on the window and slide settings used in this machine config.

```
hardware:
  rgb_dmd: pin2dmd
pin2dmd:
  # debug: True           # uncomment this if you experience any issues and need debug output
  resolution: 128x32      # or 192x64 depending on your panel
  panel: rgb              # or rbg if colors are swapped
displays:
  window:  # on screen window
    width: 600
    height: 200
  dmd:  # source display for the DMD
    width: 128      # 192 if you got a 192x64 pin2dmd panel
    height: 32      # 64 if you got a 192x64 pin2dmd panel
    default: true
    round_anchor_x: left
window:
  width: 600
  height: 200
  title: Mission Pinball Framework
rgb_dmds:
  default:
    hardware_brightness: .2
    fps: 30
slides:
  window_slide_1:    # slide we'll show in the on-screen window
    - type: display  # this widget shows the DMD content in this slide too
      effects:
        - type: color_dmd
      width: 512
      height: 128
    - type: text
      text: MISSION PINBALL FRAMEWORK
      anchor_y: top
      y: top-3
      font_size: 30
      color: white
    - type: rectangle
      width: 514
```

(continues on next page)

(continued from previous page)

```
      height: 130
      color: 444444
  dmd_slide_1:  # slide we'll show on the physical DMD
    - type: text
      text: IT WORKS!
      font_size: 30
      color: red
slide_player:
  init_done:
    window_slide_1:
      target: window
    dmd_slide_1:
      target: dmd
```

What if it did not work?

Have a look at our *Pin2DMD hardware troubleshooting guide*.

Troubleshooting Pin2DMD

If you got problems with your hardware platform we first recommend to read our *troubleshooting guide*. Here are some hardware platform specific steps:

Enable Debugging

If you got problems with your platform try to enable debug first. As described in the *general debugging section* of our *troubleshooting guide* this is done by adding `debug: true` to your `pin2dmd` config section:

```
pin2dmd:
  debug: true
```

This will add a lot more debugging and might slow down MPF a bit. We recommend to disable/remove it after finishing debugging.

Check Brightness

Your display is not showing your slides? Check if your brightness is set high enough. You can adjust brightness in your `rgb_dmds` section:

```
rgb_dmds:
  default:  # your DMD
    brightness: .8      # adjust the brightness of your display if it is too bright or dim
    fps: 30
```

Your hardware is not working at all

If your hardware is not working at all make sure that you removed the options `-X`, `-x` and `--vpx` from your `mpf both` or `mpf game` command line. Those options will overwrite the settings in your `hardware` section and MPF will not even try to connect to your hardware. If you got config errors we suggest you add -X to figure things out without interfacing real hardware all the time. Just keep that option in mind.

Another stupid thing to check: Is your hardware connected to your PC? We know it is stupid but a loose USB connector has happened to most of us.

On Linux you might want to run the command lsusb which should show both of your micro controllers connected. You should see two lines similar to

```
Bus 002 Device 014: ID 0483:5740 STMicroelectronics Virtual COM Port
Bus 002 Device 015: ID 0483:5740 STMicroelectronics Virtual COM Port
```

If you are unsure about the output, run the command once with your controllers connected and once without. If there is no difference, then for sure the USB device is not properly connected.

Add debugging to related devices

If you got problems with some switches also add debug: true to those as it will give to more insights into the intentions of those devices. Same will work for flippers, coils, lights, servos, steppers and more. See *general debugging section* for details.

Run MPF with verbose flag

See *general debugging section* for details. TLDR: run mpf both -t -v -V.

Report Your Issue and Ask For Help

If you cannot find the issue yourself please prepare some information about your issue according to our *troubleshooting guide* and ask in our forum.

Consider Improving the Documentation

Did you solve your issue but found that some relevant information in the documentation is missing or should be linked/located elsewhere? Either tell us in the forum or consider *improving the documentation* yourself to save future users some troubles the same way others saved you some troubles by writing this documentation.

Raspberry PI DMD (rpi-rgb-led-matrix)

Related Config File Sections
hardware:
rpi_dmd:
rgb_dmds:
displays:

The rpi dmd platform can be used to control a RGB LED matrix on your Raspberry Pi (any model).

1. Connect the hardware

We suggest that you follow the tutorial in the rpi-rgb-led-matrix library.

2. Install the extension

You need to install the rgbmatrix extension on your RPi using the following command:

```
git clone https://github.com/hzeller/rpi-rgb-led-matrix.git
cd rpi-rgb-led-matrix
sudo apt-get update && sudo apt-get install python3-dev python3-pillow -y
make build-python PYTHON=$(which python3)
sudo make install-python PYTHON=$(which python3)
```

3. Configure your DMD

This is an example config:

```
hardware:
  platform: rpi_dmd
rpi_dmd:
  cols: 32
  rows: 32
  gpio_slowdown: 2
  pwm_lsb_nanoseconds: 300
window:
  width: 600
  height: 200
  title: Mission Pinball Framework
displays:
  window:   # on screen window
    width: 600
    height: 200
  dmd:   # source display for the DMD
    width: 32
    height: 32
    default: true
    round_anchor_x: left
rgb_dmds:
  rpi_dmd:
    source_display: dmd
slides:
  window_slide_1:   # slide we'll show in the on-screen window
    - type: display   # this widget shows the DMD content in this slide too
      effects:
        - type: color_dmd
      width: 512
      height: 128
    - type: text
      text: MISSION PINBALL FRAMEWORK
      anchor_y: top
      y: top-3
      font_size: 30
```

(continues on next page)

(continued from previous page)

```
      color: white
    - type: rectangle
      width: 514
      height: 130
      color: 444444
  dmd_slide_1:   # slide we'll show on the physical DMD
    - type: text
      text: IT WORKS!
      font_size: 30
      color: red
slide_player:
  init_done:
    window_slide_1:
      target: window
    dmd_slide_1:
      target: dmd
```

The size of your dmd (32x32 pixel in the example) should match your physical matrix. Also make sure to configure the *rpi_dmd:* section accordingly.

Note that the *Using an RGB full-color LED DMD* guide has more details on the window and slide settings used in this machine config.

4. Start MPF as root

For this library to work you need to start MPF as root like this:

```
sudo mpf game
```

This is needed for the matrix to access the hardware and it will drop privileges after it started.

Related How To guides
Installing Fantastic with RPI DMD

What if it did not work?

Have a look at our *hardware troubleshooting guide*.

MyPinballs Segment Display Controller

Related Config File Sections
hardware:
mypinballs:
segment_displays:
segment_display_player:

Those segment displays are controlled by a very simple serial protocol. Two variants exist: The original MyPinball controller which can controll existing segments and the TNA segment displays sold by PBL which includes four segments. Both can be controlled using this platform.

Video about segment displays:

https://youtu.be/Jyf3jxGXnTw

Mypinballs Segment Displays Controller

MyPinballs sells segment display controller which can be used with MPF to control existing Bally/Stern segment displays (or replacement displays). See the *Direct-wiring MyPinballs to 3rd-Party Segment Displays* section for details about how to wire those. Connect it to your PC using USB and control up to six segment displays.

Config looks like this:

```
hardware:
  segment_displays: mypinballs
mypinballs:
  port: /dev/ttyUSB0
segment_displays:
  display1:
    number: 1
  display2:
    number: 2
  display3:
    number: 3
  display4:
    number: 4
  display5:
    number: 5
  display6:
    number: 6
```

You can configure your serial port in *port*. See *segment_display* for more informations about how to drive segment display in your game.

Total Nuclear Annihilation Remake Serial Score Display Assembly

Alternative, PBL sells TNA segment displays which use the same serial protocol. The board is ready-made with four segment displays and a controller which can be controlled by MPF via USB.

Part number:

- PBL-600-0473-00

Config looks like this:

```
hardware:
  segment_displays: mypinballs
mypinballs:
  port: /dev/ttyUSB0
segment_displays:
  display1:
    number: 1
  display2:
    number: 2
  display3:
```

(continues on next page)

(continued from previous page)

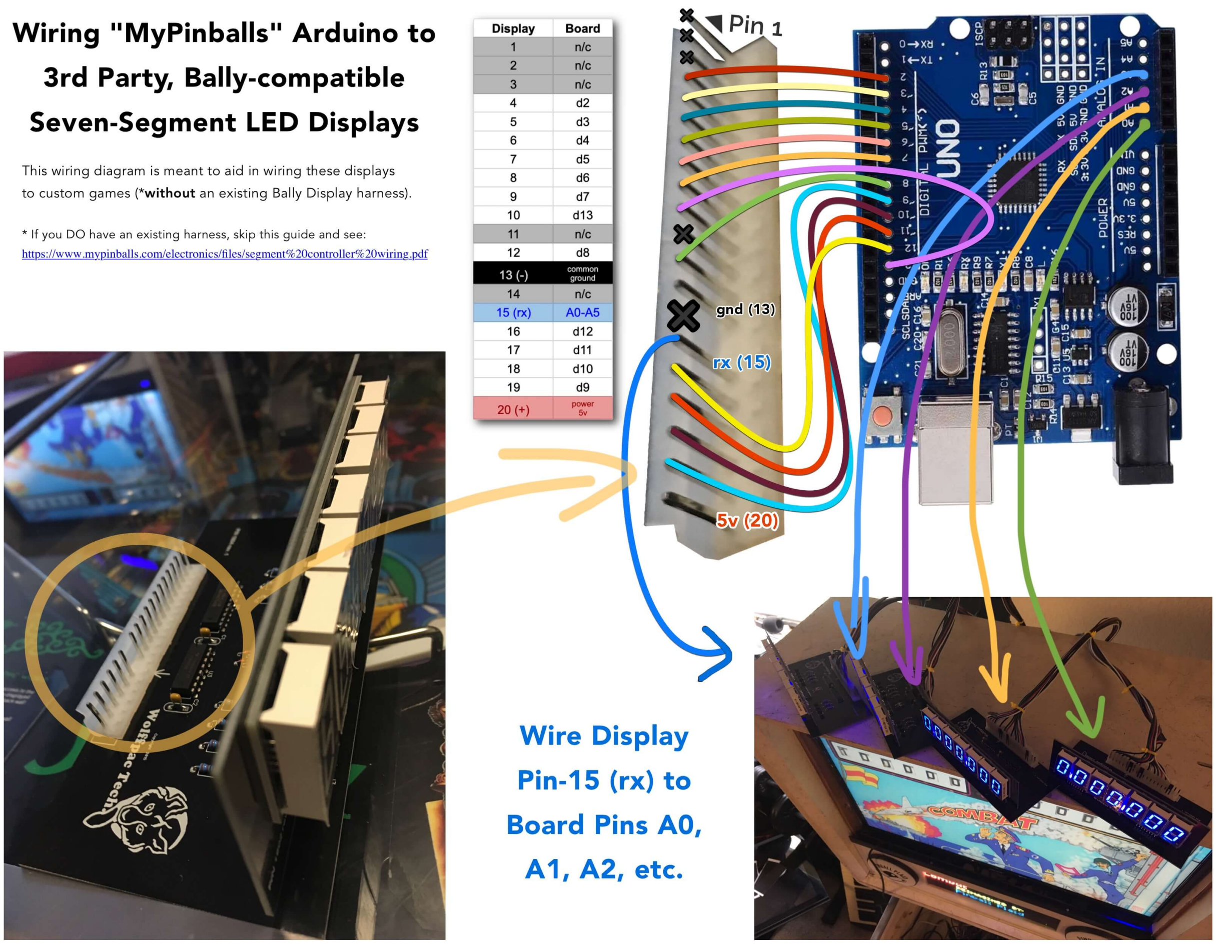
```
    number: 3
  display4:
    number: 4
```

You can configure your serial port in *port*. See *segment_display* for more informations about how to drive segment display in your game.

See Scotts description of the display for details.

Direct-wiring MyPinballs to 3rd-Party Segment Displays

The following is a wiring diagram for connecting the preloaded Arduino board (provided by MyPinballs) to 3rd-party, Bally-compatible 7 segment displays.



What if it did not work?

Have a look at our *hardware troubleshooting guide*.

What if it did not work?

Have a look at our *hardware troubleshooting guide*.

How to Connect Segment Displays as Lights to MPF

Related Config File Sections
hardware:
segment_displays:
lights:
light_segment_displays:

MPF can map segment displays to arbitrary lights which can be controlled via any hardware platform. You can select from multiple mappings (see *platform_settings* for details). Let us know if you need another mapping.

Video about segment displays:

https://youtu.be/Jyf3jxGXnTw

Hardware

BCD Seven Segment

Segment displays are readily available at most electronics suppliers. Most of them use some BCD encoder to save connectors. Those are easily recognizable because they got less than 8 connectors. You can use any driver or digital outputs on those. Be a bit careful with current driven light controllers (i.e. the PD-LED) here. Those cannot be dimmed currently (let us know if you need that).

Parallel Seven Segment

Those are not as common as BCD segment displays but still available. You can recognize them by more than 8 connectors. Make sure that your display is not multiplexed or it will not work without an additional controller chip. Those can be driven by any parallel LED controller (see *LEDs* for details). If you use drivers you will probably need current limiting resistors. In most cases BCD is simpler to use and will save you some outputs.

Those are also available as RGB. However, they often are multiplexed and will not work without an additional chip.

Serial Segment Displays

Additionally, there are serial segment displays which use chips such as WS2811 internally. Those can also be used here using a serial LED controller (see *LEDs* for details).

There is a hackaday project for monochrome serial segment displays. Furthermore, there are also full RGB serial segment displays. Both are controlled using WS2811 controllers.

You can also buy WS2811 controller with PCB in China (bulk 100 pcs) for about ten bucks solder your own display.

Color and Brightness

There is no color or brightness support for segment displays in MPF yet. Let us know if you need that. However, you can control both using normal light shows.

Config

This is an example:

```
hardware:
  segment_displays: light_segment_displays

lights:
  segment1_a:
    number: 1
  segment1_b:
    number: 2
  segment1_c:
    number: 3
  segment1_d:
    number: 4
  segment1_e:
    number: 5
  segment1_f:
    number: 6
  segment1_g:
    number: 7
  segment2_a:
    number: 8
  segment2_b:
    number: 9
  segment2_c:
    number: 10
  segment2_d:
    number: 11
  segment2_e:
    number: 12
  segment2_f:
    number: 13
  segment2_g:
    number: 14

segment_displays:
  display1:
    number: 1
    platform_settings:
      lights:
        - a: segment1_a
          b: segment1_b
          c: segment1_c
          d: segment1_d
          e: segment1_e
          f: segment1_f
          g: segment1_g
```

(continues on next page)

(continued from previous page)

```
      - a: segment2_a
        b: segment2_b
        c: segment2_c
        d: segment2_d
        e: segment2_e
        f: segment2_f
        g: segment2_g
    type: 7segment
```

Here is another example for a monochrome serial 16-segment display using a WS2811 controller on OPP:

```
hardware:
  segment_displays: light_segment_displays

lights:
  l_neoseg_0_0_a:
    start_channel: 0-0-60  #When using other RGB pixels in the chain before the display,
                           #             start_channel = 3 x start_pixel
                           #Using RGBW,  start_channel = 4 x start pixel
                           #Here, there are 20 RGB neopixels before the display
    type: w
    subtype: led
  l_neoseg_0_0_m:
    previous: l_neoseg_0_0_a
    type: w
    subtype: led
  l_neoseg_0_0_k:
    previous: l_neoseg_0_0_m
    type: w
    subtype: led
  l_neoseg_0_0_h:
    previous: l_neoseg_0_0_k
    type: w
    subtype: led
  l_neoseg_0_0_u:
    previous: l_neoseg_0_0_h
    type: w
    subtype: led
  l_neoseg_0_0_s:
    previous: l_neoseg_0_0_u
    type: w
    subtype: led
  l_neoseg_0_0_t:
    previous: l_neoseg_0_0_s
    type: w
    subtype: led
  l_neoseg_0_0_g:
    previous: l_neoseg_0_0_t
    type: w
    subtype: led
  l_neoseg_0_0_f:
    previous: l_neoseg_0_0_g
    type: w
```

(continues on next page)

(continued from previous page)

```
    subtype: led

  l_neoseg_0_0_e:
    previous: l_neoseg_0_0_f
    type: w
    subtype: led
  l_neoseg_0_0_dp:
    previous: l_neoseg_0_0_e
    type: w
    subtype: led
  l_neoseg_0_0_d:
    previous: l_neoseg_0_0_dp
    type: w
    subtype: led
  l_neoseg_0_0_r:
    previous: l_neoseg_0_0_d
    type: w
    subtype: led
  l_neoseg_0_0_p:
    previous: l_neoseg_0_0_r
    type: w
    subtype: led
  l_neoseg_0_0_c:
    previous: l_neoseg_0_0_p
    type: w
    subtype: led
  l_neoseg_0_0_n:
    previous: l_neoseg_0_0_c
    type: w
    subtype: led
  l_neoseg_0_0_b:
    previous: l_neoseg_0_0_n
    type: w
    subtype: led
  l_neoseg_0_0_na:
    previous: l_neoseg_0_0_b
    type: w
    subtype: led

segment_displays:
  display1:
    number: 1
    platform_settings:
      lights:
        - a: l_neoseg_0_0_a
          b: l_neoseg_0_0_b
          c: l_neoseg_0_0_c
          d: l_neoseg_0_0_d
          e: l_neoseg_0_0_e
          f: l_neoseg_0_0_f
          g: l_neoseg_0_0_g
          h: l_neoseg_0_0_h
          k: l_neoseg_0_0_k
          m: l_neoseg_0_0_m
```

(continues on next page)

(continued from previous page)

```
        n: l_neoseg_0_0_n
        p: l_neoseg_0_0_p
        r: l_neoseg_0_0_r
        s: l_neoseg_0_0_s
        t: l_neoseg_0_0_t
        u: l_neoseg_0_0_u
    type: 16segment
```

What if it did not work?

Have a look at our *hardware troubleshooting guide*.

Trinamic's StepRocker

Related Config File Sections
hardware:
trinamics_steprocker:
steppers:

Connect the StepRocker to USB and MPF can control any steppers connected to it.

TODO: Add a picture of a StepRocker

This is an example:

```
hardware:
  platform: virtual
  stepper_controllers: trinamics_steprocker

trinamics_steprocker:
  port: /dev/ttyACM0

steppers:
    # Scenario: 1.8 degree stepper attached to a 7:1 gear ratio with homing flag that you want to␣
↪control in units of revolutions
  positionStepper:
    number: 0
    homing_direction: clockwise        # when facing the shaft
    homing_mode: hardware
    reset_position: 0
    reset_events: test_reset
    named_positions:
      0.0: test_00
      0.6: test_01
      1.0: test_10
    platform_settings:
      move_current: 25                 # percent
      hold_current: 5                  # percent
      homing_speed: 0.1                # user units/sec
      microstep_per_fullstep: 16       # 1/16 mode (1 step = 1/16 of a full step)
      fullstep_per_userunit: 1400      # UU=1 Revolution = 200 full steps per rev (1.8 deg␣
↪stepper) * 7 gear ratio
```

(continues on next page)

(continued from previous page)

```
    velocity_limit: 0.5                   # user units/sec   (so, 0.8 RPS of output gear )
    acceleration_limit: 2.0               # user units/sec^2  (so, 2 RPS^S of output gear)
```

Overview video about *steppers*:

https://youtu.be/YaRNBU0OHGc

What if it did not work?

Have a look at our *hardware troubleshooting guide*.

How to use Step Stick Steppers in MPF

Related Config File Sections
hardware:
step_stick_stepper_settings:
digital_outputs:
switches:
steppers:

MPF can drive steppers on a StepStick (or DRV8825) connected via a *digital output*. Depending on the jitter of the output the speed might be limited to a few steps per second (like 50-200).

TODO: Add a picture of a step stick or DRV8825

Overview video about *steppers*:

https://youtu.be/YaRNBU0OHGc

Configuring your stepper

A step stick stepper needs two or three outputs which define the hardware number: `direction_output:step_output` or `direction_output:step_output:enable_output`. In addition, you need a `homing_switch` so MPF can find the `0` position of your stepper at startup.

This is an example:

```
#config_version=5
hardware:
  stepper_controllers: step_stick
digital_outputs:
  c_direction:
    number: 1
    type: driver
  c_step:
    number: 2
    type: driver
  c_enable:
    number: 3
    type: driver
```

(continues on next page)

(continued from previous page)

```
switches:
  s_home:
    number: 1
steppers:
  stepper1:
    number: c_direction:c_step:c_enable   # enable is optional
    homing_mode: switch
    homing_switch: s_home
    named_positions:
      10: test_00
      20: test_01
      50: test_10
    platform_settings:  # optional speed settings
      low_time: 20ms
      high_time: 20ms
```

You might want to change the speed in the *platform_settings* section. `1000 / (low_time + high_time)` will be your number of steps per second.

Connecting your stepper driver

Connect the `DIR` pin to your `direction_output`, `STP` to your `step_output` and `GND` to your ground. If use an `enable_output` connect it to `EN`. Otherwise, pull it to `GND` or the driver will not work. Connect `SLP` and `RST` to `VDD` (not all driver have all of them). In addition, you need to pull `M0`, `M1` and `M2` to VDD or GND to configure the step resolution. Your stepper will connect to `1A`, `1B`, `2A` and `2B`. Connect power to `VMOT` (do not forget to also connect ground of your stepper power supply; see *Voltages and Power*). See the datasheet for details about your driver.

What if it did not work?

Have a look at our *hardware troubleshooting guide*.

Choosing a PC for MPF

In addition to picking a pinball controller platform, you also need to decide what type of host computer you'll use. (By "host computer," we're talking about the computer that will run MPF which will talk to the pinball controller via USB.) There are lots of host computer options, including small single-board computers, laptops, small-form factor x86 motherboards, etc. You're also going to have to decide on what OS you use (Windows, Linux, or Mac).

Generally speaking, MPF will run on any PC or embedded system which can run Python 3. In most cases you also need a graphics card with working OpenGL to run the MPF Media Controller (MPF-MC). Most operating systems work fine (we test on Linux, Windows, Mac OS X) but be careful with virtualized environments because OpenGL might not work perfectly.

What kind of performance is required?

One of the biggest things that will affect your choice of host computer will be the performance you need. Obviously the host computer has to "keep up" with your game, so if you pick an under-powered host computer then your game loop can slow down and you'll have issues. The computing needs of a pinball machine are actually pretty small. The core game, modes, ball tracking, dealing with switches, etc.—-all of that can probably be done on a very tiny computer. The real driver these days is your video and graphics. If you have a hi-def LCD window with lots of full video and layers and on-screen elements all blended together, then you're going to need a "real" computer to drive it and will not be happy with a small single-board computer.

CPU

The trend in computing these days (for both "real" computers and small single-board computers) is multi-core. Almost every computer these days has a dual-core or quad-core processor.

MPF uses two processes (one for the game engine and one for the media controller), so it can make use of a dual-core system. However there is probably not much benefit to MPF running on machines with more than 2 cores (other than it frees up more cores for other non-MPF things.) During startup, when playing sound or loading assets additional cores may be used. Therefore, we recommend a CPU with at least two cores. MPF certainly benefits from four cores but everything above that will not help during normal games. However, during development, when using MPF Monitor and an IDE more cores will certainly help.

Disk

Disk space it not really an issue these days. The real question is disk performance in terms of SSD versus traditional spinning magnetic hard disks. SSD is fast, you can can get away with less memory since MPF can dynamically load and unload assets. To load assets quickly a SSD helps. You definitely want that during development but you might use a cheaper option (such as a SD-card) for the final game.

Filesystems can become corrupted by unsafe shut downs, so consider running a journaling filesystem or even mount them read-only.

Memory

MPF itself doesn't require much memory. The real memory use comes from loading all the images, sounds, and videos into memory. MPF can load those on demand (or automatically when a mode starts, and unload them when the mode ends). This works well if you have a fast disk (SSD).

However, if you have enough memory, MPF can pre-load everything when it starts. This will increase the startup time of your machine, but will make it so that everything runs fast once its booted.

Note that 32-bit OSes only allow individual applications to access 2GB of memory, so if you have 6 gigs of assets and want to buy an machine with 8GB of RAM, you need to run a 64-bit OS. (MPF supports both 32-bit and 64-bit systems. If you run on 64-bit, make sure you also get the 64-bit version of Python.)

MPF needs at least 512MB RAM but we recommend 2-4GB depending on the amount of assets. Again, during development you want to have more RAM (8GB+) for your IDE and other tools.

Development setup

- CPU with at least four cores

- 8GB RAM or more
- SSD

Final game

We cannot emphasize this enough: Do not use such a setup for game development.

- CPU with two to four cores
- 2-4GB RAM (mostly for assets)
- SD-Card/Embedded flash/SSD

See also the *discussion about the hardware in your final game*.

How to use native I2C on Linux (SMBUS2)

Related Config File Sections
hardware:

MPF can control I2C devices on Linux using the Python smbus2_asyncio extension.

1. Install the smbus2_asyncio extension

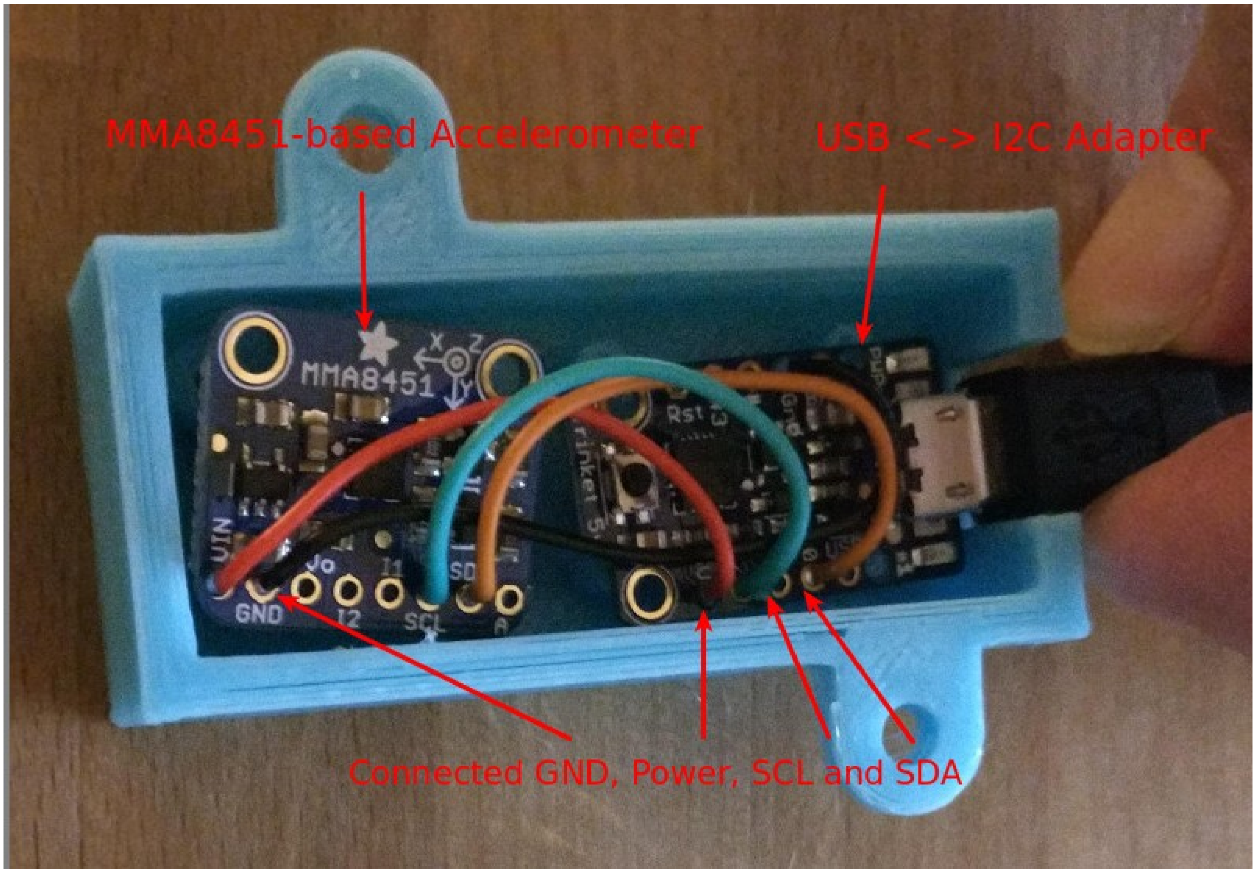
Install `smbus2_asyncio` via pip:

```
pip3 install smbus2_asyncio
```

2. Figure out which bus to use

- Some boards such as the Raspberry Pi have native I2C buses. Figure out which bus to use and make sure MPF has sufficient permissions to use it (Alternatively, you can also controll the I2C on the RPi remotely using the *RPi* platform).
- You can build an adapter to tap I2C out of a spare VGA, DVI or HDMI port: http://www.instructables.com/id/Worlds-Cheapest-I2C-I-Squared-C-Adapter/
- Commercial USB-I2C adapters exist but are usually very expensive
- You can build your own USB-I2C adapter. Hardware can be bought ready-made for less than 10 bucks. Atiny85 based boards can be bought at Adafruit as Trinket (and elsewhere just google it).

This is an adafruit trinket used as USB-I2C adapter for an *MMA8451-based accelerometer*:



3. Connect your hardware

Connect the hardware to the bus. This will be at least SDA, SCL and ground. Usually, you have to power your device somehow and in a lot of cases this power can be provided from the controller.

4. Set your I2C devices to use the "smbus2" platform

Next you need to configure I2C in MPF to use the `smbus2` platform. By default, all types of devices are assumed to be using the same platform that you have set in the *hardware:* of your machine config file. So if your platform is set to `fast`, MPF assumes your I2C devices are connected to a FAST controller, and if your platform is set to `p3_roc`, MPF assumes your I2C devices are connected to the P3-Roc board.

To configure MPF to use native I2C, you can add an entry to the `hardware:` section of your machine config to tell it to override the default platform for your I2C devices and to instead use the `smbus2` platform, like this:

```
hardware:
  i2c: smbus2
```

See the *Mixing-and-Matching hardware platforms* guide for more information about setting device-specific default platforms versus overriding the platform for individual devices.

5. Understanding I2C numbering

When using I2C addresses in I2C devices smbus2 will interpret those as bus-address. If you only provide an address it will use bus 0. On Linux bus 0 will ususally be /dev/i2c-0, 1 will be /dev/i2c-1 and so on.

6. Add udev rules if you have multiple i2c devices

If you have more than one i2c device connected to your PC via USB you can assign a name to your ports based on the USB port they are connected to.

First identify the port of your I2C hardware. Usually it should be `/dev/i2c0` or `/dev/i2c1`.

Then run `udevadm info` on your port:

```
udevadm info /dev/i2c0
```

This will show you the `DEVPATH`. Now replace the last part `i2cX` with an asterisk and add an udev rules like this in `/etc/udev/rules.d/i2c.rules`:

```
SUBSYSTEM=="i2c-dev", ACTION=="add", DEVPATH=="/devices/pci0000:00/0000:00:14.0/usb1/1-3/1-3.1/1-3.1:1.
↪0/*", SYMLINK+="i2c-front", GROUP="adm", MODE="0660
```

After a reboot you should get a `/dev/i2c-front` device if you connect an i2c device to that specific USB port. You can use that port in your config.

What if it did not work?

Have a look at our *hardware troubleshooting guide*.

Raspberry PI (pigpio)

Related Config File Sections
hardware:
raspberry_pi:
switches:
coils:
servos:

The rpi platform can be used to control inputs (switches), outputs (coils), I2C and servos on the RPi remotely (or locally) using pigpio.

Video about the Raspberry PI and MPF:

https://youtu.be/ihj5O0J-mD0

Installation

You need to install the `apigpio` extension via pip to use it:

```
pip3 install apigpio_mpf
```

The pigpiod service needs to be running (in this example on localhost port 8888, which is the default setting). To install it and enable is (on debian based systems):

```
apt install pigpiod
systemctl enable pigpiod.service
systemctl start pigpiod.service
```

The *enable* step gets the service running at startup, thus it is optional.

Using pigpio via network

If you want to use your RPi over ethernet you have to edit `/lib/systemd/system/pigpiod.service` and change `ExecStart=/usr/bin/pigpiod -l` to `ExecStart=/usr/bin/pigpiod`. This is not needed if you run MPF on the RPi itself. Make sure your Raspberry PI is not accessible from the internet and the network is segmented properly.

Config

This is an example config:

```
hardware:
  platform: rpi
raspberry_pi:
  ip: localhost
  port: 8888
switches:
  s_switch_8:
    number: 8
  s_switch_7:
    number: 7
coils:
  output_2:
    number: 2
    default_pulse_ms: 1000
servos:
  servo_26:
    number: 26
```

Configure the ip of your RaspberryPi in the `raspberry_pi` section. You may use localhost if you are running MPF on the RPi. Any pin on the RPi can be used as either input or output. Additionally, you may use servos on any pin.

Available GPIOs

You check GPIO locations on your RPi at pinout.xyz. Please note that you have to use the Broadcom GPIO numbers instead of the pin numbers. Those slightly differ between different RPi models. If you get permission errors in your MPF log this is usually because you used a GPIO number which does not exist on your hardware.

Is this a real pinball controller?

No. The RPi is not a pinball controller for various reasons:

- Drivers are missing to drive coils
- Inputs are unprotected and any error current will fry the CPU
- Hardware rules are not supported by the `pigpio` daemon
- A watchdog is missing

This platform is meant as a cheap interface for peripherals such as DMDs, segment displays lights, servos, steppers and more. You can also use it for inputs to some extend.

Can this be turned into a pinball controller?

Sure it can. We just did not do that here. Have a look at *Arduino Pinball Controller* which is kind of that already.

If you want to do it with pigpio you would have to do the following (and probably more):

- Build a PCB with FETs to drive outputs. Add proper protection.
- Protect your inputs against high and negative voltages.
- Implement hardware rules in `pigpio` (might be possible with callbacks)
- Run a realtime linux for proper timing of your rules
- Add a some watchdog (either in Linux or in hardware)

What if it did not work?

Have a look at our *hardware troubleshooting guide*.

MMA8451-based accelerometers

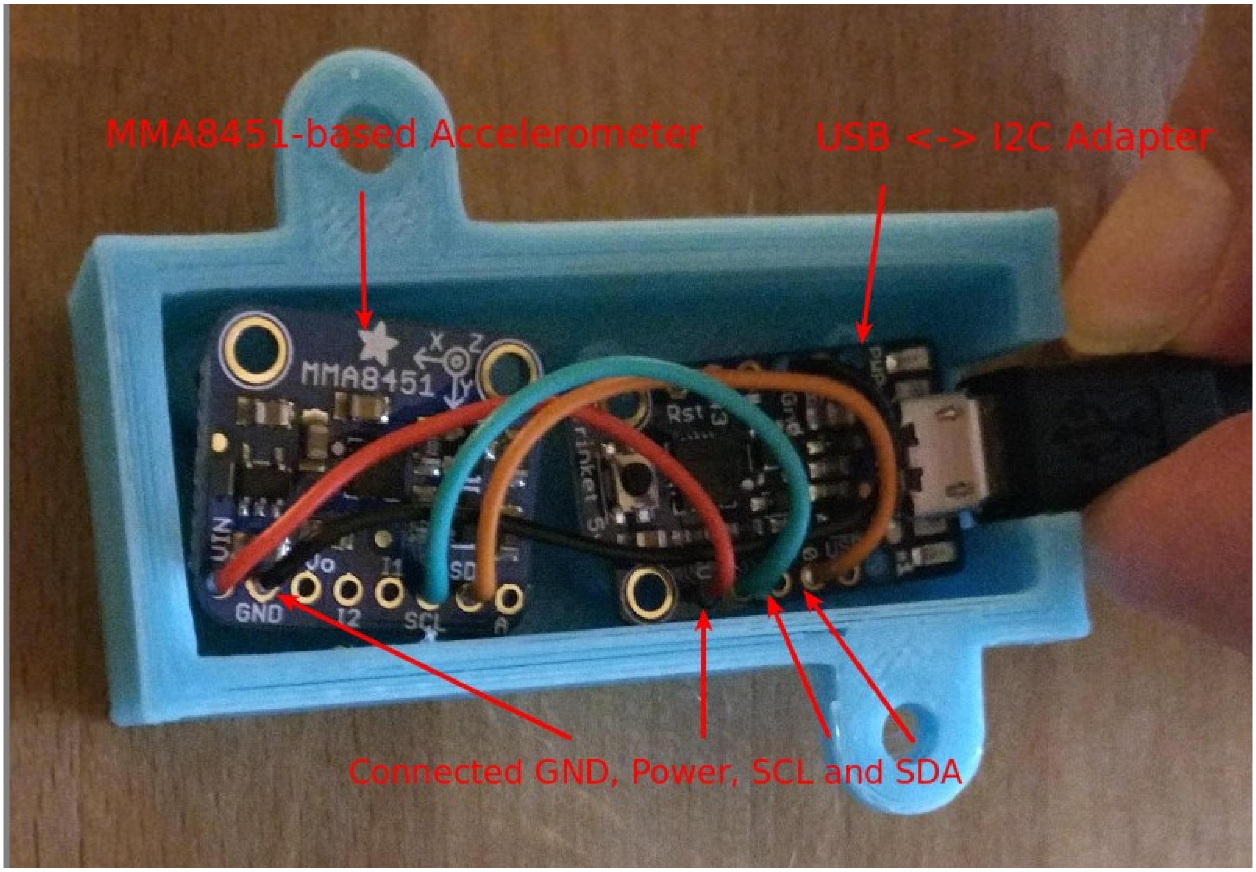
Related Config File Sections
hardware:
accelerometers:

This chips can be connected to I2C and act a tilt and leveler. Available on adafruit (and elsewhere).

Configure using:

```
hardware:
  accelerometers: mma8451
accelerometers:
  my_accelerometer:
    level_x: 0
    level_y: 0
    level_z: 1
    number: 1-29
```

This will configure an MMA8451 on I2C bus 1 with address 0x1D (29 decimal which is the default for this device). The exact numbering depends on your i2c platform.



The device in the picture is using *smbus on linux* as i2c platform with an Atiny85-based I2C-USB adapter.

What if it did not work?

Have a look at our *hardware troubleshooting guide*.

How to use SPI Bit Bang in MPF

Related Config File Sections
spi_bit_bang:
digital_outputs:
switches:

Sometimes you want to read switches from PCBs which contain a shift register or SPI chip (i.e. a 74HCT165). This platforms uses two *digital_outputs:* and one *switches:* on another platform to address the SPI chip. Please note that this is relatively slow compared to platforms which interface to SPI natively (such as *How to use MPF with Stern SPIKE / SPIKE 2 machines*). The main purpose of this platform is to access Stern Spike boards using other control systems than Spike. Primarily, this allows you to use the *Spike Trough* on any system. However, if you are on Spike or any other platform which natively reads switches via SPI use those means since they are much more efficient.

This is an example:

```
hardware:
  platform: your_platform, spi_bit_bang      # add your platform first here
spi_bit_bang:
  miso_pin: s_miso
  cs_pin: o_cs
  clock_pin: o_clock
  bit_time: 50ms
  inputs: 8
digital_outputs:
  o_cs:
    number: 1
    type: driver
  o_clock:
    number: 2
    type: driver
switches:
  s_trough_0:
    number: 0
    platform: spi_bit_bang
  s_trough_1:
    number: 1
    platform: spi_bit_bang
  s_trough_2:
    number: 2
    platform: spi_bit_bang
  s_trough_3:
    number: 3
    platform: spi_bit_bang
  s_trough_4:
    number: 4
    platform: spi_bit_bang
  s_trough_5:
    number: 5
    platform: spi_bit_bang
  s_trough_6:
    number: 6
    platform: spi_bit_bang
  s_trough_7:
    number: 7
    platform: spi_bit_bang
  s_miso:
    number: 10
```

The refresh rate of your platform will be `bit_time / (inputs + 2)`. For instance 8 inputs with 50ms `bit_time` will result in 2Hz update rate which is not terribly good.

`bit_time` determines how long MPF will wait after clocking the chip for `miso_pin` to settle. Depending on your platform it might this might need a while. Especially if your platform is connected via USB because of USB latency and jitter. If your inputs are local (i.e. on a RPi) this might be very short compared and you might be able to achieve 50Hz. At the default 2Hz you will wait in average 250ms for a switch change and 500ms in the worst case. Have that in mind.

What if it did not work?

Have a look at our *hardware troubleshooting guide*.

How to use MPF with OSC Devices or Hardware

Related Config File Sections
hardware:
osc:
switches:
lights:

MPF can use the Open Sound Control (OSC) to interface with other software or hardware devices. As OSC messages are not standardized we define a few custom messages:

Incoming:

- `/sw/switch_name` with the state `True` or `False` as parameter to set the state of an OSC switch in MPF
- `/event/event_name` with parameters in the form `key1, value1, key2, value2, ...` to post events to MPF.

Outgoing:

- `/light/light_name/color` with the brightness of the color as float (0-1).
- `/event/event_name` with parameters in the form `key1, value1, key2, value2, ...` for all events configured in `events_to_send` in your `osc` config section.

This is an example:

```
hardware:
  platform: osc

osc:
  remote_ip: 127.0.0.1
  remote_port: 8000

  events_to_send:
    - player_score
    - some_non_osc_switch_active
    - some_non_osc_switch_inactive

lights:
  test_light1:
    channels:
      red:
        - number: light1/red
      blue:
        - number: light1/blue
      green:
        - number: light1/green
  test_light2:
    number: light2
```

(continues on next page)

(continued from previous page)

```
switches:
  switch_1:
    number: 1
  switch_2:
    number: 2
  some_non_osc_switch:  # not an OSC switch but used for the events above
    number: 23
    platform: virtual
```

You need to install python-osc to use the OSC platform:

```
pip3 install python-osc
```

What if it did not work?

Have a look at our *hardware troubleshooting guide*.

Using MPF without physical hardware

It's possible to run MPF even if you don't have a physical pinball machine attached to your computer. This is great if you're just starting out, or if you want to work on your MPF config when you're not around your pinball machine.

MPF achieves this through "virtual" platform interfaces, of which there are two options:

Video about developing your game without hardware:

https://youtu.be/7XmIIhzEREk

Note that if you want to use MPF without a physical pinball machine, you probably also want to use the *MPF Monitor* which is a graphical tool that lets you interact with lights, switches, and pinball mechs in MPF which works nicely with the smart virtual platform.

The "Smart Virtual" Platform

Related Config File Sections
hardware:
smart_virtual:

MPF's *Smart Virtual Platform* is based on the *virtual platform* with one key difference: The Smart Virtual platform watches for coil pulse events and adjusts switches in response to simulate how those switches would have changed if that coil fired on real hardware.

To understand why the smart virtual platform exists, consider this simple machine config for a trough, a plunger lane, and keyboard key mappings to simulate their switches:

```
switches:
  s_trough1:
    number: s31
```

(continues on next page)

(continued from previous page)

```
  s_trough2:
    number: s32
  s_trough3:
    number: s33
  s_trough4:
    number: s34
  s_plunger_lane:
    number: s27
coils:
  c_trough_eject:
    number: c01
    default_pulse_ms: 25
  c_plunger_eject:
    number: c03
    default_pulse_ms: 25
ball_devices:
  bd_trough:
    tags: trough, home, drain
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4
    eject_coil: c_trough_eject
    eject_targets: bd_plunger
  bd_plunger:
    ball_switches: s_plunger_lane
    eject_coil: c_plunger_eject
playfields:
  playfield:
    default_source_device: bd_plunger
    tags: default
keyboard:
  1:
    switch: s_trough1
    toggle: true
  2:
    switch: s_trough2
    toggle: true
  3:
    switch: s_trough3
    toggle: true
  4:
    switch: s_trough4
    toggle: true
  p:
    switch: s_plunger_lane
    toggle: true
```

MPF's regular virtual platform interface is "dumb" in the sense that all switch actions need to be controlled externally (either via keyboard keys, the OSC interface, etc.)

So if you have the above configuration and then MPF wants to eject a ball from the trough, it will fire the trough coil but the switches won't actually change. (In fact this will cause MPF to think that the eject failed, because it will fire the eject coil and not see the ball leave.)

If you wanted to "play" an MPF game with the example config above, you'd have to manually simulate the ball leaving the trough by hitting the "1" key to deactivate a trough switch, and then hitting the "P" key to activate the plunger lane switch. (And you'd have to do this fast enough for the eject failure detection not to kick in.)

A better solution? The "smart" virtual interface.

In order to address these challenges, MPF includes a smart virtual platform interface. The smart virtual interface works by watching for coil pulse commands. If it sees a coil pulse from a coil that's configured in a mechanism that would ordinarily cause a switch to change state, then it will automatically change that switches state.

For example, if you have the trough config from above and the trough's eject coil fires, the smart virtual platform will look to see if there are any balls in that device, and, if so, simulate the ball leaving (which could be by deactivating one of the device's ball switches).

The smart virtual platform also knows (thanks to the *eject_targets:* ball device setting) where the ball is ejected to, so when a ball is ejected from a device, the smart virtual platform will also simulate the ball going into the target ball device.

Going back to the example machine config above, if the smart virtual platform interface is being used, when a game is started, you'll see the *s_trough1* switch automatically deactivate in response to the trough coil pulsing, and then 100ms later you'll see the *s_plunger* switch activate to simulate a ball going into the plunger lane. So simply starting a game with the smart virtual platform puts the ball in the plunger lane without you having to mess with the "1" and "P" keys.

Using the smart virtual platform

There are three ways you can use the smart virtual platform:

1. No platform setting

If you do not have a `platform:` setting in your machine config's `hardware:` section (or if you don't have a `hardware:` section, then MPF will use the smart virtual platform anyone you run it.

2. Manually setting the platform

You can also manually specify the smart virtual interface in the machine config, like this:

```
hardware:
  platform: smart_virtual
```

3. Via the command line

You can also specify the smart virtual platform interface via the `-X` (uppercase *X*) from the command line, like this:

```
mpf -X
```

Or

```
mpf both -X
```

etc.

What does the smart virtual platform do?

The smart virtual platform currently simulates the following pinball mechanisms. You can configure some of them in the *smart_virtual section*.

Ball Devices

If a ball device's eject coil is pulses, it will simulate a ball leaving that device (as long as that device has at least one ball). It is smart enough to know how many balls are in a device, and works with special scenarios (such as timed entrance switches that are only active when the device is full and eject confirmation switches).

It will also simulate a ball entering the target device when a ball is ejected, and again it knows how to work with various ball switch and entrance switch combinations.

Drop Targets

The smart virtual platform will reset drop target switches if their associated reset coil is pulsed.

The Virtual Platform

Related Config File Sections
hardware:

MPF's virtual platform interface is a software-only platform you can use if you don't have a physical pinball controller attached.

Note: MPF also has a *smart virtual platform* which is probably what you'd use in most cases instead of the virtual platform.

Note for P-ROC and P3-ROC users: P-ROC's pyprocgame includes a virtual P-ROC interface called *FakePinPROC*. We don't use that in the MPF because doing so requires that pyprocgame is installed, and it's likely that people using MPF won't have pyprocgame. Using MPF's virtual hardware interface is conceptually similar to *FakePinPROC*.

Using the virtual platform

There are three ways you can use the virtual platform:

1. Manually setting the platform

You can manually specify the virtual platform in the machine config, like this:

```
hardware:
  platform: virtual
```

3. Via the command line

You can also specify the smart virtual platform interface via the `-x` (lowercase *X*) from the command line, like this:

```
mpf -x
```

Or

```
mpf both -x
```

etc.

The Virtual Pinball (VPX) Platform

Related Config File Sections
hardware:

VPX can be used (on Windows) to emulate the hardware of a pinball machine to test your game without real hardware. For instructions to download and install Visual Pinball please see here.

This can be useful for software and hardware development. To use the VPX platform you need to install it, along with the MPF-VPX bridge and add some VPX scripts to your VPX table.

The bridge will connect to the running MPF machine when you start your VPX table. As the VPX table is used only to emulate the hardware and should not contain any game logic.

Installation

Copy the file `register_vpcom.py` to your local machine folder. To register the bridge run a CMD shell as Administrator, then

```
python register_vpcom.py --register
```

(You can use `--unregister` to uninstall the bridge)

Use VPX in MPF

In your `config.yaml` configure virtual_pinball as your platform:

```
hardware:
  platform: virtual_pinball
```

or if you already have physical hardware configured start MPF with the `--vpx` commandline option (similar to `-X`):

```
mpf both --vpx
```

Configure VPX

In VPX you need to adjust your script to talk to MPF. You can also looks this up in the example project inside the bridge repository. The GameName set in VPX is not used to check or validate the MPF machine.

Setup controller and timers

- add `Set Controller = CreateObject("MPF.Controller")`
- add a Timer `MPFTimer` with an interval of 10 to 50ms. Keep this well below the minimal `default_pulse_ms` set in MPF for solenoids
- add a `Sub MPFTimer_Timer` to update all the lights and solenoids

In Table_Init

- call Controller.run
- set the Trough switch(es) or create balls for a physical Trough (as in the demo table)
- init NC switches to False

In Drain_Hit (for 1-Ball games)

- set the Trough switch(es)

To use autofire_coils (Slings, Pops etc)

- add a droppable wall to each bumper. This will be used in `Sub CheckAutofireCoils` to disable/enable the bumper
- if necessary place an invisible wall behind the slingshots to stop the ball once the slingshots are disabled
- add each autofire object in VPX to `Sub CheckAutofireCoils`
- use the normal `Slingshot_Slingshot` and `Bumper_Hit` events

Switch Handling

- add `Controller.Switch(MPFSwitchNumber)=SwitchState` to the `Switch_Hit/_Unhit` events. Use "" to include string type numbers
- add `Controller.PulseSW(MPFSwitchNumber)` to the `Switch_Hit` event of targets, slingshots and bumpers

Controlled Lights (all types)

- create a collection "ControlledLamps"
- assign all controlled VPX lamps (Matrix, GI, LED, Flashers) to the collection "ControlledLamps" and call `InitLamps`

LEDs and Lamps

- in `Sub UpdateLamps` add a case for every MPF light and LED number, setting the state of the VPX lamp/LED. Use "" to include string type numbers

GI strings

- create a collection for each GI string, assign the GI lamps to the collections as needed
- assign all GI lamps to the collection "ControlledLamps" and call‘‘ InitLamps‘‘
- in `Sub UpdateGI` add a case for every MPF gi string number, setting the state of the VPX GI collection. Use "" to include string type numbers

Flashers

- assign all flasher lamps to the collection "ControlledLamps" and call `InitLamps`
- in `Sub UpdateFlashers` add a case for every MPF flasher number, setting the state of the VPX flasher. Use "" to include string type numbers

Solenoids

- add all normal solenoids to the `Sub InitSolenoids`, to initialize them as `False`
- in `Sub UpdateSolenoids` add a case for every MPF coil number, setting the state of the VPX solenoid. Use "" to include string type numbers

Flippers

- add the Flipper routines (`Solenoids` and `KeyUp/KeyDown`) as in the demo table. Flippers are handled as autofire coils and can be enabled/disabled using hrdware rules.

To run a game

1. start VPX as Administrator
2. start MPF, wait until the display has been initialized
3. start VPX table

To exit a game shut down the VPX table first

What if it did not work?

Have a look at our *hardware troubleshooting guide*.

Connecting Your Computer Keyboard to MPF Switches

Related Config File Sections
keyboard:
switches:

The MPF media controller includes a keyboard interface which allows you to interact with your running machine via a computer keyboard. In most cases you'd use this to simulate pinball switch events via keys on your keyboard, but you can also post MPF events via keyboard presses. You can map single key presses or combinations of keys, and you can use the keyboard module with or without a physical pinball machine connected to your computer.

To use the keyboard interface, you add a *keyboard:* section to your machine configuration file and then create a list which maps keyboard keys to pinball machine switch names or MPF events. Then when you press a key on the keyboard, the switch controller receives that event and sends it to the game. The keyboard module tracks both key-down and key-up events, so you can hold down a key to represent a ball sitting on a switch. You can also set several options for each key, including:

- Specify that a key is a "toggle" key, meaning the switch stays in the state even after you let go of the key. (In other words, tap the key once to activate the switch. Tap it again to deactivate it.) This is helpful for things like your trough or ball locks where you want to simulate a ball sitting on a switch but you don't want to play a crazy game of keyboard Twister where you're trying to hold down all these keys at once.
- Specify that a key is inverted, so pressing (or holding) the keyboard key deactivates the switch, and releasing it activates the switch. (Note this is *not* needed to compensate for normally-closed switches, as the switch controller handles that automatically. This is just is you want to invert the computer's keyboard action.)
- Specify combo keys, so you can set up one switch action for the `` S``key, a different one for ``CTRL+S ``, another one for `SHIFT+S`, etc.

Note that you can also use the *MPF Monitor* for this. However, often it is faster to use the keyboard to change switch states. You can also use the MPF monitor and your keyboard in tandem. Most people use keyboard mappings to change balls in troughs for example.

Additionally, the `keyboard:` section is nice for posting ad-hoc and debug events. For instance, it can be very useful to be able to start modes using the keyboards when you are testing them if it is nontrivial to start them.

Here's an example of it in action:

```
keyboard:
  z:
    switch: left_flipper
  slash:
    switch: right_flipper
  s:
    switch: start
```

(continues on next page)

(continued from previous page)

```
1:
  switch: trough1
  toggle: true
2:
  switch: trough2
  toggle: true
shift+p:
  switch: lock_post
  invert: true
q:
  event: machine_reset
ctrl+shift+4:
  event: advance_reel_test
  params:
    reel_name: score_1p_10
    direction: 1
```

You can also read more about the `keyboard:` section in the *Tutorial step 6: Add keyboard control* documentation.

Key & key combination entries

Once you create your `keyboard:` section, you create subsections for each key or key combination you want to configure. For simple keys (without modifiers), you can just enter the key. (In the sample file above, this is `z`, `s`, `1`, `2`, `q`, and `4`.)

These entries are not case sensitive.

Using special keys

For "special" keys, it's probably just easiest to enter the keys as words. Here are some examples of words that map to keys:

- equals
- minus
- dash
- leftbracket
- rightbracket
- backslash
- apostrophe
- semicolon
- colon
- comma
- period
- slash

- question

Note that you can't use the Escape key because that's currently hard-coded to exit out of MPF when you hit it.

Note that this keyboard interface focuses on keys, not symbols. In other words the "plus" key is if you have a full size keyboard with a number pad which has a dedicated plus key. If you're using a laptop with the shared plus & equals key, that is the equals key, or the equals key with a shift modifier.

Adding SHIFT, CTRL, and ALT modifiers

Since there are probably more switches in your machine then there are keys on your keyboard, you can also specify key combinations along with the key entries. These are called "modifier keys," and MPF supports them in combination with regular keys, like this:

```
  t:
    switch: foo
  shift-t:
    switch: tilt
  shift+ctrl+t:
    switch: slam_tilt
```

Starting in MPF 0.33, you an add `debug: true` in the `keyboard:` section to get a printout on the console of the current key and/or modifiers that are pushed down which is helpful in figuring out exactly what the modifier keys are called on your system.

Use it like this:

```
keyboard:
   debug: yes
```

This will print out results live as you hit keys and combinations which will look something like this:

```
KEYS: d
KEYS: s
KEYS: shift
KEYS: shift+s
KEYS: f
KEYS: super
KEYS: meta+c
KEYS: shift
KEYS: shift+d
KEYS: lctrl
KEYS: ctrl+f
KEYS: escape
```

What if it did not work?

Make sure debug: true is set under keyboard as described above.

Look at your log files to see what your key strokes are.

It is possible that numlock key is on by default (especially with a laptop that does not have dedicated numlock key and running Windows).

You might see something like this:

```
Keyboard : Processing key stroke for key s-numlock
Keyboard : Processing key stroke for key s-numlock
```

If that is the case you may have to edit your computer's registry or run powershell to turn off numlock.

Virtual Segment Display Emulator

Related Config File Sections
hardware:
segment_displays:
segment_display_player:
widgets:

MPF's virtual segment display emulator is supported by the virtual and smart virtual platform interfaces in conjunction with the MPF-MC media controller. It is a software-only segment display emulator you can use if you don't have any physical segment display hardware connected.

Here is a simple example of what the display can be configured to look like:

Video about segment displays:

https://youtu.be/Jyf3jxGXnTw

For more information visit the following pages: * *How to setup and use the virtual segment display emulator* * *Segment Display Emulator widget* * *Segment Display Platforms in MPF* * *Segment Display player*

Using MPF with existing pinball machines (Williams, Stern, Gottlieb, etc.)

MPF supports all kinds of pinball machines. In this section, we highlight how to connect and configure existing machines with MPF.

- *Williams, Bally, Midway WPC, WPC-S, WPC-95*
- *Williams, Bally System 11*
- *Gottlieb System 1*
- *Gottlieb System 80*

- *Data East*
- *Stern Whitestar*
- *Stern SAM*
- *Stern SPIKE / SPIKE 2*
- *Pinball 2000*
- *Williams System 3 to 9*
- *Bally/Stern with AS-2518-17 or AS-2518-35 MPU*

See *Controlling an existing machine with MPF* for a platform machine matrix. Please let us know if you want to connect any other machine.

How to use MPF with WPC machines

You can use MPF to control existing Williams / Bally / Midway WPC, WPC-S, and WPC-95 pinball machines.

1. Connecting the physical hardware

The main options for pinball controller hardware is the *Multimorphic P-ROC* (not the P3-ROC). FAST has a WPC controller too but it never hit general availability.

In all cases, you remove the existing MPU board from the backbox of your machine and replace it with the new controller. You then connect up all the existing cables and connectors to the new controller, so in effect the new WPC controller becomes the new MPU of your machine.

A few notes:

- Both the P-ROC and the FAST WPC controller have USB connections on them, and the actual "code" that makes up MPF runs on a computer which remotely controls the pinball controller (and therefore the machine)
- Switch connectors are connected directly to the P-ROC or FAST WPC controller.
- Drivers, coils, lamps, and GI are controlled via the existing WPC power driver board (which is connected to the P-ROC or FAST WPC controller via the existing 34-pin ribbon cable).
- The existing WPC sound board in the backbox is not used, as sounds are generated via the computer running MPF. There are articles online showing how you can modify the existing sound board to add a headphone plug you can connect into the computer running MPF, though most people end up replacing the speakers with new ones and a more powerful and better sounding amp. This means you can remove the existing sound board from the backbox.
- The existing DMD, if you choose to use it, is unplugged from the WPC DMD driver board and instead plugged into a 14-pin header on the P-ROC or FAST WPC controller. This means you can remove the existing DMD driver board from the backbox.

For the P-Roc connect your machine according to the P-Roc connector mappings.

More technical information can be found in the PinWiki Williams WPC page.

2. Configuring MPF for WPC machines

In order to use MPF in a WPC machine, you need to configure the `driverboards:` section of your hardware platform.

If you're using a FAST WPC controller, it will look like this:

```
hardware:
  platform: fast
fast:
  driverboards: wpc
```

And if you're using a P-ROC:

```
hardware:
  platform: p_roc
p_roc:
  driverboards: wpc
```

Note that with the P-ROC, it is *very important* that you specify `driverboards: wpc` in your config if you're using a WPC machine. The reason for this is the P-ROC can be used to control either PD-16 (the P-ROC driver boards) or WPC driver boards, but the polarity of each type is the inverse of the other.

In other words, if you put a P-ROC in a WPC machine but specify `driverboards: pdb`, when you run MPF, it will disable all the drivers, but since the polarity is reversed, it will actually enable every driver in your machine at once. This will (1) be very loud and cause you to jump back about 10 feet, and (2) blow all your fuses.

3. Configuring switches

When using MPF with WPC machines, you can use the switch numbers from the machine's operator's manual. The exact format depends on the type of switch:

Matrix switches

Matrix switches start with the letter `S`, followed by the switch number. For example:

```
switches:
  s_left_slingshot:
    number: s41
  s_right_jet:
    number: S45
```

Note that the "S" is not case-sensitive.

Switch numbers in WPC machines correspond to the column and row, so switch "11" is column 1, row 1, switch "26" is column 2, row 6, etc. This means that there are no 0s or 9s in a standard 8x8 switch matrix.

Also, some WPC-95 machines have a 9th column in the switch matrix (meaning they'll have switch numbers 91-98). In this case, just enter those switch numbers like normal, and MPF will notice that there are switch numbers in the 90s and automatically configure the controller hardware to use the 9th column.

Our experience with using MPF with many different WPC machines is that many times, the switch numbers in the operator's manual are incorrect. (We see this in many 25% of WPC machines.) Usually it's the case where two switches have been swapped, though sometimes there are unused switches that really are used and vice-versa. So if you don't get switch activities that you expect, check out neighboring switches to see if the numbers are wrong.

Direct switches

Direct switches (which are typically the coin and front door switches) are entered with the `SD` prefix, then the number, like this:

```
switches:
  s_left_coin:
    number: sd1
  s_enter:
    number: SD8
```

Again, case doesn't matter.

Fliptronics switches

Fliptronics switches (on machines that have them) are entered with the `SF` prefix.

There are 8 Fliptronics switches on machines with Fliptronics. Typically four of them are used for flipper buttons, and four are used for EOS switches. (The flipper buttons on most Fliptronics machines actually have two switches stacked together behind each flipper button. If you push the flipper button part way in, the switch connected to the lower flipper engages, and if you push the button the rest of the way in, the switch connected to the upper flipper engages. This means if you're good, it's technically possible to flip just the lower flipper without flipping the upper one (or it's possible to hold a ball on the lower flipper while flipping the upper one).

That said, some machines needed a few extra switches for other things, and if they don't have four flippers, it's possible that the extra Fliptronics switches are used for other things.

You would use Fliptronics switches in your config like this:

```
switches:
  s_flipper_lower_right_eos:
    number: sf1
  s_flipper_lower_right:
    number: sf2
    tags: player, right_flipper
  s_flipper_lower_left_eos:
    number: sf3
  s_flipper_lower_left:
    number: sf4
    tags: player, left_flipper
```

4. Configuring coils & drivers

The drivers section of your WPC machine's operators manual will list all the driver numbers as well as the devices they're attached to. Note that WPC machines use drivers for coils, motors, and flashers.

You only enter your coils and motors in the `coils:` section of your config. Flashers go in the `flashers:` section (discussed below).

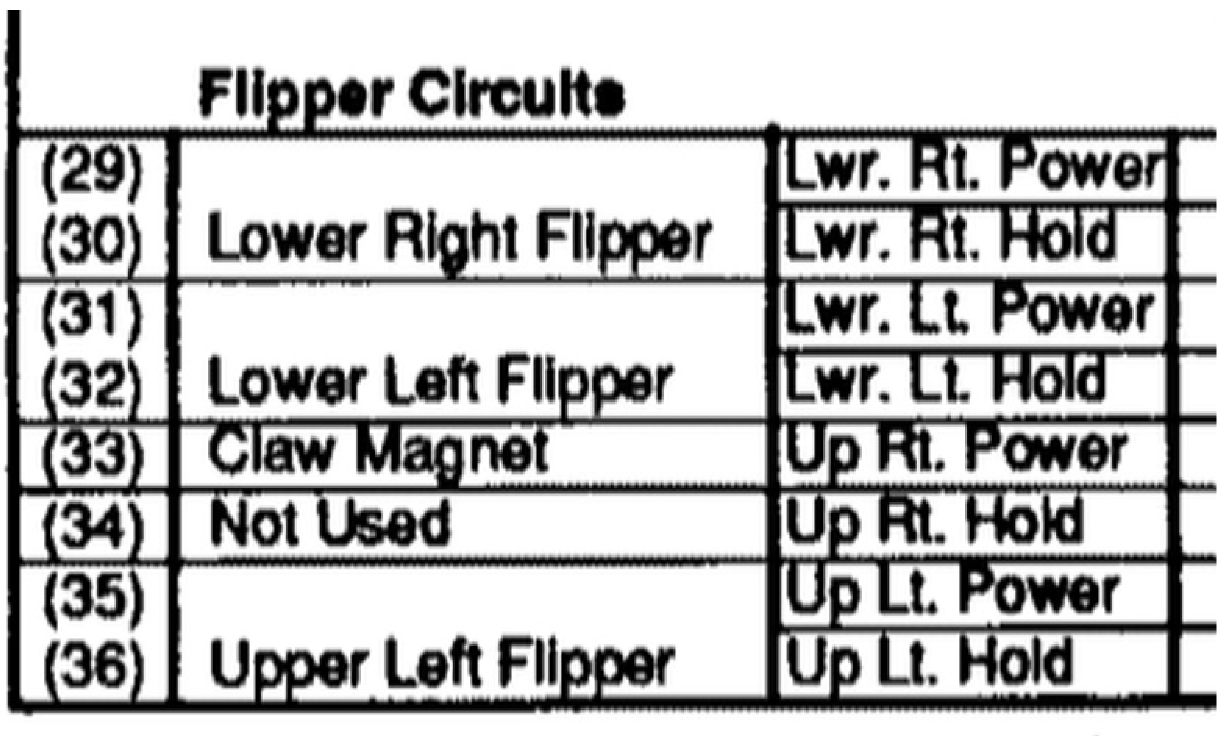
Configuring regular coils

To configure the regular coils (from the "Solenoid / Flasher" table in your machine's operator's manual, enter the letter `C` followed by the solenoid number, like this:

```
coils:
  c_trough_eject:
    number: c01
    default_pulse_ms: 25
  c_bottom_popper:
    number: c02
    default_pulse_ms: 25
  c_plunger_lane:
    number: c03
    default_pulse_ms: 25
```

Fliptronics coils

You'll also see a section in the solenoid table in your ooperator's manual with "Flipper Circuits", like this:



Flipper Circuits

(29)	Lower Right Flipper	Lwr. Rt. Power
(30)		Lwr. Rt. Hold
(31)	Lower Left Flipper	Lwr. Lt. Power
(32)		Lwr. Lt. Hold
(33)	Claw Magnet	Up Rt. Power
(34)	Not Used	Up Rt. Hold
(35)	Upper Left Flipper	Up Lt. Power
(36)		Up Lt. Hold

That section shows the 8 driver outputs that are connected to the Fliptronics board (if your machine has one).

For those coil numbers, you can either enter `C` followed by the number, or the four-letter code indicating which output the driver is connected to, like this:

- c29 or FLRM - Lower Right Main (Power)
- c30 or FLRH - Lower Right Hold
- c31 or FLLM - Lower Left Main (Power)
- c32 or FLLH - Lower Left Hold
- c33 or FURM - Upper Right Main (Power)
- c34 or FURH - Upper Right Hold
- s35 or FULM - Upper Left Main (Power)
- s36 or FULH - Upper Left Hold

Many machines do not use all eight of these, and many machines also connect Fliptronics coils up to other random things (typically magnets and diverters).

An example in your config might be:

```
coils:
  c_flipper_left_main:
    number: fllm
    default_pulse_ms: 30
  c_flipper_left_hold:
    number: fllh
    allow_enable: true
  c_flipper_right_main:
    number: flrm
    default_pulse_ms: 30
  c_flipper_right_hold:
    number: flrh
    allow_enable: true
  c_vanish_magnet:
    number: c35
    allow_enable: true
  c_loop_post_diverter:
    number: c36
    allow_enable: true
```

5. Configuring lights (lamps)

Lights are configured with the letter L followed by the lamp number from the manual:

```
lights:
  l_ball_save:
    number: l11
    subtype: matrix
  l_fortress_multiball:
    number: L12
    subtype: matrix
  l_museum_multiball:
    number: L13
    subtype: matrix
  l_cryoprison_multiball:
    number: l14
```

(continues on next page)

(continued from previous page)

```
    subtype: matrix
  l_wasteland_multiball:
    number: L15
    subtype: matrix
  l_shoot_again:
    number: l16
    subtype: matrix
```

See *lights:* and *light_player:* for details on how to use them.

5. Configuring GI (general illumination)

GI strings are configured with `G` followed by the number, like this:

```
lights:
  gi_back_panel:
    number: g01
    subtype: gi
  gi_upper_right:
    number: g02
    subtype: gi
  gi_upper_left:
    number: g03
    subtype: gi
  gi_lower_right:
    number: g04
    subtype: gi
  gi_lower_left:
    number: g05
    subtype: gi
```

See *lights:* and *light_player:* for details on how to use them.

6. Configuring flashers

Since flashers in WPC machines are technically drivers (coils), they are also configured with the letter `C` followed by their number similar to `coils`.

```
coils:
  f_claw:
    number: c17
  f_jets:
    number: c21
  f_side_ramp:
    number: c22
  f_left_ramp_upper:
    number: c23
  f_left_ramp_lower:
    number: c24
```

See *flashers:* for details on how to use flashers.

Controlling Stern Whitestar Machines

You can connect a *Multimorphic P-ROC* (not the P3-ROC) to your Stern Whitestar machine. Connect your machine according to the P-Roc connector mappings.

More technical information can be found in the PinWiki Stern White Star page.

Controlling Data East Machines

Data east machines can be controlled using the *SNUX System 11 board* connected to any *WPC controller* such as the *Multimorphic P-ROC* (not the P3-ROC). This is similar to *Controlling Williams, Bally System 11 Machines*.

More technical information can be found in the PinWiki Data East/Sega page.

Controlling Williams, Bally System 11 Machines

You can connect your machine using the *SNUX System 11 board* to any *WPC controller* such as the *Multimorphic P-ROC* (not the P3-ROC). This solution requires the existing driver boards.

Another option is to use the *Arduino Pinball Controller (APC)* which also replaces the original driver board. Nothing except the APC board is needed to control the machine using MPF. This can be optionally used together with *LISY* to emulate the original ROM.

More technical information can be found in the PinWiki Williams System 9 and 11 page.

Controlling Gottlieb System 1 Machines

You can connect your machine using a *LISY 1 board*. The documentation regarding switch, coil and light numbering can be found in LISY documentation.

More technical information can be found in the PinWiki Gottlieb System 1 page.

Controlling Gottlieb System 80 Machines

You can connect your machine using a *LISY 80 board*. The documentation regarding switch, coil and light numbering can be found in LISY documentation.

More technical information can be found in the PinWiki Gottlieb System 80 page.

Controlling Williams System 3 to 9 Machines

You can use *APC* to connect your machine. It will replace the CPU, sound and driver board (almost all the PCBs). All you need it a working playfield and the PSU of your machine.

Controlling Stern SAM Machines

You can connect a *Multimorphic P-ROC* (not the P3-ROC) to your Stern SAM machine. Connect your machine according to the P-Roc connector mappings.

More technical information can be found in the PinWiki Stern S.A.M. page.

Controlling Stern SPIKE/SPIKE 2 Machines

MPF can control Stern Spike machines directly using the *SPIKE platform*.

More technical information can be found in the PinWiki Stern Spike page.

Controlling Pinball 2000 Machines

There is a libpinproc-compatible board by Jimmy which can connect to and controll Pinball 2000 machines. You can configure it like the *P-ROC*.

Let us know if you need more informations about this.

More technical information can be found in the PinWiki Williams Pinball 2000 page.

Bally/Stern Machines with AS-2518-17 or AS-2518-35 MPU

To support machines with AS-2518-17 MPU or AS-2517-35 MPU you can use *LISY35*. See Pinwiki Stern/Ball for more hardware details.

Voltages and Power in Pinball Machines

This section is about some general electric details in pinball machines.

Voltages and Power

A pinball machine uses multiple different voltages for different purposes. You need at least one power supply unit (PSU) to transform the AC power to multiple DC rails. See *Wiring and Connectors in Pinball Machines* for more details on the wire thickness and connectors to use for the different power rails below.

Warning: If you are unsure ask a professional electric engineer. This guide does not provide all information needed to design and operate a high-voltage/high-current system in a pinball machine. Use this at your own risk. Electricity can be dangerous and might kill you or burn down your house.

Video about electronics basics:

https://youtu.be/3ZdSLruAYM0

Primary side - 230/110V

At the mains your machine usually runs at 230V or 110V depending on the region. Some PSUs are able to work with both voltages. Sometimes there is a switch to select the input voltage. In other cases a PSU might only work with a certain input voltage. Make sure to check this before connecting the PSU to the mains.

In case you run a traditional transformer you usually have to wire the windings differently depending on the input voltage. If you get this wrong the output voltages might be different or the transformer may burst into flames.

In any case it is a good idea and common practise to add a fuse before your PSU or transformer in case anything goes wrong. This is for your own safety and for the safety of your neighborhood because if stuff starts burning it might cause a lot of damage.

High Voltage - 48V to 80V

High voltage (HV) is used to drive coils in your machine. In modern machines 48V is used which technically classifies as low voltage in most countries around the world (it is still not safe to touch and can kill you). This is preferred if you start a new design as PSUs for 48V are readily available at a good price. Most machines use supplies with around 6A to 10A.

Older machines used transformers with 70V to 80V. Those are more expensive, heavier and harder to get nowadays. They are generally not recommended for new designs. If you want to produce a machine this will also be harder to certify in most countries. Some people use 24V supplies which technically works but is not recommended because coils tend to be quite weak and unreliable in those settings.

A large capacitor might help to keep this rail stable since pulsing and PWMing coil causes large electric and magnetic spikes. In some cases a PSU might turn off while driving coils without a capacitor on this rail. In other cases pulses might be unstable because the voltage will drop too much during the pulse (seems to be common with 24V supplies). If you are increasing pulse times and there seems to be no change in the power of the coil you are likely experiencing the second issue. Adding large capacitors or using a power entry board (see below) is recommended in those cases.

You want to use at least one fuse on the HV rail to prevent coils from burning. Most coils will start burning after a while if you enable them permanently without PWM (see `hold_power` in *coils:* for details). You do not want that. Instead the fuse should trip and cut the power. It might be wise to use multiple fuses (e.g. one per bank of coils).

Common power supplies for 48V:

- Meanwell SP320-48 - Used by Stern Spike (not recommended because it is a bit too weak)
- Meanwell RSP500-48 - Used by Stern Spike 1 (starting from Ghostbusters) and Spike 2
- Meanwell SE-600-48 - Used by Spooky

Common power supplies for 70V - 80V (not recommended for new designs):

- AnTek PS-4N70R5R12 - 70V + unregulated 5V and unregulated 12V

Light Power

Your lights will require a lot of power. Depending on the type of light the voltage might differ. Traditional incandescent bulbs need something around 12V to 24V. LEDs usually run at 5V (sometimes 12V). Make sure to understand how much power you need for your lights. Then calculate which wires, connectors, PSU and fuses you need. This is very likely a high-current setup and standard connectors with thin wires will certainly cause problems (or fire) in your machine.

For instance, every LED will draw around 20mA. Triple that for RGB LEDs. With 80 RGB LEDs for inserts and 80 RGB GIs you will end up at 10A power or 50W. Most connectors are rated for less than 10A and you will see some voltage drop with thin wires (check the resistance).

Make sure this is properly fused since this may easily burn down your machine.

Common power supplies:

- Standard ATX power supplies - Work well but you might have to cut the connectors
- Meanwell SP/MW for 12V or 24V - Precalculate your current and get one with some headroom

Display Power

RGB DMDs usually need either 5V or 12V and might draw a few amps at full brightness. Traditional DMDs might need very high voltages. Definitely ask a professional before getting started with traditional DMDs.

As with any power rail: Add a fuse.

Common power supplies (for 12V):

- Standard ATX power supplies
- Meanwell RD65A - A cheap 5V and 12V supply

Logic Power

In most cases this will be 5V and 12V. Most systems use 12V for switches and 5V to power logic components. In most cases you don't need many amps on those rails. It might be wise to run separate 12/5V rails for logic components and light/display power to prevent problems with interferences.

As with other rails: Add a fuse to be safe.

Common power supplies (for 12V):

- Standard ATX power supplies
- Meanwell RD65A - A cheap 5V and 12V supply

PC Power

Most machines run embedded PCs which come with their own PSU. Sometimes they run off 5V (such as the Raspberry Pi). Others use standard ATX power supplies. See *Controlling your machine & computer power on / power off* for details about power on/off.

Electromagnetic Compatibility EMC/EMI

You need to make sure that your machine complies with regulations and will not disturb police radios/air traffic control or your neighbors Wifi. Especially pulsing or PWMing coils will cause a lot of interferences. This might cause RF emissions and make you a lot of enemies. The most important step to mitigate EMI is to run your power and return wire in parallel and make them the same length. There will be a magnetic field between HV and GND to your coil when current flows. If current changes, the field will change and you will transmit signal which is what you want to avoid. Additionally, add free flow diodes to your coils to prevent self-induction voltage from travelling back to your driver board and PSU (which will transmit another signal).

EMC is a complex topic. If in doubt consult an electic engineer.

Common Ground

> **Warning:** It is very important to connect all grounds if you use multiple PSUs. We cannot stress this enough. Not ensuring this will be very dangerous!

In general, it is preferred to connect the ground at the PSUs than below the playfield. Then run a separate ground for each power rail from the PSU to the playfield.

Interferences on the ground of the HV rail might cause problems in other rails. Especially for serial LEDs and logic power. In case you run into those problems consult with an electric engineer. The right capacitors and the right wiring might help with this case for example.

Common "ground" generally refers to the neutral wire of your PSU which should not be confused with ground/electric earth. See *Ground and Appliance Classes* for details about ground vs neutral.

Power Filter Boards

Some vendors sell power filter boards which help you to build your different power rails. Additionally, those boards allow you to disconnect components at a central location. Usually, those boards will also connect all ground for you.

Some common boards:

- *Multimorphic Power Entry Board*
- *FAST Power Filter Board*
- Spooky/PBL Power Entry Board (part number: #600-0253-00)
- OPP Power Filter Board
- Stern Spike Power Distribution Board (part number: 520-5343-01)
- *CobraPin Pinball Controller* has a built in power filter.

Have a look at the PCB section of hardware.missionpinball.org for DIY designs.

Wiring and Connectors in Pinball Machines

Usually there are two types of wires/connectors used in a pinball machine. One for all low current connections (i.e. switches or logic) and one for high current connections (i.e. coils). See *Voltages and Power* for details about the different voltages and power requirements.

> **Warning:** If you are unsure ask a professional electric engineer. This guide does not provide all information needed to design and operate a high-voltage/high-current system in a pinball machine. Use this at your own risk. Electricity can be dangerous and might kill you or burn down your house.

Video about wiring in pinball:

https://youtu.be/3ZdSLruAYM0

High Current

High currents require proper wires and connectors. Otherwise stuff might get hot and start a fire. This applies to coils and in some cases also to lights (if you power more than one light with a wire). In general, everything above 1A current should use thick wires.

For high current wires you usually want to use AWG 18 or smaller (thicker). The metric equivalent would be 1mm^2 or more. Also consider the resistance per meter/inch of your wire and calculate the voltage drop in advance.

Your connectors should also be spec'd for your expected current. Most 100 mil Molex connectors allow up to 1A which definitely is not enough for coils. For that reason, 156 mil Molex connectors are used for coils. Usually, they are spec'd for 7A (depends on housing and crimp). If you need more than 7A use multiple pins.

Molex part numbers (KK series):

- 2 positions: 09-50-3021
- 3 positions: 09-50-3031
- 4 positions: 09-50-3041
- 5 positions: 09-50-3051
- 6 positions: 09-50-3061
- 8 positions: 09-50-3081
- 9 positions: 09-50-3091
- 10 positions: 09-50-3101
- 11 positions: 09-50-3111
- 12 positions: 09-50-3121
- Crimps: 39-00-0342 or 08-52-0072

Low Current/Logic Power

For logic power you don't need thick wires. Typically, AWG 20-24 or 0.5mm^2 to 0.25mm^2 is used. Connectors are usually 100 mil Molex connectors.

Molex part numbers (KK series):

- 2 positions: 22-01-2027
- 3 positions: 22-01-2037
- 4 positions: 22-01-2047
- 5 positions: 22-01-2057
- 6 positions: 22-01-2067
- 9 positions: 22-01-2097
- 10 positions: 22-01-2107
- 11 positions: 22-01-2117
- 12 positions: 22-01-2127

- Crimps: 08-51-0108 or 08-50-0114

There are also a lot of very cheap no-name replacements for 100 mil KK which work just fine since there should not be any high current on those connectors.

Wire-to-Wire Connections

While most of the wiring in a pinball machine involves Wire-to-Board connections as referenced above, you may also find yourself needing to make wire-to-wire connections with modular connectors. For this purpose, the standard size connector is the .093" Molex connector from the "Standard .093" Pin and Socket Connectors" series. These connectors and crimps are generally rated for 250V/14A.

For AWG 18-22 wires, use the following Molex parts (Note: there are not housings for 5, 7, 8, 10, 11, 13 or 14 circuit options):

- 1 position: 03-09-1011 (receptacle/female) / 03-09-2011 (plug/male)
- 2 positions: 03-09-1022 (receptacle/female) / 03-09-2022 (plug/male)
- 3 positions: 03-09-1032 (receptacle/female) / 03-09-2032 (plug/male)
- 4 positions: 03-09-1042 (receptacle/female) / 03-09-2042 (plug/male)
- 6 positions: 03-09-1064 (receptacle/female) / 03-09-2062 (plug/male)
- 9 positions: 03-09-1094 (receptacle/female) / 03-09-2092 (plug/male)
- 12 positions: 03-09-1126 (receptacle/female) / 03-09-2122 (plug/male)
- 15 positions: 03-09-1157 (receptacle/female) / 03-09-2159 (plug/male)
- Pin (male) crimp: 02-09-1118
- Socket (female) crimp: 02-09-1119
- Pin extractor: Molex 0011030006 or GC Electronics W-HT-2038
- Recommended crimping tool: IWISS SN-28B

Note: While you can purchase large quantities of the crimp pins above on a reel/tape for a slightly cheaper price, you then have to cut off the side wings on each pin while being careful not to cut too much off or the pins will not lock inside the housings. For people new to crimping, this can be a frustrating experience so the "loose/bag" option listed above (02-09-1118 and 02-09-1119) is generally worth the extra .02 or .03 per crimp terminal.

Sourcing Connectors

Those connectors and crimps can be purchased from Digikey or Mouser. Additionally, you can buy those at your pinball supplier but they tend to be quite pricy.

Power Distribution Boards

Your power rails will fan out below the playfield to various mechs and boards. The simplest solution to implement this are terminal blocks which work fine but make it hard to disconnect stuff temporarily. Luckily, various boards exist to solve this issue:

- Spooky/PBL Power Distribution board (part number: #600-0224-00)

- FAST Playfield Interchange Board
- Multimorphic Power Distribution board (part number: PCBA-0031-0003)

Have a look at our PCB section of hardware.missionpinball.org for DIY designs.

Ground and Appliance Classes

Pinball machines commonly are classified as class 1 devices according to IEC 61140 (US) and EN 61140 (Europe). When building or modifying pinball machines you should understand which requirements need to be met for safe operations.

Warning: If you are unsure ask a professional electric engineer. This guide does not provide all information needed to design and operate a high-voltage/high-current system in a pinball machine. Use this at your own risk. Electricity can be dangerous and might kill you or burn down your house.

Video about ground:

https://youtu.be/cFdT7YdXGO0

Class 1 appliances

Class 1 appliances typically connect to a 3-prong AC connector which contains separate ground/electrical earth and neutral. They require that a single fault (e.g. a disconnected conductor wire touching the lock down bar of the machine) may not cause an electric shock. For that reason, all conducting parts need to be connected to the ground. In pinball machines, those are all metal parts such as:

- Legs
- Backbox connector metal parts
- Speaker grills
- Lockdown bar
- Service door
- Screws on the cabinet side

In a lot of cases braid copper wire is used to connect those parts to ground. You should test that a low-impedance connection between any conducting parts and ground exist. See Application classes for details.

Common Ground

If you operate more than one power supplies in your machine make sure to connect all their neutral connectors (N; 0V; commonly referred as ground). Funcionally, this is needed for logic components to maintain a common reference. Additionally, a floating ground might become dangerous when working with voltage multiple voltages. See *Voltages and Power* for details.

Power Management in Software

Related Config File Sections
psus:
coils:

MPF will try to prevent concurrent pulses on the same power supply unit to reduce the maximum current draw. This is important for certain switching power supplies since they might just shutdown on over current. However, MPF will not mess with any timing critical things such as slings, pops or flippers as they are controlled by hardware rules. Instead MPF will delay resets of drop target, ejects of ball devices or advancing of score reels for up to a few milliseconds (configurable). You won't notice this in your machine but it makes eject power much more consistent and drop target resets more reliable. Without this kind of magic most score reels won't work at all because if you pulse 15 coils at once none of them will move.

By default MPF assumes that you have only one single power supply unit for all your coils. If this is not true you can configure multiple PSUs and assign them to coils:

```
psus:
  default:  # this is configured by default
    voltage: 48
  psu_12v:
    voltage: 12

coils:
  c_score_reel_1k_p1:
    psu: psu_12v
    number:
  c_score_reel_100_p1:
    psu: psu_12v
    number:
  c_score_reel_10_p1:
    psu: psu_12v
    number:
  c_score_reel_1_p1:
    psu: psu_12v
    number:
```

This way MPF will sequentialize those coils independently from your coils on the other PSU.

To give your PSU some breathing room MPF will apply some spacing between two pulses. This can be configured using `release_wait_ms`:

```
psus:
  default:
    voltage: 48
    release_wait_ms: 50    # defaults to 10ms
```

Videos about wiring in pinball (series of 6 videos):

https://youtu.be/8ByqYkYKnFc

https://youtu.be/3ZdSLruAYM0

https://youtu.be/cFdT7YdXGO0

https://youtu.be/7G8LFBbYNOY

https://youtu.be/peVNGqb2Wp8

https://youtu.be/C9GzkMduEKY

How MPF handles "quick response" mechs (flippers, slingshots, etc.)

As you can imagine, many types of mechanisms in a pinball machine require near "instant" response to switches. For example, you do not want any "lag" between the time you press the flipper button and the time the flipper physically moves.

To address this, MPF and the control systems handle "quick response" devices in a special way. This includes things like:

- flippers
- pop bumpers
- slingshots
- kicking targets
- kickback lanes
- diverters
- and maybe others?

What's the problem?

To understand why MPF and the hardware control systems work this way, first think about how MPF works in general.

When you configure (and enable) a flipper in MPF, what you're really doing is saying, "when this switch becomes active, fire this coil" (and do that as fast as possible).

The challenge is that MPF is software running on a computer connected to a pinball control system via USB. So if you think about the entire process that needs to happen to flip a flipper, you have:

1. The hardware control system is continuously scanning switches to see if they change state.
2. The player pushes the flipper button.
3. The hardware control system notices the change.
4. The hardware control system adds the message with the switch state change to the queue to be sent to the computer via USB.
5. The computer processes the USB message.
6. MPF gets notification of the switch change.
7. MPF looks at its configuration and notices that a coil should be fired.
8. MPF creates the instruction to fire the coil.
9. That instruction is put in the queue to be sent to the hardware controller via USB.
10. The USB bus transfers that command to the hardware controller.

11. The hardware controller receives that command and fires the coil attached to the flipper.

Of course computers are really fast, and this can all happen in 10 or 20ms. But again, with the desire for "instant" response of these devices, that isn't fast enough.

The solution? "Hardware rules"

So the way this is handled is that all the pinball control systems have the ability to have simple "rules" written to them which lets them do simple things on their own.

These rules are *very* simple and only involve switches and coils. For example, a rule might be "when this switch is activated, pulse that coil", or "when this switch is released, cut off the power to that coil".

Then when one of these "hardware rules" (as we call them in MPF) is written to the hardware pinball controller, that controller can handle it all by itself with minimal delay (usually in a millisecond or two) without having to deal with USB and MPF and all that.

These rules are *not* permanently stored on the hardware controller, and in fact they're constantly added, removed, and updated throughout the course of a game. (Rules for flipper buttons and coils are removed when a ball ends and added when a ball starts, etc.)

By the way, even when MPF writes hardware rules to the pinball controller, the switch notification is still sent to MPF (since you might want to have scoring or play a sound or something when that switch is hit). It's just that in that case, the switch notification is sent to MPF for MPF's game logic purposes, but the actual coil firing would have already happened thanks to the hardware rule on the pinball controller.

This is all automatic

The good news about these hardware rules is that there's nothing you need to do to use them. This is just one of the things that MPF does behind the scenes, thanks to the smart people who designed the pinball controllers.

What kind of rules does MPF use?

1. Pulse + Cancel: This means that we pulse a coil when a switch becomes active and cancel the pulse when the switch becomes inactive.
2. Pulse + Cancel + Hold: This means that we pulse and then enable a coil with pwm when a switch becomes active and cancel the pulse when the switch becomes inactive.
3. Just Pulse: This means that we pulse a coil when a switch becomes active but never cancel the pulse.
4. Pulse + Cancel + Hold + EOS: This means that we pulse and then enable a coil with pwm when a switch becomes active and cancel the pulse when the switch becomes inactive. Additionally, the pulse is changed to pwm when EOS becomes inactive (it's usually normally closed).

For most platforms 1 and 2 is basically the same rule (e.g. rule 1 is rule 2 with `hold power = 0`).

We use type 2 for single wound flippers. For dual wound we use type 1 on the main/high power coils and type 2 on the hold coil (often with 100% pwm = full enable). When flippers have EOS we use type 4 rules (for dual wound flippers with hold=0). Rule 3 is used for pop bumpers.

Not all platforms support all types of rules. In those cases we use the next best available rule (e.g. 1 instead of 3 or the other way around).

How to configure "number:" settings

All of the physical "hardware" mechanisms in MPF config files have a `number:` setting which is used by the hardware platform to know which device is which.

Since MPF supports many different types of hardware, the exact way you configure the "number" entry depends on what type of device and what type of hardware you're using.

We have full guides that explain it all in the hardware controller documentation, but here are the links all in one place to make it easy.

Switches

- *FAST Pinball*
- *P-ROC*
- *P3-ROC*
- *Open Pinball Project (OPP)*
- *Stern SPIKE*
- *Snux (System 11)*

Drivers / Coils

- *FAST Pinball*
- *P-ROC / P3-ROC*
- *Open Pinball Project (OPP)*
- *Stern SPIKE*
- *Snux (System 11)*

LEDs

- *FAST Pinball*
- *P-ROC / P3-ROC (PD-LED)*
- *Open Pinball Project (OPP)*
- *Stern SPIKE*
- *FadeCandy*

Lamp Matrix-based lights

- *FAST Pinball (WPC Machine)*
- *P-ROC / P3-ROC (PD-8x8)*
- *P-ROC (WPC Machine)*
- *P-ROC (Stern SAM / Whitestar)*
- *Open Pinball Project (OPP)*
- *Snux (System 11)*

Servos

- *FAST Pinball (Servo daughterboard)*
- *I2C servo controllers*
- *Pololu Maestro*

Mixing-and-Matching hardware platforms

In MPF it's possible to mix-and-match your hardware platforms. For example, you could use a P-ROC for your coils and switches while using a FadeCandy for your LEDs. (Or, if you wanted to be crazy, you could use a FAST controller for your switches and a P-ROC for your coils and lamps.)

You can specify hardware platforms in three ways:

1. Machine-wide default platform

Whatever you set in the `hardware: platform:` section of your machine-wide config is the default platform for all types of mechanisms across all of MPF.

For example:

```
hardware:
  platform: p_roc
  driverboards: pdb
```

In the above config, the P-ROC platform will be the default for everything. (switches, coils, lights, LEDs, DMDs, servos, etc.)

2. Device-specific default platform

If you want to specify a default for a certain class of devices that is different than the machine-wide default, you can also do that in the `hardware:` section by adding an entry for the type of device you want to specify the default for.

For example, if you want to use a P-ROC as the default for everything except for LEDs, which you want to be FadeCandy, you would do it like this:

```
hardware:
  platform: p_roc
  driverboards: pdb
  lights: fadecandy
```

You can enter a device-specific default for the following types of devices here:

- coils:
- switches:
- matrix_lights:
- lights:
- dmd:
- rgb_dmd:
- gis:
- flashers:
- servo_controllers:
- accelerometers:
- i2c:

3. Overriding the platform of individual devices

Finally, you can override the platform of an individual device by adding a `platform:` setting to that device.

For example, if you're using a FAST Pinball controller which can control up to 256 LEDs, but you also want to add some more LEDs that will be attached to a FadeCandy, you could set up your config like this:

```
hardware:
  platform: fast
lights:
  led00:
    number: 0-0
  led01:
    number: 0
    platform: fadecandy
```

In this example, *led00* will use the FAST platform (and the number 0-0 is a FAST configuration number), and *led01* will use the FadeCandy platform (and the number 0 is a Fadecandy number).

You could also invert this, like so:

```
hardware:
  platform: fast
  lights: fadecandy
lights:
  led00:
    number: 0-0
```

(continues on next page)

(continued from previous page)

```
    platform: fast
  led01:
    number: 0
```

In the example above, *led00* is still a FAST LED and *led01* is still a FadeCandy LED, but the difference is that while the default platform is FAST, the default platform for LEDs is FadeCandy. That means you don't have to specify the platform for LEDs attached to the FadeCandy, but you do need to specify the platform for LEDs attached to the FAST controller.

Hardware Roadmap

There are a few hardware platforms we would like to add in the future because we think they would be a good fit for custom/homebrew pinball machines:

- Pololu Jrk - An USB motor controller
- Ion Motion RoboClaw - A USB motor controller
- Stern Spike 2 - The bus is similar to Spike 1. Needs testing.
- MyPinballs Custom Pinball Board - Control System for older machines
- I2C Segment Displays (such as this display from adafruit)

Let us know in the MPF User Forum if you want to use any of those hardware platforms. Please also let us know if you know other hardware which we should support.

Troubleshooting Hardware Platforms

If you got problems with your hardware platform we first recommend to read our *troubleshooting guide*. Here are some hardware platform specific steps. This is a generic guide so please check if there is a more specific guide for your specific platform.

Enable Debugging

If you got problems with your platform try to enable debug first. As described in the *general debugging section* of our *troubleshooting guide* this is done by adding `debug: true` to your platform config section. This will add a lot more debugging and might slow down MPF a bit. We recommend to disable/remove it after finishing debugging.

Reducing light update rate

If you got a lot of lights you might run into bus contention issues. You can reduce the light update rate in MPF:

```
mpf:
  default_light_hw_update_hz: 30   # defaults to 50
```

If you set this too low fades will be less smooth but otherwise it should not affect your game.

Coils Are Not Firing

What to do if your coils are not working?

Check if Your Hardware is Working at all

Sounds stupid but this is a good start: Is the hardware working at all? Do you see switch hits in the logs? If not, check our section `Your hardware is not working at all`.

Check the Watchdog

If switches (or other features of the platform) are working but coils are not we have to dig deeper. Most hardware platforms have some kind of watchdog. Often there is some LED which indicates if the watchdog is received. The MPF log might also contain clues (especially if you have enabled `debug` and run MPF with verbose flags `-v -V`). If the watchdog is not received by your platform it will not enable coils.

In most cases watchdog related problems indicate wiring problems. Check if your boards are properly wired.

Test Your Coil Numbers using MPF Service CLI

Hardware is connected and generally working, watchdog is good but still your coils are not working? Maybe something with the numbering is odd. Lets tests that using the *MPF Service CLI*. Alternatively, you can also use *service mode* if you have already configured it. Both ways work similarly.

To use service cli:

1. Open two consoles
2. Start your game (e.g. using `mpf both`)
3. Start the service cli from within your game folder using `mpf service`.
4. Type `list_coils` and press `ENTER` to see a list of coils.
5. Type `coil_pulse your_coil` and press `ENTER` to pulse it.

Does it work? If not check the log and try verify the coil number. If you do not specify `default_pulse_ms` MPF will use `10ms` which might not be enough for some mechs. Try to increase that gently (maybe `20ms` or `30ms`).

Your hardware is not working at all

If your hardware is not working at all make sure that you removed the options `-X`, `-x` and `--vpx` from your `mpf both` or `mpf game` command line. Those options will overwrite the settings in your `hardware` section and MPF will not even try to connect to your hardware. If you got config errors we suggest you add `-X` to figure things out without interfacing real hardware all the time. Just keep that option in mind.

Another stupid thing to check: Is your hardware connected to your PC? We know it is stupid but a loose USB connector has happened to most of us.

On Linux you might want to run the command `lsusb` which should show both of your micro controllers connected. You should see two lines similar to

```
Bus 002 Device 014: ID 0483:5740 STMicroelectronics Virtual COM Port
Bus 002 Device 015: ID 0483:5740 STMicroelectronics Virtual COM Port
```

If you are unsure about the output, run the command once with your controllers connected and once without. If there is no difference, then for sure the USB device is not properly connected.

Add debugging to related devices

If you got problems with some switches also add `debug: true` to those as it will give to more insights into the intentions of those devices. Same will work for flippers, coils, lights, servos, steppers and more. See *general debugging section* for details.

Run MPF with verbose flag

See *general debugging section* for details. TLDR: run `mpf both -t -v -V`.

Report Your Issue and Ask For Help

If you cannot find the issue yourself please prepare some information about your issue according to our *troubleshooting guide* and ask in our forum.

Consider Improving the Documentation

Did you solve your issue but found that some relevant information in the documentation is missing or should be linked/located elsewhere? Either tell us in the forum or consider *improving the documentation* yourself to save future users some troubles the same way others saved you some troubles by writing this documentation.

Browse Platforms by Capabilities

I2C Platforms in MPF

The following platforms allow controlling I2C devices in MPF:

- *Linux Nativ I2C* - If your linux PC has a driver for the I2C interface it will work in MPF
- *P3-Roc* (but not the P-Roc)
- *Raspberry Pi* - Remote via network or locally using pigpio

The following platforms need to be interfaced by one of the above platforms: * *PCA9685/PCA9635 I2C Servo Controllers* * *MMA8451 Accelerometers*

Servo Platforms in MPF

The following platforms allow controlling *servos* in MPF:

- *FAST servo daughter boards*
- *PCA9685/PCA9635-based servo controllers via I2C*
- *Raspberry Pi* - Remote via network or locally using pigpio
- *Pololu Maestro*
- *P3-Roc and P-Roc via PD-LED*
- *Penny K Pinball PKONE Extension boards*

Overview video about *servos*:

https://youtu.be/wA6KEODwQ5w

Stepper Platforms in MPF

The following platforms allow controlling steppers in MPF:

- *P3-Roc and P-Roc via PD-LED*
- *Trinamics Steprocker*
- *StepStick Steppers*
- *Pololu Tic*

Overview video about *steppers*:

https://youtu.be/YaRNBU0OHGc

Segment Display Platforms in MPF

The following platforms support segment displays in MPF:

- *LISY DMDs*
- *MyPinballs Displays*
- *Multimorphic P-Roc*
- *Segment Displays connected as Lights*
- *NeoSeg Serial Segment Displays*
- *Virtual Segment Display Emulator*

Video about segment displays:

https://youtu.be/Jyf3jxGXnTw

DMD Platforms in MPF

The following platforms support DMDs in MPF:

RGB DMDs

- *SmartMatrix*
- *RGB.DMD*
- *FAST Pinball RGB DMD*
- *Raspberry Pi DMD*
- *PIN2DMD RGB DMD*

Monochrome DMDs

- *Multimorphic P-Roc*
- *Stern Spike 1*

Segment Display Transitions

When MPF switches the current text on a segment display with another text entry, a transition effect can be set that controls what text transition between the new and existing text looks like. You can use these transitions with the *Segment Display player* and within shows. You can set transitions as a property of the new text entry that comes in, or as a property of the outgoing transition when the current text entry is removed (incoming transitions always take precedence over outgoing transitions).

Here's a list of all the types of segment display text transitions that MPF supports.

none

Setting a transition type of `none` means that no transition will be used, and the incoming text instantly replaces the current text.

push

The push transition means that the incoming text "pushes" the outgoing text out of the way. (e.g. the outgoing text moves out while the incoming text moves in)

Options for the push transition:

- direction: `left` or `right` (defaults to `right`).
- text: An optional text string that is inserted between the old and new text during the transition. Defaults to empty.
- text_colors: The color for each character in the optional transition `text` string (if the platform supports it). If a single color is supplied, all characters in the transition `text` string will be set to that color. See *Specifying Colors in Config Files* for more information on specifying colors in config files.

cover

The cover transition means that the incoming text moves in on top of to cover the current text. The outgoing text is not animated.

Options for the cover transition:

- direction: `left` or `right` (defaults to `right`).
- text: An optional text string that is inserted between the old and new text during the transition. Defaults to empty.
- text_colors: The color for each character in the optional transition `text` string (if the platform supports it). If a single color is supplied, all characters in the transition `text` string will be set to that color. See *Specifying Colors in Config Files* for more information on specifying colors in config files.

uncover

The uncover transition means that the current text is moved out to uncover the new incoming text.

Options for the uncover transition:

- direction: `left` or `right` (defaults to `right`).
- text: An optional text string that is inserted between the old and new text during the transition. Defaults to empty.
- text_colors: The color for each character in the optional transition `text` string (if the platform supports it). If a single color is supplied, all characters in the transition `text` string will be set to that color. See *Specifying Colors in Config Files* for more information on specifying colors in config files.

wipe

The wipe transition means that the display text is wiped/switched from the current text to the incoming text.

Options for the wipe transition:

- direction: `left` or `right` (defaults to `right`).
- text: An optional text string that is inserted between the old and new text during the transition. Defaults to empty.
- text_colors: The color for each character in the optional transition `text` string (if the platform supports it). If a single color is supplied, all characters in the transition `text` string will be set to that color. See *Specifying Colors in Config Files* for more information on specifying colors in config files.

split

The split transition means that the text is split and either moved in or out to reveal the other text value.

Options for the split transition:

- mode: push or wipe (defaults to push).
- direction: in or out (defaults to out).

Configuring Transitions

Transitions are specified as an additional property of a segment_display_player: config or the segment_displays: section of a show config. For example:

```
segment_display_player:
  jackpot_completed:
    display1:
      text: JACKPOT
      priority: 1000
      expire: 2s
      transition:
        type: push
        direction: right
        text: " *** "
      transition_out:
        type: push
        direction: right
        text: " *** "
```

When the event "jackpot_completed" occurs, MPF will update the text in the segment display called "display1" using the push transition. After 2 seconds, the "JACKPOT" text will expire and be removed, pushing the text out to the right, restoring the previous text.

Note: If the current text has a transition_out: setting, and the new text has a transition: setting, then the new text's transition setting will take precedence.

CHAPTER 7

Pinball Mechanisms

MPF supports all the various pinball hardware mechanisms you'd expect. Some of these are basic (switches, LEDs, coils), and others are built up by combining multiple simpler mechs (Switch X plus Coil Y = Flipper 1, etc.)

Pinball mechs are mostly configured in machine-wide config files. Each one has a name, and there are configuration options for each which control exactly how it behaves (or how its behavior changes depending on what's going on in your game).

Pinball Mechs in MPF include (but are not limited to):

Accelerometers

Related Config File Sections
accelerometers:

- *Monitorable Properties*
- *Related How To guides*
- *Related Events*

An accelerometer is a device that measures proper acceleration; proper acceleration is not the same as coordinate acceleration (rate of change of velocity). For example, an accelerometer at rest on the surface of the Earth will measure an acceleration due to Earth's gravity, straight upwards (by definition) of g ~= 9.81 m/s2. By contrast, accelerometers in free fall (falling toward the center of the Earth at a rate of about 9.81 m/s2) will measure zero.

TODO: Add a picture of an accelerometer

Accelerometers in pinball could be used to measure a machine's TILT, replacing the tilt bob, to measure vibration, or even the angle of the playfield at a given time.

Learn more at: https://en.wikipedia.org/wiki/Accelerometer

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for accelerometers is `device.accelerometers.<name>`.

value A three-item tuple (x, y, z) of the current accelerometer values.

Related How To guides

Help us to write it

Related Events

None Varies based on the configured (you can configure events to be emitted when certain G-force thresholds are exceeded).

Autofire Coils

Related Config File Sections
autofire_coils:

An autofire coil in MPF is used for "instant response" type devices (like pop bumpers and slingshots) where you want a switch activation to trigger a coil as close to instantaneous as possible.

First, some background…

The Mission Pinball Framework is based on Python. Running a "real" pinball machine means you have some kind of computer-like board running Python (Mini ITX x86 computer, Raspberry Pi 3, etc.) which runs your game, controls the display, and plays your sounds. That computer connects to your hardware controller (P-ROC, FAST, etc.) to interface with your actual pinball machine components (switches, coils, lights, motors, LEDs. . .).

There are several types of devices in a pinball machine that you want to react "instantly." For example, when a switch in a slingshot or pop bumper is activated, you want the coil to fire as fast as possible. When the player pushes a flipper button, you want that flipper to fire instantly, and when the player releases the flipper button, you want the machine to cut power to that flipper coil instantly. Unfortunately if you think about what the flow chart of activity looks like for that to happen, there are a lot of steps. (And it's certainly not instant.) For example, imagine what happens when a ball hits a slingshot:

1. The slingshot switch is activated.
2. The hardware controller debounces that switch.
3. The hardware controller sends a notification that the slingshot switch changed state to your Python game code via USB.
4. Something in your code says, "if the slingshot switch is activated, fire the slingshot coil."
5. The Python game code sends the "fire the slingshot coil" command to the hardware controller via USB.
6. That command is queued on the USB bus and transmitted.
7. The hardware controller fires the slingshot coil.

Wow! That's a lot of steps just to fire a coil when a switch is hit! Unfortunately the entire process of all this going from the hardware to the computer to the game code to the hardware to the coil takes some time—-maybe 10ms or so. But with a fast moving pinball you might find that it's not fast enough. (What if your game code was in the middle of updating a bunch of lights and that delayed it another 5ms?) You might find that by the time your game code gets around to firing the coil it's too late. In effect your slingshot firing has lag and might miss the ball altogether. Not good!

Fortunately the people who designed the hardware controllers know this, so they have options where "autofire" or "trigger" rules can be written into the hardware controller which the hardware controller can handle on its own. In the Mission Pinball Framework, we call these types of rules "Autofire" rules, because we specify that a coil fires automatically based on some switch event without any involvement of our host computer or the Python game code.

To use an autofire rule, you specify the name of a switch, the state of the switch (whether it goes active or inactive), the name of a coil or driver, and what you want that coil to do. (Turn on, turn off, pulse for a certain number of milliseconds, receive a pwm pulse pattern, etc.)

So for example, if you want to configure a slingshot, you might use a rule on your hardware controller which says, "when switch *left_slingshot* goes active, fire coil *left_slingshot_coil* for 30ms." Or you might have a rule which says, "When switch *right_flipper* becomes inactive, cut power to the coil called *right_flipper_hold*.

You can set any combination of rules you want onto a hardware controller. In fact, MPF will use several individual rules on the same set of switches and coils to do what might seem like simple things. For example, think about what rules you'd need for a dual-wound (power and hold windings) flipper coil:

- When the flipper button becomes active, enable the power coil.
- When the flipper button becomes active, enable the hold coil.
- When the EOS switch becomes active, disable the power coil.

- When the flipper button becomes inactive, disable the hold coil.
- When the flipper button becomes inactive, disable the power coil. (We need this one to "cancel" the flip action if the player releases the flipper button before the flipper hits the EOS switch at the top of its stroke.)
- If the flipper button is active *and* the EOS switch becomes inactive, enable the power coil. (This causes the flipper to go back to the "up" position if for some reason it comes down when the player is holding the flipper button.)

Now look at that above list. That's six rules just for one flipper! If you have four flippers in your game, you'll have 24 autofire rules just to get your flippers set up!

Fortunately MPF makes this easy and hides the complexity from you. :)

How MPF interacts with autofire rules

The hardware controllers in your pinball machine have no concept of what your game code is doing at any given time. (Actually they don't even know what a "game" is, or really what a "pinball machine" is.) They just know that they have rules programmed into them, and those rules specify what instantaneous actions they should take based on certain switches changing state. So your game code can overwrite rules at any time (and as often as you want) to overwrite existing rules with new actions. For example, if your player tilts the machine, then you need to disable the flippers. To do so you would overwrite the above six rules with the following:

- When the flipper button becomes active, do nothing.
- When the flipper button becomes inactive, do nothing.
- When the EOS switch becomes active, do nothing.
- When the EOS switch becomes inactive, do nothing.

And just like that, your flippers are disabled! You can also see how you can use these autofire rules to do all sorts of fun things, like reversing the flippers (so the right button controls the left flipper and vice versa), or making "no hold" flippers, or inverting the flipper buttons so pushing them in disables the flippers and letting go enables them. :)

The final thing that's important to know about these autofire rules you program into your hardware controller is that they do not prevent the hardware controller from doing everything else it might do. For example, if you have a pop bumper then you will probably install an autofire onto your hardware controller that causes the pop bumper coil to fire instantly to knock the ball away.

When that rule is installed, the hardware controller will do two things when the pop bumper switch is activated. First, it will fire the coil, but second, it will also notify MPF that the pop bumper switch was hit (since it notifies your game of any switch that was hit). Then your game code can respond how you want, perhaps by scoring some points and playing a sound effect. When this happens, *technically speaking* they won't happen at the same time. The hardware controller will probably fire the coil in under 1ms, and it might take your game code 5 or 10ms to add the score and play the sound. But that's fine. 10ms is still 1/100th of a second and no human player is going to notice that delay. (Heck, the speed of sound is so slow it takes another 1/100th of a sound for the sound wave to travel from your machine's speaker in the back box to the player's ear!)

The point is that just because you install autofire rules doesn't mean you can't also service those switches in your game code. It's just that you end up dividing the duties-—the hardware controller handles the coil responses on its own, and you handle audio and scoring in your game code.

Oh, by the way, it's not like you need to use these autofire rules for *all* your coil activity. Most things like ejecting balls, resetting drop targets, and firing your plunger can all be handled in your game code because in those cases you don't care about the extra 1/100th of a second delay. You only need autofire rules for things you want to happen instantly, which is usually only pop bumpers, slingshots, and flippers.

How MPF handles autofire rules

Now that you just read 1500 words on how autofire rules work, the good news is that you don't really have to worry about these details of them when using the Mission Pinball Framework. In MPF, you use the configuration files to setup devices like pop bumpers, slingshots, and flippers, and the framework handles all the autofire hardware rule programming based on the switches and coils you specify in your config files.

In fact the framework automatically creates lists of your devices and gives them enable() and disable() methods, so rather than having to know all the intricacies of all those different rules, enabling your flippers is as simple as self.flippers.enable(). Nice! (But if you dig through the source code you'll see that the framework uses all these rules behind the scenes.)

You can also configure autofire coils manually for simpler things like pop bumpers and slingshots. See the *autofire_coils* section of the configuration file reference for details.

Debounce and Recycle in Autofire Coils

In MPF you can *configure debounce for each switch* and *recycle for each coil*. If you do that MPF will respect that configuration for autofire hardware rules. However, if you do not configure it (or set debounce to `auto`) MPF will try to select a reasonable default. For autofire coils it selects debounce `quick` if you either did not specify debounce or set it to `auto`. Recycle will be set to `true` if you do not specify it.

In some platforms MPF might reconfigure your switch debounce settings when activating the hardware rules (if the platform does not allow separate settings). This happens when debounce is set to `auto` (or unspecified) as switches are then automatically configured as debounce `normal` and then reconfigured as `quick` when the rule is send to the hardware (if the platform only supports one configuration at a time).

You can overwrite both settings using `switch_overwrite` and/or `coil_overwrite` in your `autofire_coils` section.

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for autofire coils is `device.autofires.<name>`.

enabled Boolean (true/false) which shows whether this autofire coil is enabled.

Fully working basic example

Let's learn by example. Though the following example is a fully working minimal set for the Cobra controller, it is as well helpful to understand the concpet more if you use a different set of hardware. For this example to work physically, the Cobra board needs to have the micro controllers powered up only. No need for a high voltage power supply, neither for any coil. The `config.yaml` below is the only configuration file you need in your project. The config file is fully valid for the Cobra board connected to a Linux PC running MPF. If you have a Cobra board but run Windows or macOS you have to change the `ports`. If you run a completely different hardware you have to adapt the `hardware` section.

```
#config_version=5

hardware:
   platform: opp
   driverboards: gen2

opp:
   ports: /dev/ttyACM0, /dev/ttyACM1

playfields:
playfield:    #playfield must exist for autofire coils
   tags: default
   default_source_device: bd_plunger   #value must be set, default "none" not allowed when having autofire coils

ball_devices:
bd_plunger:
   ball_capacity: 1
   mechanical_eject: true

coils:
c_my_coil:
   number: 0-0-11

switches:
s_my_switch:
   number: 0-0-16

autofire_coils:
my_autofire_1:
   coil: c_my_coil
   switch: s_my_switch
   enable_events: simulate_start
   disable_events: simulate_stop

keyboard:
1:
   event: simulate_start
2:
   event: simulate_stop
```

Now run `mpf both` to start above example. The Cobra board has a little LED next the coil output which will light up yellow when the coil is activated, see the *Cobra board* documentation for details. Now press the connected switch, you will see that the LED will not light up since the coil has not been activated. Press key 1 and afterwards press again the switch, this time you will see the LED light up for a short time. After you pressed the key 2, the LED won't light up anymore when the switch is activated, because you deactivated the coil.

A few comments on the above example:

- The playfield is needed even in this basic example, in a real setup you have it anyways.
- Coils are enabled by MPF upon the `ball_started` event and disabled by the events

ball_will_end, service_mode_entered. In our basic example we don't have these events, thus added our own events when the keys are pressed. In a real pinball most likely you won't have these additional events.

- Both, coil and switch, need to be controlled by the same micro controller for autofire coils, as you can see both `number` value starts with 0. If you would use different values MPF will throw an exception once the coil is being enabled, but not directly at startup. The error message is `Config File Error in OPP: Invalid switch being configured for driver. Driver = 1-0-1 Switch = 0-0-16. Driver and switch have to be on the same board.`
- The auto fire rules are stored in the micro controller. If you execute the above example, then change the coil to another coil (on micro controller 0) and run it again. Now the switch will then trigger both coils. If you do these kind of changes you want to power down the micro controllers to have a fresh start and avoid strange behavior.

Related How To guides

- *Tutorial step 13: Add slingshots, pop bumpers, and other "autofire" devices*

Related Events

None The autofire coils can be configured to enable or disable based on other events.

Ball Devices

Related Config File Sections
ball_devices:

A *ball device* is any physical thing in a pinball machine which is able to hold (i.e. "capture") a ball and then release it. (Either automatically or based on some action by the player.) Examples of ball devices include the trough, the plunger lane, VUKs, poppers, playfield locks, etc.—basically anything that can hold a ball. (Even the playfield is technically a ball device since balls rolling around are "in" the playfield device.)

Ball devices are usually made up of switches (which are typically used to count how many balls the ball device has) and coils (which are typically used to eject a ball from a device.) Most games have several ball devices. At a minimum they'll have the device that holds the ball when it drains and the playfield.

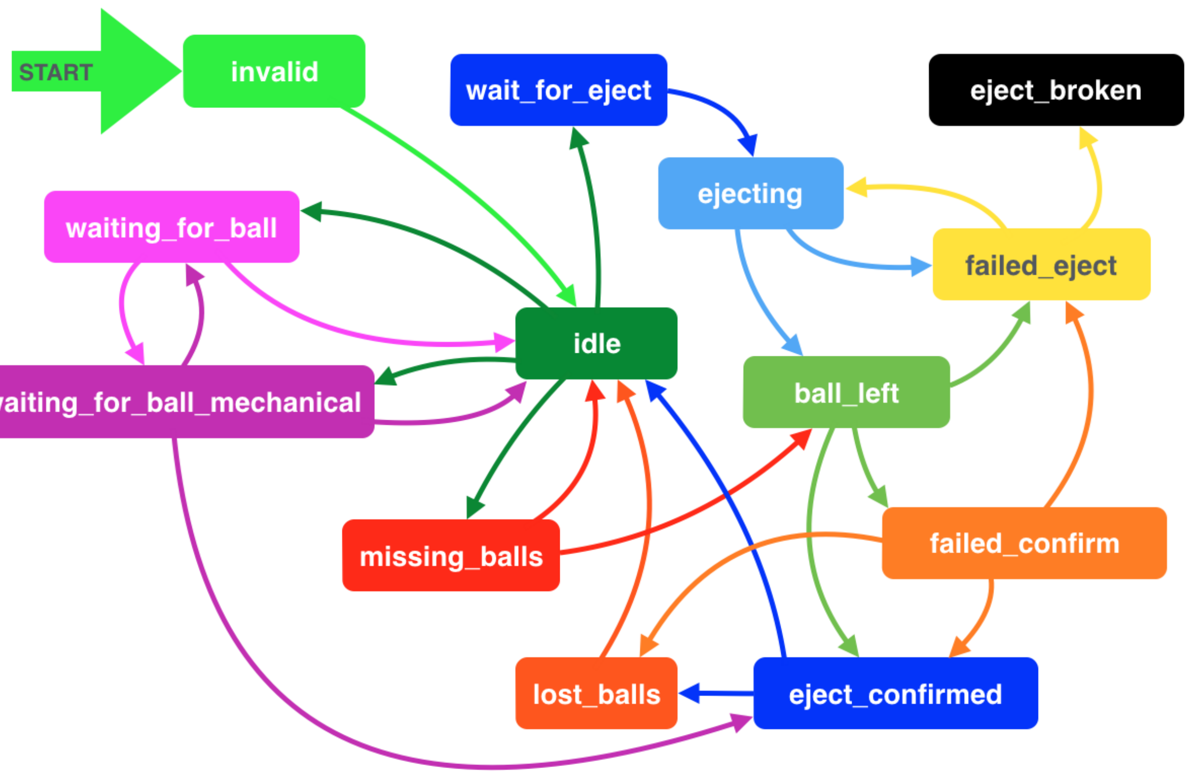
Ball devices are probably the most important element of MPF (because no one likes it when a machine gets confused about where the balls are) and something we've spent a lot of time on. They work hand-in-hand with MPF's Ball Controller to keep track of where all the balls are at any given time.

In MPF, ball devices are implemented as finite state machines.

Each ball device is responsible for managing its own state, which can be:

- idle
- missing_balls
- waiting_for_ball
- waiting_for_ball_mechanical
- ball_left
- wait_for_eject
- ejecting
- failed_eject
- eject_confirmed

Here's a diagram which shows the relationships between the various states. A device can only transition from its current state to one of the states an arrow is connected to.



When you configure ball devices in MPF, you configure the list of other devices that a ball device can eject to. This allows MPF to have an understanding of the "chain" of devices and enables it to route balls to where they need to go. (*Diverters* also figure into this chain, meaning MPF can ensure that diverters are set properly as it's routing balls around.)

Here's a simplified example of how the "chain" of ball devices works:

A simple modern machine would have a minimum of three ball devices:

- The trough
- The plunger lane
- The playfield (remember in MPF, the playfield is technically a ball device)

When you configure your ball devices, the trough is configured so that the plunger lane is its eject target, the plunger lane is configured with the playfield as its eject target, and the playfield is configured to know that it drains into the trough. So you have a complete loop of devices.

This means that, for example, if the playfield wants another ball (like for a multiball), MPF knows that the playfield gets balls from the plunger lane, and if the plunger lane doesn't have a ball, MPF knows that the plunger lane can get a ball from the trough.

Pretty cool!

Of course in a real machine, you'll have a lot more than the three ball devices listed above.

Picking a random machine as an example, *Judge Dredd* has eight(!) ball devices:

1. *The trough*
2. The right *plunger lane*
3. The left plunger lane
4. The Sniper VUK
5. The Hall of Justice VUK
6. The Deadworld orbit thingy
7. The crane
8. *The playfield*

MPF keeps track of how many balls are in each ball device at all times, and it knows which devices are in the process of ejecting (and which target devices they're ejecting to), so it also knows if balls get stuck along the way.

Ball devices support all sorts of settings and events. You can also configure counting delays to account for balls bouncing around before they settle, you can specify how devices confirm that balls have successfully ejected, as well as dozens of other options that allow MPF to support every known type of device in every pinball machine ever created. (Seriously.)

Video on ball tracking in MPF:

https://youtu.be/Yh8ittsuWIc

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for ball devices is `device.ball_devices.<name>`.

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

state What state this device is in.

balls How many balls this device is currently holding.

Related How To guides

- *How to configure a modern trough with opto switches*
- *How to configure a modern trough with mechanical switches*

- *How to configure an older style trough with two coils and switches for each ball*
- *How to configure an older style trough with two coils and only one ball switch*
- *How to configure a classic single-ball trough*

Troubleshooting P-Roc/P3-Roc

If you got problems in general we first recommend to read our *troubleshooting guide*. Here we go into details for ball devices (troughs, plungers, lock devices and more) in particular.

Add debug

First, add a `debug: true` entry into your trough config in the `ball_devices:` section. Then when you run with verbose logging (`-v`), you'll get extra debugging information in the log.

Received unexpected ball

You might get a line in your log telling your that the device received an unexpected ball. This is usually not an issue. It means that the device did not expect the ball. For instance, if a ball drains into a trough or jumps into the shooter that is unexpected for the device. In most cases you will see this message when a ball drained.

So what are expected balls? In modern machines the trough ejects a ball into the shooter lane in which case the shooter lane device expects the ball from the trough. This is connected to ball tracking and retry behaviour of devices.

Ball Count Does Not Match

If your log file shows a number of balls contained in your trough that doesn't match how many balls you actually have, that could be:

- You didn't add all the ball switches to the *ball_switches:* section of the trough configuration
- You're using a physical machine but a switch isn't adjusted properly so the ball is not actually activating it. (Seriously, we can't tell you how many times that's happened! We've also found that on some machines, if you only have one ball in the trough that the single ball isn't heavy enough to roll over the top of the eject coil shaft. In that case we just add a few more balls to the machine and it seems to take care of it.) Either way, if you have a ball in the trough, the switch entry in your log should show that the switch is active (*State:1*), like this:

```
2014-10-27 20:05:29,891 : SwitchController : <<<<< switch: trough1, State:1 >>>>>
```

If you see State:1 immediately followed by another entry with State:0, that means the ball isn't activating the switch even though it might be in the trough.

Add debugging to related devices

If you got problems with some switches also add `debug: true` to those as it will give to more insights into the intentions of those devices. Same will work for flippers, coils, lights, servos, steppers and more. See *general debugging section* for details.

Run MPF with verbose flag

See *general debugging section* for details. TLDR: run `mpf both -t -v -V`.

Report Your Issue and Ask For Help

If you cannot find the issue yourself please prepare some information about your issue according to our *troubleshooting guide* and ask in our forum.

Consider Improving the Documentation

Did you solve your issue but found that some relevant information in the documentation is missing or should be linked/located elsewhere? Either tell us in the forum or consider *improving the documentation* yourself to save future users some troubles the same way others saved you some troubles by writing this documentation.

Related Events

- *balldevice_(name)_ball_count_changed*
- *balldevice_(name)_ball_eject_attempt*
- *balldevice_(name)_ball_eject_failed*
- *balldevice_(name)_ejecting_ball*
- *balldevice_(name)_ball_eject_success*
- *balldevice_(name)_broken*
- *balldevice_captured_from_(captures_from)*
- *balldevice_(name)_ball_enter*
- *balldevice_(name)_ball_entered*
- *balldevice_(name)_ball_missing*
- *balldevice_ball_missing*
- *balldevice_balls_available*

Coils (Solenoids)

Related Config File Sections
coils:
coil_player:

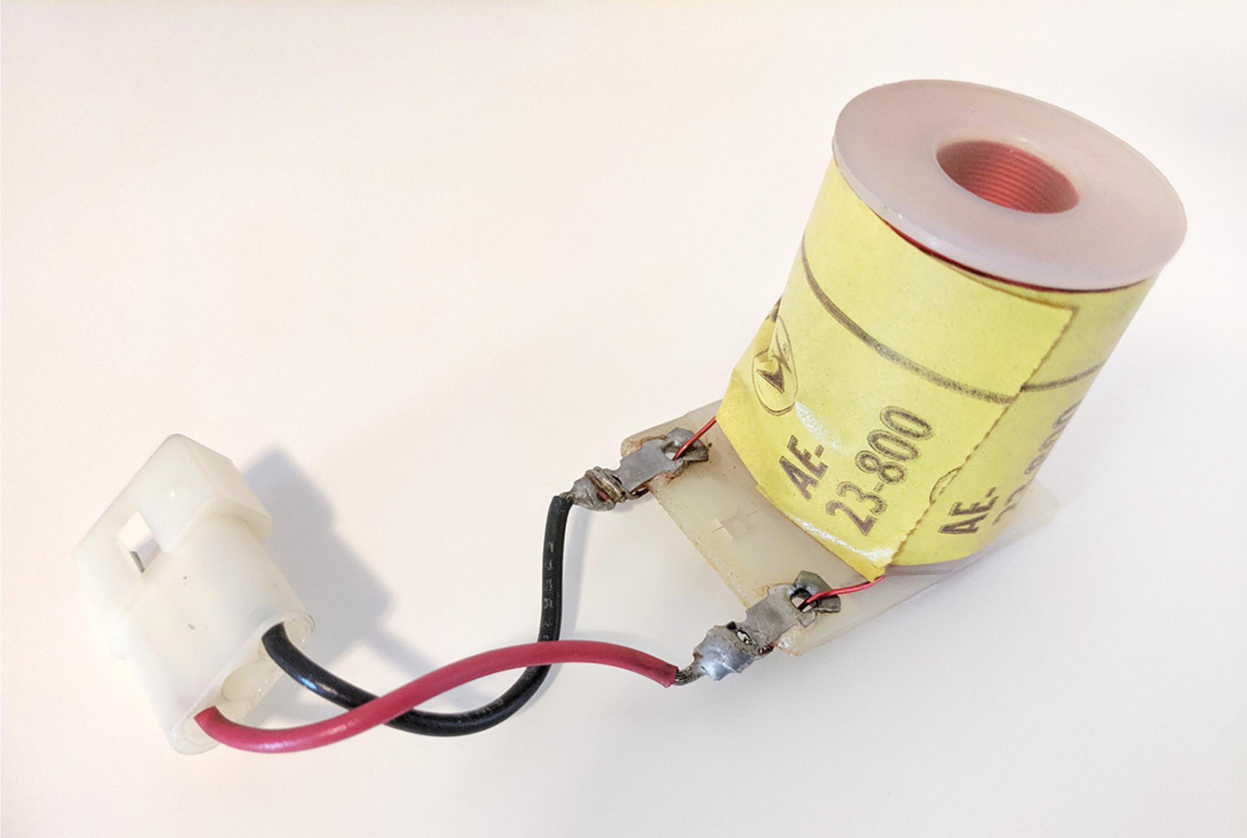
Warning: Please ensure that you have established *common ground between logic and coil power* before turning on high voltage on your coils (especially on homebrew machines). Ignoring this might lock on your coils, overheat them, burn down your house or kill you. We are serious, floating grounds are dangerous. If you are not an electrical engineer read the *guide about voltages and power*.

In a nutshell: You need to connect your logic ground (5V/12V) and your high voltage ground (48V or 80V). A *power entry or power filter board* is a convenient solution to solve this (and more) issues.

Always turn all PSUs off when connecting power or you might fry all boards at once. This is generally a good idea but even more important when connecting more than one power supply to a board.

IF YOU DID NOT UNDERSTAND WHAT THIS WARNING MEANS STOP NOW AND TRY TO UNDERSTAND IT. OTHERWISE YOUR HARDWARE WILL LIKELY BURST INTO FLAMES AND YOU NEED TO WAIT A FEW DAYS FOR A REPLACEMENT OR EVEN WORSE IT MIGHT KILL YOU. IGNORING THIS IS THE MOST COMMON CAUSE FOR BROKEN DRIVER BOARDS.

In MPF, you typically list all the coils in your machine in the *coils: section* of your machine configuration file, along with default options for them, like pulse times, PWM values, whether they can be enabled (held on), etc.



You don't typically work with coils directly, rather, you tend to add them to other devices once they've been defined (flippers, autofires, ball devices, diverters, etc). You can configure *Dual-wound Coils* on top of coils.

That said, it is possible to perform actions on coils directly, such as pulsing, enabling, or disabling them. You can do this via the *coil_player:* section of a config file or via the `coils:` section of a *show*.

Hardware

Connecting Coils

If you coil has more than two terminals please have a look at *Dual-wound Coils* because you got a dual-wound coil- If your coil has two terminals it is a single wound coil.

TODO: Add a picture a single wound coil with diode TODO: Add a electrical drawing a single wound coil with diode

In general, polarity does not matter for a coil. However, there might be a diode between the terminals of your coil which needs to be inverse to the voltage. This means that at the side of the stripe of the diode is where you connect high voltage. Normally, diodes are in the opposite direction but in this case this is intentional to short the coil when it deactivates (because of self-induction).

If you are unsure about the direction of your diode measure the resistance between the two terminals in both directions. You should get 1-300 ohms (depending on the coil) in on direction and almost zero in the other direction. Connect the coil in the direction with higher resistance. Plus/red plug of your multimeter would be where high voltage is connected. We recommend a diode on any coil to prevent interferences and damages to your driver boards.

Most machines use a common color for high voltage and an individual color for the return terminal of the coil. The "output" of your driver board is usually considered ground for the coil and the other terminal is connected to high voltage. Check with the documentation of your hardware platform to confirm this but it should be the case for all modern machines.

Video about wiring coils:

https://youtu.be/peVNGqb2Wp8

Strength and Current

Coils vary in strength relative to the pulse time you use. The strength of the magnetic field of a coil is a product of some constant `u`, the current `I` and the number of windings `N` divided by the length of the coil `L`: `B = u * I * N / L`

The length of coils in pinball is almost the same for most coils (3.5cm; so ignore that). However, the number of windings is not. Additionally, the thickness of the wire differs between coils which influences how much current can flow though the coil. Thicker wires generally means stronger coils. Unfortunately, this is not generally true for windings even though the formula above suggests it. The reason is for that more windings also mean longer wires which will result in higher resistance and less current. At least for typical coils in pinball more windings means slightly less powerful.

If you want to compare the strength of different coils you can get the number of windings and their resistance from one of the following pages:

- Pinball Medic Coil Chart
- Flippers.com Coil Resistance

Get windings `N` and resistance `R` from the chart. To get the current you can use `I = U/R`. Depending on your power supply `U` is either 48 or 70V. Length is roughly `3.5cm` for most coils.

Relative strength: `s = U / R * N / L`. More is stronger. In most cases you can leave out `L` as this is not terribly scientific anyway (and there is slightly more to it but this should be a good start). In general, reducing resistance `R` (by using thicker wires) will give you more powerful coils.

Video about electronics basics:

https://youtu.be/8ByqYkYKnFc

Config

This is an example for a single-wound coil:

```
coils:
  c_your_coil:
    number: 00   # depends on your platform and hardware
    default_pulse_ms: 20
```

This is an example for dual-wound coils which are configured separately:

```
coils:
  c_your_coil_main:
    number: 00   # depends on your platform and hardware
    default_pulse_ms: 20
  c_your_coil_hold:
    number: 01   # depends on your platform and hardware
    default_pulse_ms: 10
    default_hold_power: .2
```

See *Dual-wound Coils* for more details.

Related How To guides

- *Tutorial step 3: Get flipping!*

Adjust coil strength (pulse times)

Modern pinball controller systems that MPF use have the ability to precisely control how long (in milliseconds) the full power is applied to a coil. (Longer time = more power.) This is called the "pulse time" of a coil, as it controls how long the coil is pulsed MPF sends the coil a pulse command.

You can adjust this setting for all the coils in your machine, including flippers, trough ejects, pop bumpers, etc.

This is much nicer than the old days (even the 1990s WPC era) where pulse times were fixed, and you adjusted the strength of a mechanism by literally swapping out the coil with a stronger or weaker one!

Note: If you have *"dual wound"* coils, which are common for flippers, diverters, and other mechs which are "held" in the on position, you can use the pulse settings defined in this guide to control the initial "pulse" portion of that coil's activation.

Adjusting the pulse time is a bit of an art. If the pulse time is too long, you'll risk breaking something and the ball will fly off the mechanism too fast. Times that are two low will make the machine seem sluggish.

We suggest that you start with a slow time and slowly increase it until it feels right.

Unfortunately there's no universal pulse time setting that will work on every machine since the "pulse time" to "actual strength" mapping varies depending on:

- What type of coils you have (wire gauge and number of windings).
- How much voltage your power supply provides.
- How much current is available.
- How clean or worn your mechanisms, return springs, and/or coil sleeves are.
- How warm your coils are.

Pulse values can vary widely. One of our machines using new Williams flipper mechanisms with a 70vdc power supply has flipper pulse times of 14ms. Our 1974 Gottlieb Big Shot machine using the original flipper mechs has a pulse time over 100ms.

You adjust the pulse time for each coil by adding a `default_pulse_ms:` setting to the coil's entry in the *coils:* section of your machine config file. (Notice that you make this change in the `coils:` section of your config, not the section for the individual mech that coil is part of.)

If you don't specify a time for a particular coil, then MPF will a default pulse time of 10ms. (10ms is almost certainly too low, but it's a very safe default starting point.)

For example, for coils used in dual-wound flippers:

```
coils:
  c_flipper_left_main:
    number: 00
    default_pulse_ms: 20
  c_flipper_left_hold:
    number: 01
    allow_enable: true
  c_flipper_right_main:
    number: 02
    default_pulse_ms: 20
  c_flipper_right_hold:
    number: 03
    allow_enable: true
```

Or for single-wound flipper coils:

```
coils:
  c_flipper_left:
    number: 0
    allow_enable: true
    default_hold_power: 0.125
    default_pulse_ms: 20
  c_flipper_right:
    number: 1
    allow_enable: true
    default_hold_power: 0.125
    default_pulse_ms: 20
```

Again, you just need to play your game and see how it feels. Then keep on adjusting the `default_pulse_ms:` values up or down until your flippers feel right.

You might find that you have to adjust this `default_pulse_ms:` setting down the road too. If you have a blank playfield then you might think that your coils are fine where they are, but once you add some ramps you might realize it's too hard to make a ramp shot and you have to increase the power a bit. Later on when you have a real game, you can even expose these pulse settings to operators via the service menu.

Advanced settings vary based on hardware

In addition to being able to specify how long a coil is pulsed for, some pinball control systems allow you to control the power that's applied to the coil during the initial pulse. (So instead of 100% power for 50ms, you might be able to set a coil to 75% power for 60ms.)

See the *hardware documentation for your platform* for links to specific coil settings your hardware might allow.

Adjust coil hold power

In MPF, a coil is said to be "held" (or "enabled") any time it's activated for more than 255ms (since 255ms is the maximum pulse time for most platforms).

Most coils are only used in the "pulse" mode (slingshots, pop bumpers, trough and ball device ejects, etc.).

However, some pinball devices need to hold a coil on for longer (flippers, diverters, some older types of trough ball releases, etc.).

In MPF, you can adjust the power that's applied when these coils are held on past their initial pulse point.

Single-wound versus dual-wound coil holds

The way you configure coil holds depends on whether the coil in question is a "single wound" or "dual wound" coil. See the *Dual-Wound versus Single-Wound coils* guide for details.

Adjusting single-wound coil "hold" strength

Coils in MPF have a `default_hold_power:` setting which is used to control the amount of power that's applied to the coil after the initial pulse time.

The default_hold_power setting is a value from 0.0-1.0, with 0 being 0% power (off), and 1.0 being 100% power.

Consider the following example:

```
coils:
  some_coil:
    number:
    default_pulse_ms: 30
    default_hold_power: 0.250
```

In the example from a machine config file, the if the coil called *some_coil* is enabled (turned on) then that coil will receive full (100%) power for 30ms, and then after 30ms, the power drops down to 25%. The power will stay at 25% until the coil is turned off.

Note that the pinball control hardware cannot vary the voltage or current applied to a coil, rather it simulates lower power by rapidly pulsing the power. The example of `default_hold_power: 0.250` would equate to 25% power, which would mean the coil would get full power for 1ms, then it would get no power for 3ms, then full power for 1ms, etc (details vary per platform).

The default_hold_power: setting is valid with every type of pinball control system that MPF supports. However, some control systems have additional options which you can use to fine-tune how the hold power is applied to a coil.

See the *hardware documentation for your platform* for links to specific coil settings your hardware might allow.

The big question is what default_hold_power: setting is appropriate for your scenario? Unfortunately we don't have any good guidance for what your default_hold_power: values should be. Really you can just start with a value of 0.1 or 0.2 and then keep increasing it until your holds are strong enough not to break their hold when a ball hits them.

Adjusting dual-wound coil "hold" strength

If you have dual-wound coils then, the hold winding is designed to be held on, for long periods of time so you can safely keep it on full strength solid and don't have to mess with default_hold_power: settings.

The important caveat there is that the hold windings are designed around certain voltages. So if you have a dual-wound coil from a Stern machine that was designed to run at 48v, and you're using it in a new machine that's running at 70v, you'd probably want to use a default_hold_power: setting that's lower.

Again, you'll need to play with the settings to see what makes sense, and always choose the lowest one that works since if you have a setting that's too high, you probably won't know it until it's too late and the coil has burned up.

Recycle / "Cool Down" Time

Related Config File Sections
coils:

Recycle time is the time a coil will rest after it has been pulsed. This is either calculated as ratio on the pulse time (for instance, two times the pulse time) or as absolute time. In both cases this time is used to prevent thermal overheating of coils similar to hold_power.

If your machine constantly triggers a coil with 50ms pulse time for some reason then it would practically stay on permanently without recycle time. Howver, with a recycle factor of 2 (or 100ms cool down time) it would be enabled for at most 33% of the time.

How this recycle is implemented differs between platforms. MPF exposes a very basic interface to enable or disable `recycle` per coil. Usually, you want to keep it enabled. This is an example:

```
coils:
  c_coil_with_recycle:
    number:
    default_recycle: true
  c_coil_without_recycle:
    number:
    default_recycle: false
```

Some platforms allow you to fine tune the recycle time.

- *recycle_factor for OPP*
- *recycle_ms for FAST*

Dual-wound Coils

Related Config File Sections
dual_wound_coils:
flippers:

Warning: Please ensure that you have established *common ground between logic and coil power* before turning on high voltage on your coils (especially on homebrew machines). Ignoring this might lock on your coils, overheat them, burn down your house or kill you. We are serious, floating grounds are dangerous. If you are not an electrical engineer read the *guide about voltages and power*.

In a nutshell: You need to connect your logic ground (5V/12V) and your high voltage ground (48V or 80V). A *power entry or power filter board* is a convenient solution to solve this (and more) issues.

Always turn all PSUs off when connecting power or you might fry all boards at once. This is generally a good idea but even more important when connecting more than one power supply to a board.

IF YOU DID NOT UNDERSTAND WHAT THIS WARNING MEANS STOP NOW AND TRY TO UNDERSTAND IT. OTHERWISE YOUR HARDWARE WILL LIKELY BURST INTO FLAMES AND YOU NEED TO WAIT A FEW DAYS FOR A REPLACEMENT OR EVEN WORSE IT MIGHT KILL YOU. IGNORING THIS IS THE MOST COMMON CAUSE FOR BROKEN DRIVER BOARDS.

A *dual-wound coil* is a coil (solenoid) with two windings–one “strong” power (or “main”) winding for moving the coil, and a second weaker / lower-power winding for “holding” the coil in the active position.

image:: /mechs/images/dual_coil_no_diode.jpg

Dual-wound coils are typically used for flippers, diverters, gates, and other devices in pinball machines that need a strong initial movement followed by an extended hold period.

There are many places in MPF config files where you need to specify a coil name. Rather than adding dual-wound coil logic in many different sections of MPF, we have a dual-wound coil config where you can specify the settings for a particular dual-wound coil (and give it a new name), and then you can use that dual-wound coil anywhere in MPF that a coil is configured.

Hardware

Dual wound coils are like two coils in one but instead of two times two terminals they only have three terminals. Both coils share one of those terminals. Unfortunately, this is not standardized and different for different types of coils.

To make sure you connect things right you need a multimeter and measure the resistance between all three terminals. It might be wise to remove all free-fly diodes while measuring (or at least make sure to measure the inverse direction). You are looking for the main coil with low resistance (2-20 Ohm) and one with higher resistance (50 to 200 Ohm). You can look up the expected resistance in one of the linked charts in our *coil hardware section*.

Assumed you now got those three measurements:

- Terminal 1 to 2: 4 Ohm
- Terminal 2 to 3: 124 Ohm
- Terminal 1 to 3: 120 Ohm

What does that mean? It means that your main coil is between terminal 1 and 2 and your hold coil is between terminal 1 and 3. Terminal 2 and 3 is just the sum of the resistance of both coils. In general, the highest of the three readings is the combination you want to remove from your list.

How do you connect that? Typically, driver boards connect your coils to ground so you connect power to the terminal which is common between both coils. In this case this would be terminal 1. Terminal 2 and 3 would be connected to your driver board. See the following picture for example:

The common terminal on the right (terminal 1) has both wires connected. You see that the diode band is facing in that direction and that two copper wires for the coil are connected here. One wire is a bit thicker, which means less resistance thus more current thus stronger. Hence, the thinner wire is for the hold position. You might not be able to spot that on every coil.

Please note, that you not just connect two wires across each diode. But that one pair of wires is across one diode and the other pair of wires is across two diodes. If you don't do it that way you will fry one of your FETs on the driver board.

> **Warning:** Please make sure that any diodes on your coil are in reverse to the voltage (i.e. the stripe needs to be at the HV side). This is often not the case for older coils as they have been connected differently in older machines. Ignoring this will fry the FET on your driver board.

See *coil hardware* for more details about the current, resistance, number of windings and the strength of coils.

Config

This is an example for dual-wound coils which are configured separately:

```
coils:
  c_your_coil_main:
    number: 00   # depends on your platform and hardware
    default_pulse_ms: 20
  c_your_coil_hold:
    number: 01   # depends on your platform and hardware
    default_pulse_ms: 10
    default_hold_power: .2
```

On top of that you can configure *dual_wound_coils:* or other devices such as *flippers:*.

Related How To guides

- *Dual-Wound versus Single-Wound coils*
- *How to configure dual-wound flippers*

Related Events
None

Dual-Wound versus Single-Wound coils

> **Warning:** Please ensure that you have established *common ground between logic and coil power* before turning on high voltage on your coils (especially on homebrew machines). Ignoring this might lock on your coils, overheat them, burn down your house or kill you. We are serious, floating grounds are dangerous. If you are not an electrical engineer read the *guide about voltages and power*.
>
> In a nutshell: You need to connect your logic ground (5V/12V) and your high voltage ground (48V or 80V). A *power entry or power filter board* is a convenient solution to solve this (and more) issues.

Always turn all PSUs off when connecting power or you might fry all boards at once. This is generally a good idea but even more important when connecting more than one power supply to a board.

IF YOU DID NOT UNDERSTAND WHAT THIS WARNING MEANS STOP NOW AND TRY TO UNDERSTAND IT. OTHERWISE YOUR HARDWARE WILL LIKELY BURST INTO FLAMES AND YOU NEED TO WAIT A FEW DAYS FOR A REPLACEMENT OR EVEN WORSE IT MIGHT KILL YOU. IGNORING THIS IS THE MOST COMMON CAUSE FOR BROKEN DRIVER BOARDS.

It's common for pinball machines to include coils that are "held on" for periods of time longer than the maximum pulse time of 255ms. The obvious example of this is for flipper coils, though other types of devices use these too. (Diverters, the trolls in *Medieval Madness*, certain ball release coils, etc.)

In many cases, these types of coils need to have strong initial pulses to quickly move the mechanism from its resting to active position, but they also need to be able to be held "on" for a long period of time.

These two requirements are conceptually incompatible.

The way you make a coil strong is you give it lots of power and make it really big. Unfortunately the byproduct of that is heat, which means if you make a nice, big, powerful coil that's strong enough to move the mechanism with the quick power it needs, then when the coil is left in the "on" state, it generates so much heat that it will burn up the coil. :(

Fortunately the pinball companies solved this 60+ years ago with the concept of "dual wound" coils. A dual-wound coil is essentially two separate coils in one. (There are literally two separate wires wrapped around the coil sleeve instead of one.)

Dual-wound coils have a strong (often called the "main" or "power") winding which is used for the initial "kick" of the coil, and they also have a lower-powered ("hold") winding which is used to hold the active position.

Note: You can tell if a coil is dual-wound because the coil will have three wire connection points instead of two. There's a power winding connector, a hold winding connector, and a common connector that's shared by both.

The way these are used is that the strong winding is pulsed initially (usually for a fraction of a second) to provide the initial strength to move the mechanism, then it cuts off, leaving just the weaker hold winding active to keep the mechanism active. The hold winding can safely be enabled for a long time, even multiple minutes. When the machine wants to disable the device (or when the player releases the flipper button in the case of a flipper), the power to the hold winding is cut, and a spring causes the mechanism to return to the initial position.

Transitioning from the power to the hold winding: The old way

In old pinball machines (from the 1940s through the early 2000s), the "transition" from the power winding to the hold winding was purely mechanical and done using something called an "end of stroke" (EOS) switch.

The EOS switch is a physical leaf switch in the mechanism under the playfield with a switch that is mechanically opened by the movement of the device. When the coil is first activated, the current flows to both the power and the hold windings, and the mech starts to move. A few fractions of a second later, the mech reaches its full "up" position, and a little arm under it hits the EOS switch which opens it and breaks the connection to the power winding, leaving only the hold winding energized.

When the hold winding is de-energized, the spring causes the mechanism to move back to the original position, and the EOS switch is closed (by the movement of the mech) meaning that the next time the mech is activated, the current will again flow to both the power and hold windings.

Advantages of using this "old style" EOS switch

- It's simple. No computers or fancy timing has to be involved, and the transition from the power to the hold windings is automatic.
- If a coil gets dirty, gummed up, or weak, the transition from the power to the hold winding always occurs only after the mech is all the way in the "active" position.
- Only a single "driver" connection from the control system is needed since that single control line is used for both the power and hold windings.

Downsides to using this "old style" EOS switch

- No fine tuning. Since the transition from the power to the hold winding is purely mechanical, you can't change the power of the mechanism unless you physically switch out the coil and/or change the voltage used.
- For flippers, you don't get any "novelty" flipper modes. You can't do things like "weak flippers" or "no hold flippers" since the flipper behavior is mechanically controlled.

Transitioning from the power to the hold winding: The modern way

Modern machines do not use EOS switches in the same way they have been used in older machines.

The main reason for this is that modern pinball control systems (including all the *control systems* that MPF supports) have the ability to activate coils with millisecond-level precision (something that was not possible even in 1990s WPC machines).

Using flippers as an example, in modern machines, when the player presses the flipper button, the control system will send current to both the power and hold windings at the same time, and then at a very precise moment (e.g. 27ms later or 14ms later or whatever), the control system will cut off the power winding, leaving just the hold winding active.

This has the same effect of the mechanical EOS switch in that the power winding is only used for the initial power motion, and the lower-current hold winding is then used to keep the flipper in the up position.

Advantages of using the modern transition from power to hold

- You can fine-tune coil strength by changing settings in software.
- You can use novelty modes like weak flippers, no hold flippers, etc.

Downsides of using the modern transition from power to hold

- You have to play with your settings to get them right.
- A dirty, gummed up, or worn-out coil or mechanism might mean that the initial power timing setting you originally configured might not be strong enough to move the mechanism all the way into the "up" position.

Single-wound coils

So far both options (EOS and non-EOS) we discussed use dual-wound coils with power and hold windings.

However there's a third option that some modern machines use as well. The third option is to use more traditional (e.g. "single wound") coils for your machine that do not have the dual "power" and "hold" windings.

Of course you might be thinking, "How does that work? Wouldn't the coil burn up if the mechanism was active for too long?"

This is another case where modern technology can be used to address that.

In electronics, there's a concept called "Pulse Width Modulation" (or "PWM"), which (in this case) basically means the control hardware turns the power on and off really fast. (Like, hundreds of times per second.)

So the way this works is that you have a high-powered, strong coil which is activated a full strength in order to provide the strong initial motion. However once the mechanism is in the up position (based on either an EOS switch, or based on the millisecond-level precise timing), the control system stops powering that coil at 100% and instead cuts the power back (using that PWM thing) to a smaller percent (like maybe 12.5% or 25% or so). That reduced power is enough to keep the mech in the up position, but not enough to cause the coil to overheat and burn out.

Advantages to using single-wound coils

- You only need a single driver output per coil (instead of two).
- You can still do the modern things, like use software to tune the strength of the coil and novelty flipper modes.

Downsides to using single-wound coils

- You have to figure out the PWM (low power) settings which need to be strong enough to hold the mechanism up but not too strong so they don't burn it up.
- Sometimes the PWM "hold" makes an annoying buzzing sound (since the power is being turned on and off hundreds of times per second).

We should note that the decision to use a single-wound versus dual-wound flipper coil is technically a separate decision from whether or not to use an EOS switch. See the *Flipper end-of-stroke (EOS) switches* for more on that decision.

Which option should you choose?

Ok, so basically there are three options for coils that need to be held on for more than 255ms:

- Dual-wound, with a mechanical EOS switch to transition from power to hold.
- Dual-wound, with the control system timing to transition from power to hold.
- Single-wound

The good news is that MPF supports all three options.

If you're retheming an existing machine, and you're using the original driver boards and power supplies, then you should probably just use whatever method was used in that machine and keep it simple.

If you're building a new machine, most people choose the second option, where you use a dual-wound coil but with the transition of the power to hold windings done via software and the modern control systems. The reasons for this include:

- It's simple. You don't have to mess with trying to figure out the PWM timings for the hold winding.
- It works. You know the hold winding was designed to be held on at full power, so you don't have to worry about breaking things.
- It's less wear-and-tear and emissions. Rapidly cycling power (in the PWM way) for the hold phase in a single-wound coil has the potential to add wear to the components in your system and potential to cause EMI emissions.

People have also pointed out that Stern's S.A.M. system (which they used in from about 2006-2015) used the single-wound PWM-style flippers, but then with the SPIKE system (from 2015 onwards) went back to the dual-wound computer controlled option for a while. However, they later switched back to single-wound PWM-style flippers. We can only speculate why they did that and it might involve that dual-wound flippers are easier to control from software with a new control system.

Really the only reasons to use the single-wound coils are:

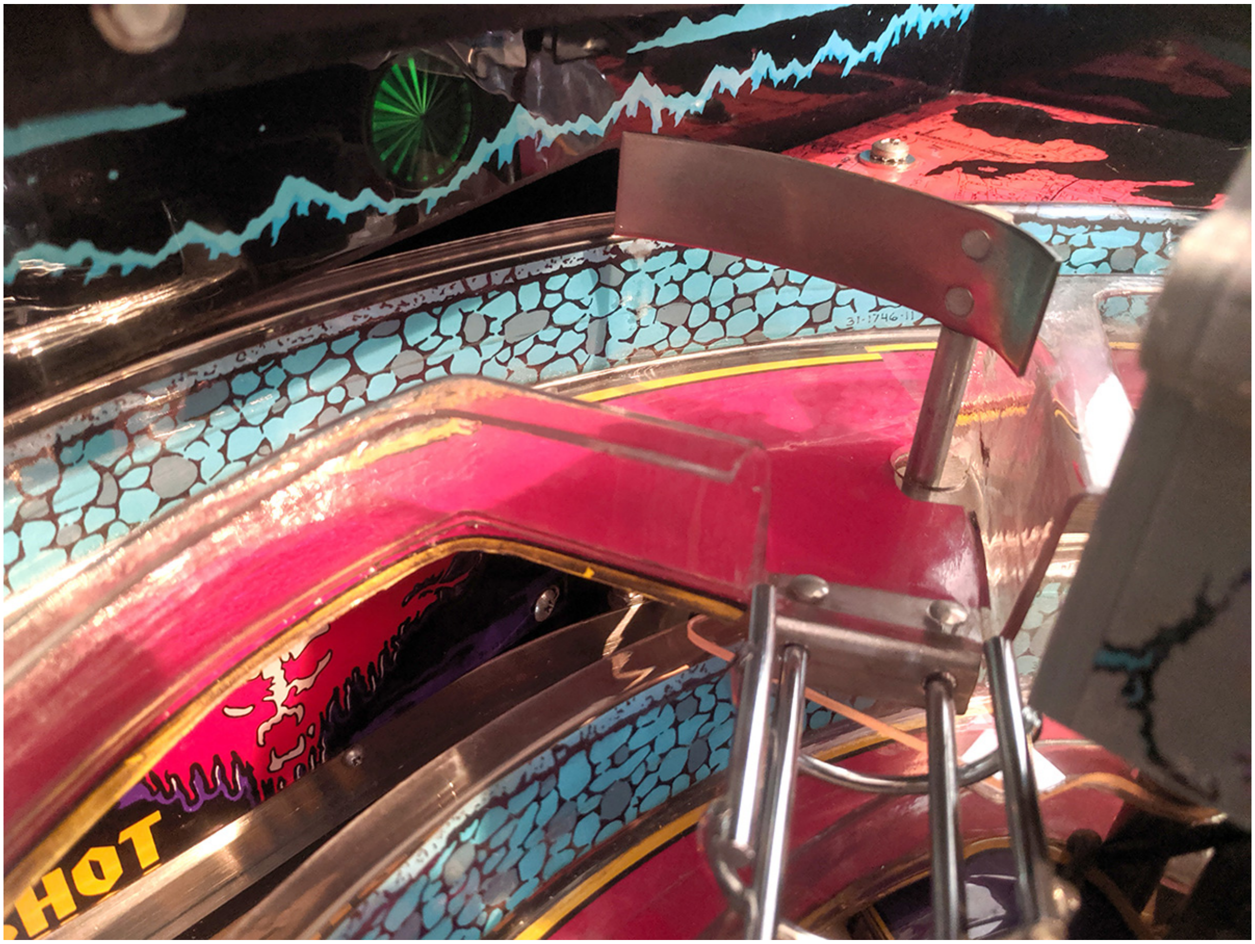
- You already have mechanisms that use single-wound coils
- You're running out of driver outputs in your control system and you don't want to "waste" two drivers per mech.
- Single-wound are cheaper to produce

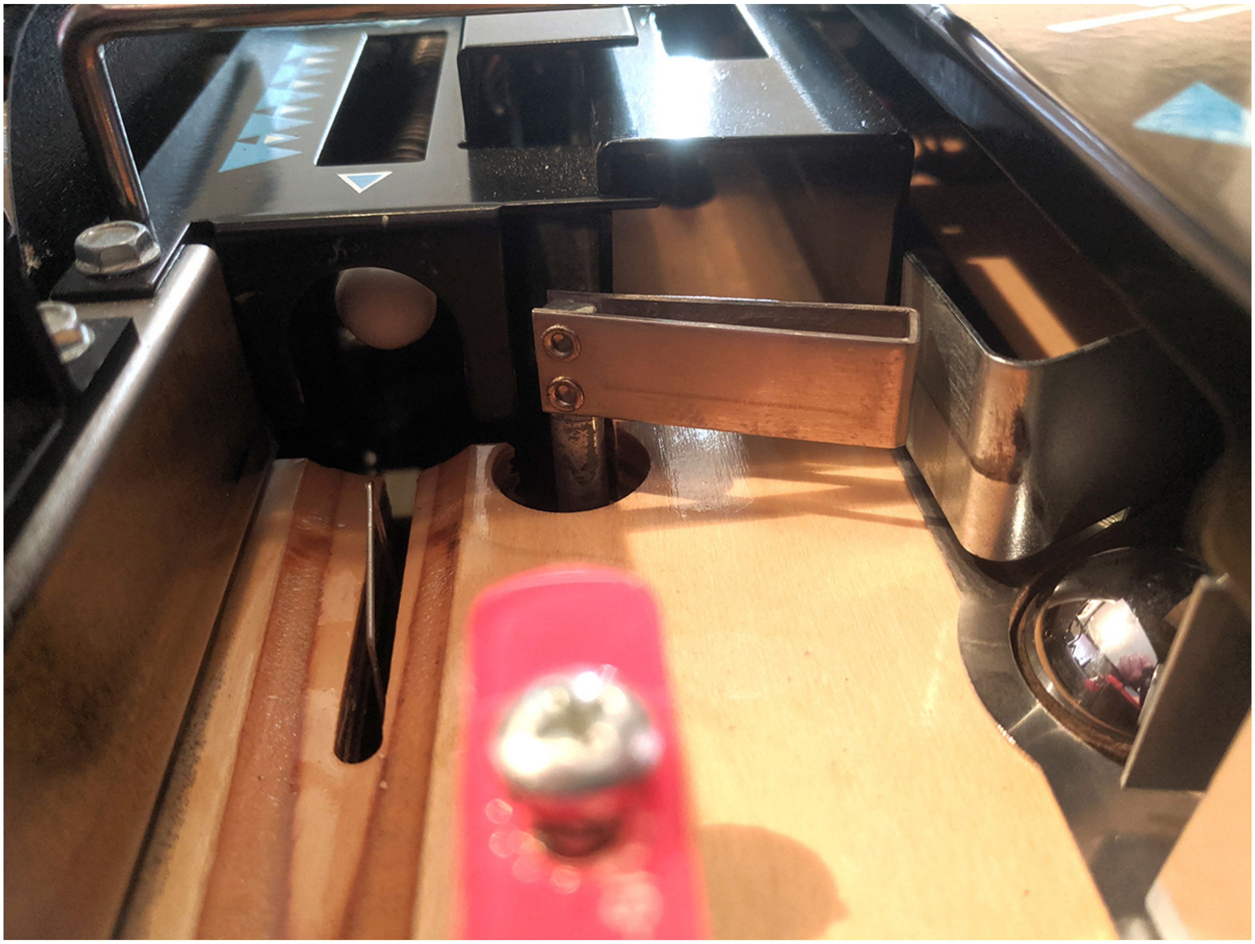
Related Events
None

Diverters

Related Config File Sections
diverters:

In MPF, a diverter (sometimes spelled "divertor") is anything that alters the path of the ball based on the state it's in, including:





- A traditional diverter which is a metal flap at the end of a rod, typically used on ramps to “divert” the ball one way or the other.
- A coil-controlled post that pops up (or down) to let the ball either pass over it or bounce back in some other direction. (This is sometimes called an “up/down” post.)
- A coil-controlled gate, typically which only allows the ball to flow through it in a single direction, but lifted out of the way via a coil when active which allows the ball to travel through it in both directions.
- A “trap door” pop-up which captures the ball when it’s up but lets the ball roll over it to another shot when it’s down. (Like the trap door / basement in Theatre of Magic.)
- A single drop target that blocks the entrance to a shot when it’s up, such as in the back of the saucer in Attack from Mars or the ones that block the ramps in Ghostbusters.
- Something else completely custom, such as the Ringmaster in Cirqus Voltaire. (When it’s up the ball can hit it and drop down under the playfield, and when it’s down the ball rolls over it and hits standup targets behind it.)

At this point you might be thinking, “Wait, you consider a trap door or the Ringmaster to be a diverter?? What???” But if you think about it from the perspective of pinball software, yeah, trap doors and the Ringmaster *are* diverters because when then are not active, a ball shot to them goes towards one place, and when they’re active, a ball is “diverted” to go somewhere else.

Note: MPF's diverters are integrated with *Ball Devices* and MPF's ball management and routing system so they can be used to ensure that MPF is able to move balls to where they need to be.

Most diverters are held in their "on" position as long as their driver coil enabled, and then when they're disabled they return back to their off position. That said, some are different. The Ringmaster has a motor which raises and lowers it, and drop targets have coils that are just pulsed to raise/lower them, so this is not a hard and fast rule.

So based on all that, let's look at how the MPF actually handles diverters. At the most basic level, most diverters are just a coil, so fundamentally we don't really need to do anything special to control a diverter. As a game programmer you just need to enable a coil. But if you want to program your game code to control a diverter, there's a lot of glue you need to fully integrate it into your machine, and that's the glue that we've pre- written into our diverter device code.

For example, many diverters attached to ramps do not hold their coils in the "on" position for the entire time that they're on. Instead they use the ramp entry switch to see when a ball is coming their way, and when one is they quickly activate so they can catch the ball in time to divert it. They also typically have a timeout where they deactivate themselves if they don't actually see a ball get diverted, (like with a weak ramp shot that trips the ramp entry switch but that isn't powerful enough to make it all the way up the ramp to the diverter.)

MPF's diverter devices also include support for automatic enabling and disabling (based on events), and they include intelligence to know which target devices a diverter will send a ball to when it's enabled or disabled.

Understanding the difference between "enabling" and "activating" diverters

When talking about diverters in MPF, we use the terms *activate* and *enable* (as well as *deactivate* and *disable*). Even though these words sound like they're the same thing, they're actually different, so it's important to understand them.

When a diverter is *active*, that means it's physically activated in its active position. A diverter that is *enabled* means that it's ready to be activated, but it's not necessarily active at this time. To understand this, let's step through an example.

Imagine a typical ramp in a pinball machine which has one entrance and two exits. These kinds of ramps usually have a diverter at the top of them that can send the ball down one of the two paths. When the diverter is *inactive* (its default state), the ball goes down one path, and when the diverter is *active*, the ball is sent down the other path (perhaps towards a ball lock).

There is typically an entrance switch on the ramp which lets the game know that a ball is potentially headed towards that diverter, so when the game wants to route the ball to the "other" ramp exit, rather than turning on that diverter and holding it on forever, the game just watches for that ramp entry switch and then quickly fires the diverter to route the ball to the other exit. Then once the ball passes by the diverter, it hits a second switch which turns off the diverter. (Typically the diverter activation also has a timeout which is used when a weak shot is made where the ball trips the ramp entrance switch but doesn't actually make it all the way up the ramp to the diverter.)

So in MPF parlance, we say that the diverter is *enabled* whenever it's ready to be fired, but it's not actually *active* until the coil is physically on.

Again using our example, let's say we have a ramp with a diverter, and when that diverter is *active* it sends a ball into a lock. When the game starts, the diverter is *disabled* and *inactive*. Ramp shots just go up the ramp and come out the default path, and the diverter ignores the ramp entrance switch.

Then when the player does whatever they need to do to light the lock, the diverter is *enabled*. At this point the diverter is *not* active since it's not actually firing, but it's *enabled* (which means it's ready to fire) and the diverter is watching that ramp entrance switch. (So the diverter is *enabled* but *inactive*.) Then when the player shoots the ball up that ramp, the diverter sees the ramp entrance switch hit and the diverter activates. (So now the diverter is *enabled* and *active*.)

Then once the ball passes by the diverter, the diverter deactivates. At this point whether the diverter is disabled or enabled depends on the game logic. If the lock should stay lit, then the diverter remains enabled even though it's not *active*, and if the player has to do something else to re-light the lock, then the diverter is *disabled* and *inactive*.

Hopefully that makes sense? :)

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for diverters is `device.diverters.<name>`.

active Boolean (true/false) as to whether this diverter is actively on and in the powered state.

enabled Boolean (true/false) as to whether this diverter is enabled (meaning it will be activated when a ball approaches it).

eject_state Boolean (true/false) which shows whether this diverter will be activating to route a ball eject from an upstream ball device.

Related How To guides

- *Dual Coil Diverter*
- *Up-Down Ramps*

Related Events

- *diverter_(name)_enabling*
- *diverter_(name)_disabling*
- *diverter_(name)_activating*
- *diverter_(name)_deactivating*

Up-Down Ramps

Related Config File Sections
diverters:

Some machines have ramps which can be moved up and down. Those mechanism typically act as a diverter and should be configured as such.

Hardware

Up-Down ramps either work with one coil or two coils. Single-coil ramps use the coil to move the ramp up or down temporarily Typically, they use a spring or gravity to move it back, However, since the coil has to stay energized those ramps can only be active for a short amount of time. Two-coil ramps only pulse one coil to move the ramp up or down. This poses the advantage that the ramp stays at one position without energizing a coil. However, this adds complexity (and a second coil) and is only used if the ramp has to stay at both positions for extended periods of time.

Some part numbers:

- Getaway: B-12576 assembly
- RFM: A-22989
- Apollo 13: #500-6044-00-44 assembly
- Golden Eye: #515-6494-00

Config

Up-Down ramps are configured like a normal diverter:

```
diverters:
  up_down_one_coil:
    activation_coil: c_ramp1_up
    type: hold
  up_down_two_coils:
    activation_coil: c_ramp2_up
    deactivation_coil: c_ramp2_down
    type: pulse
```

Related How To Guides
Diverters

Using a Servo as Diverter

Related Config File Sections
diverters:
servos:

You can use a servo as a diverter by tying it into a diverter using events. Specifically, we are using *diverter_(name)_deactivating* and *diverter_(name)_activating*. This is an example:

```
diverters:
  d_diverter:
    debug: true
    feeder_devices: bd_trough
    targets_when_active: playfield
    targets_when_inactive: bd_target
servos:
  s_diverter:
    number:
    positions:
      0.7: diverter_d_diverter_activating
      0.2: diverter_d_diverter_deactivating
```

This diverter will not wait for the servo to reach the position. If you need that let us know in the forum.

Using a Stepper as Diverter

Related Config File Sections
diverters:
steppers:

You can use a stepper as a diverter by tying it into a diverter using events. Specifically, we are using *diverter_(name)_deactivating* and *diverter_(name)_activating*. This is an example:

```
diverters:
  d_diverter:
    debug: true
    feeder_devices: bd_trough
    targets_when_active: playfield
    targets_when_inactive: bd_target
steppers:
  s_diverter:
    number:
    named_positions:
      20: diverter_d_diverter_activating
      400: diverter_d_diverter_deactivating
```

This diverter will not wait for the stepper to reach the position. If you need that let us know in the forum.

Dual Coil Diverter

Related Config File Sections
diverters:
dual_wound_coils:

In this example we use a standard flipper mechanism with a dual wound coil as a diverter. Much like a flipper, we'll want to control the main coil for enabling the diverter, and then the hold coil to hold it in the active position for as long as you need.

Config

First we need to define the coils in our hardware section:

```
coils:
  c_diverter_upper_right_main:
    number: 25
    default_pulse_ms: 4
    default_hold_power: 0.2
  c_diverter_upper_right_hold:
    number: 26
    allow_enable: true
```

Next we'll define the dual wound coil for the diverter to use:

```
dual_wound_coils:
  c_diverter_dualcoil:
    hold_coil: c_diverter_upper_right_hold
    main_coil: c_diverter_upper_right_main
```

Then we define the Diverter itself:

```
diverters:
  ramp_diverter:
    activation_coil: c_diverter_dualcoil
    type: hold
    activation_time: .5s
    activation_switches: s_r_rampexit, s_l_rampexit
    enable_events: ball_started
    disable_events: ball_ended
```

Related How To Guides
Diverters

Flippers

Related Config File Sections
flippers:

Warning: Please ensure that you have established *common ground between logic and coil power* before turning on high voltage on your coils (especially on homebrew machines). Ignoring this might lock on your coils, overheat them, burn down your house or kill you. We are serious, floating grounds are dangerous. If you are not an electrical engineer read the *guide about voltages and power*.

In a nutshell: You need to connect your logic ground (5V/12V) and your high voltage ground (48V or 80V). A *power entry or power filter board* is a convenient solution to solve this (and more) issues.

Always turn all PSUs off when connecting power or you might fry all boards at once. This is generally a good idea but even more important when connecting more than one power supply to a board.

IF YOU DID NOT UNDERSTAND WHAT THIS WARNING MEANS STOP NOW AND TRY TO UNDERSTAND IT. OTHERWISE YOUR HARDWARE WILL LIKELY BURST INTO FLAMES AND YOU NEED TO WAIT A FEW DAYS FOR A REPLACEMENT OR EVEN WORSE IT MIGHT KILL YOU. IGNORING THIS IS THE MOST COMMON CAUSE FOR BROKEN DRIVER BOARDS.

Flippers are probably the first thing you think of when you think about building your own pinball machine. In fact when most people get their own hardware and start drilling holes in a piece of plywood, the first visible thing they do is to get their flippers flipping.

MPF has support for lots of different kinds of flippers (as there are many different ways they've been wired over the years), as well as a lot of different options for how flippers are fine tuned.

MPF also has support for various "novelty" flipper modes (no-hold flippers, reversed flipper buttons, *weak flippers*, etc.)

We recommend you read the *Dual-Wound versus Single-Wound coils* guide to understand the difference between "dual wound" and "single wound" coils, as flippers in pinball machines can be either type.

You should also probably read the *EOS Switches guide* if your machine has flipper EOS switches. (In general EOS switches are not needed for flippers with MPF.)

See *coil hardware* for more details about the current, resistance, number of windings and the strength of coils.

Debounce and Recycle on Flipper Coils

In MPF you can *configure debounce for each switch* and *recycle for each coil*. However, both will be overwritten when you enable flippers. Debounce will be set to `quick` and `recycle` will be disabled. In some platforms MPF might reconfigure your switch debounce settings when activating the hardware rules (if the platform does not allow separate settings) which might lead to more switch events when flippers are active.

Generally, this is how flipper work in most machines and this is how players will expect flippers to behave. If you want to change this let us know in the forum (or you could change it in by overloading the flipper device class).

Default Events

MPF contains built-in support for the flipper cancel combo. If you add the tag `left_flipper` to your left flipper switch, and `right_flipper` to your right flipper switch, then whenever the player hits both flippers at the same time, an MPF event called *flipper_cancel* will be posted. This is implemented as *combo switch*.

Additionally, MPF contains a default *timed switch* for flipper cradle. It will post `flipper_cradle` when a player cradles a ball for 3s. Later it will post `flipper_cradle_release` when the player releases the ball.

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for flippers is `device.flippers.<name>`.

enabled Boolean (true/false) which shows whether this ball hold is enabled.

Related How To guides

How to configure dual-wound flippers

> **Warning:** Please ensure that you have established *common ground between logic and coil power* before turning on high voltage on your coils (especially on homebrew machines). Ignoring this might lock on your coils, overheat them, burn down your house or kill you. We are serious, floating grounds are dangerous. If you are not an electrical engineer read the *guide about voltages and power*.
>
> In a nutshell: You need to connect your logic ground (5V/12V) and your high voltage ground (48V or 80V). A *power entry or power filter board* is a convenient solution to solve this (and more) issues.

Always turn all PSUs off when connecting power or you might fry all boards at once. This is generally a good idea but even more important when connecting more than one power supply to a board.

IF YOU DID NOT UNDERSTAND WHAT THIS WARNING MEANS STOP NOW AND TRY TO UNDERSTAND IT. OTHERWISE YOUR HARDWARE WILL LIKELY BURST INTO FLAMES AND YOU NEED TO WAIT A FEW DAYS FOR A REPLACEMENT OR EVEN WORSE IT MIGHT KILL YOU. IGNORING THIS IS THE MOST COMMON CAUSE FOR BROKEN DRIVER BOARDS.

This guide shows you how to configure dual-wound flippers in MPF. If you don't know what "dual-wound" flippers are, or whether you have them, take a look at the coil that your flipper uses. If it has three wires (or three tabs to connect three wires), then it's a dual-wound coil and this guide is for you.

If it has two wires (or two tabs), then read the *How to configure single-wound flippers* guide.

Read more about "dual wound" versus "single wound" coils in the *Dual-Wound versus Single-Wound coils* guide.

See *coil hardware* for more details about the current, resistance, number of windings and the strength of coils. See *dual-wound hardware* for details about how to find out which terminals on your coils are hold, which are the main coil and how to connect them.

1. Add your flipper buttons

First, make sure you have entries in your machine config for your flipper buttons.

Here's an example `config.yaml` with two switches added:

```
switches:
  s_left_flipper:
    number: 1
    tags: left_flipper
  s_right_flipper:
    number: 2
    tags: right_flipper
```

You can pick whatever names you want for your switches. We chose `s_left_flipper` and `s_right_flipper`.

Note that we configured this switches with numbers `1` and `2`, but you should use the actual switch numbers for your control system that the flipper buttons are connected to. (See *How to configure "number:" settings* for instructions for each type of control system.)

We also added tags called `left_flipper` and `right_flipper`. These are optional, but recommended. The reason is that MPF includes a *combo switch* feature which posts events when player switches are held in combination. If you add these tags to your flipper switches, an event called *flipper_cancel* will be posted when the player hits both flipper buttons at the same time which you can use to cancel shows and other things you want the player to be able to skip.

2. Add your flipper coils

Next you need to add entries for your flipper coils to your machine-wide config. These will be added to a section called `coils:`. Since we're using dual-wound coils, there will actually be two coil entries for each coil—one for the power (main) winding, and one for the hold winding.

```
coils:
  c_flipper_left_main:
    number: 0
  c_flipper_left_hold:
    number: 1
    allow_enable: true
  c_flipper_right_main:
    number: 2
  c_flipper_right_hold:
    number: 3
    allow_enable: true
```

Again, the `number:` entries in your config will vary depending on your actual hardware, and again, you can pick whatever names you want for your coils.

Also note that the two hold coils have `allow_enable: true` entries added. (In MPF config files, values of "yes" and "true" are the same.) The purpose of the `allow_enable: true` setting is that as a safety precaution, MPF does not allow you to enable (that is, to hold a coil in its "on" position) unless you specifically add `allow_enable: true` to that coil's config.

So in the case if your flippers, the hold coil of a flipper needs to have `allow_enable: true` since in order for it to act as a flipper, that coil needs to be allowed to be enabled (held on).

3. Add your flipper entries

At this point you have your coils and switches defined, but you can't flip yet because you don't have any flippers defined. Now you might be thinking, "Wait, but didn't I just configure the coils and switches?" Yes, you did, but now you have to tell MPF that you want to create a flipper mechanism which links together the switch and the coils to become a "flipper".

You create your flipper mechanisms by adding a `flippers:` section to your machine config, and then specifying the switch and coils for each flipper that you defined in Steps 1 and 2.

Here's what you would create based on the switches and coils we've defined so far:

```
flippers:
  left_flipper:
    main_coil: c_flipper_left_main
    hold_coil: c_flipper_left_hold
    activation_switch: s_left_flipper
  right_flipper:
    main_coil: c_flipper_right_main
    hold_coil: c_flipper_right_hold
    activation_switch: s_right_flipper
```

4. Enabling your flippers

By default, MPF only enables flippers when a game is in progress. So if this is a first-time config and you haven't configured your ball devices and start button and everything, you can't actually start a game yet, which means you can't test your flippers.

Fortunately we can get around that by configuring your flippers to just automatically enable themselves when MPF starts. To do this, add the following entry to each of your flippers in your config file:

```
enable_events: machine_reset_phase_3
```

So now the `flippers:` section of your config file should look like this:

```
flippers:
  left_flipper:
    main_coil: c_flipper_left_main
    hold_coil: c_flipper_left_hold
    activation_switch: s_left_flipper
    enable_events: machine_reset_phase_3
  right_flipper:
    main_coil: c_flipper_right_main
    hold_coil: c_flipper_right_hold
    activation_switch: s_right_flipper
    enable_events: machine_reset_phase_3
```

5. Configure your control system hardware

At this point your flipper configuration is technically complete, though there are two other important things you may have to do first:

If you're using physical hardware, you may need an additional section in your machine config for your control system. (For example, FAST Pinball and Open Pinball Project controllers require a one-time port configuration, etc.) See the *control system documentation* for details.

6. Adjust your flipper power

As a safety precaution, MPF uses very low (10ms) default pulse times for coils. In most cases, 10ms will not be enough power to physically move the flippers when you hit the button. (You might hear them click or buzz without actually seeing them move.)

So check out the documentation in the coils section for instructions on how to adjust the *pulse power* and the *hold power* for the coils you're using for your flippers.

Here's the complete config

Here's the complete machine config file (or sections of the machine config file) we created in this How To guide:

Listing 1: /config/config.yaml

```yaml
#config_version=5

hardware:
  platform: fast
  driverboards: fast

switches:
    s_left_flipper:
        number: 0-0
        tags: left_flipper
    s_right_flipper:
        number: 0-1
        tags: right_flipper

coils:
    c_flipper_left_main:
        number: 0-0
        default_pulse_ms: 30
    c_flipper_left_hold:
        number: 0-1
        default_hold_power: 1.0
    c_flipper_right_main:
        number: 0-2
        default_pulse_ms: 30
    c_flipper_right_hold:
        number: 0-3
        default_hold_power: 1.0

flippers:
    left_flipper:
        main_coil: c_flipper_left_main
        hold_coil: c_flipper_left_hold
        activation_switch: s_left_flipper
        enable_events: machine_reset_phase_3
    right_flipper:
        main_coil: c_flipper_right_main
        hold_coil: c_flipper_right_hold
        activation_switch: s_right_flipper
        enable_events: machine_reset_phase_3
```

Related How To guides

- *Dual-wound Coils*
- *Dual-Wound versus Single-Wound coils*

How to configure single-wound flippers

Warning: Please ensure that you have established *common ground between logic and coil power* before turning on high voltage on your coils (especially on homebrew machines). Ignoring this might lock on your coils, overheat them, burn down your house or kill you. We are serious, floating grounds are dangerous. If you are not an electrical engineer read the *guide about voltages and power*.

In a nutshell: You need to connect your logic ground (5V/12V) and your high voltage ground (48V or 80V). A *power entry or power filter board* is a convenient solution to solve this (and more) issues.

Always turn all PSUs off when connecting power or you might fry all boards at once. This is generally a good idea but even more important when connecting more than one power supply to a board.

IF YOU DID NOT UNDERSTAND WHAT THIS WARNING MEANS STOP NOW AND TRY TO UNDERSTAND IT. OTHERWISE YOUR HARDWARE WILL LIKELY BURST INTO FLAMES AND YOU NEED TO WAIT A FEW DAYS FOR A REPLACEMENT OR EVEN WORSE IT MIGHT KILL YOU. IGNORING THIS IS THE MOST COMMON CAUSE FOR BROKEN DRIVER BOARDS.

This guide shows you how to configure single-wound flippers in MPF. If you don't know what "single-wound" flippers are, or whether you have them, take a look at the coil that your flipper uses. If it has two wires (or two tabs to connect two wires), then it's a single-wound coil and this guide is for you.

If it has three wires (or three tabs), then read the *How to configure dual-wound flippers* guide.

Read more about "dual wound" versus "single wound" coils in the *Dual-Wound versus Single-Wound coils* guide.

See *coil hardware* for more details about the current, resistance, number of windings and the strength of coils.

1. Add your flipper buttons

First, make sure you have entries in your machine config for your flipper buttons.

Here's an example `config.yaml` with two switches added:

```
switches:
  s_left_flipper:
    number: 1
    tags: left_flipper
  s_right_flipper:
    number: 2
    tags: right_flipper
```

You can pick whatever names you want for your switches. We chose `s_left_flipper` and `s_right_flipper`.

Note that we configured this switches with numbers `1` and `2`, but you should use the actual switch numbers for your control system that the flipper buttons are connected to. (See *How to configure "number:" settings* for instructions for each type of control system.)

We also added tags called `left_flipper` and `right_flipper`. These are optional, but recommended. The reason is that MPF includes a *combo switch* feature which posts events when player switches are held in combination. If you add these tags to your flipper switches, an event called *flipper_cancel* will be posted when the player hits both flipper buttons at the same time which you can use to cancel shows and other things you want the player to be able to skip.

2. Add your flipper coils

Next you need to add entries for your flipper coils to your machine-wide config. These will be added to a section called `coils:`.

```
coils:
  c_flipper_left:
    number: 0
    allow_enable: true
    default_hold_power: 0.125
  c_flipper_right:
    number: 1
    allow_enable: true
    default_hold_power: 0.125
```

Again, the `number:` entries in your config will vary depending on your actual hardware, and again, you can pick whatever names you want for your coils.

Also note that the coils have `allow_enable: true` entries added. (In MPF config files, values of "yes" and "true" are the same.) The purpose of the `allow_enable: true` setting is that as a safety precaution, MPF does not allow you to enable (that is, to hold a coil in its "on" position) unless you specifically add `allow_enable: true` to that coil's config.

Since flippers need to be held on (as long as the flipper button is active), you need `allow_enable: true` in the coil config for them.

Finally, notice that there's a `default_hold_power: 0.125` setting for each coil. That is the power value (from 0-1) which controls how much power is applied to the flipper when it's held on. A value of 0.125 is 12.5% power, a value of 2 is 25% which, a value of 0.375 is 37.5%, 0.5 is 50%, etc.

We just start with the lowest setting for now and you can increase it later if it's not enough.

3. Add your flipper entries

At this point you have your coils and switches defined, but you can't flip yet because you don't have any flippers defined. Now you might be thinking, "Wait, but didn't I just configure the coils and switches?" Yes, you did, but now you have to tell MPF that you want to create a flipper mechanism which links together the switch and the coils to become a "flipper".

You create your flipper mechanisms by adding a `flippers:` section to your machine config, and then specifying the switch and coils for each flipper that you defined in Steps 1 and 2.

Here's what you would create based on the switches and coils we've defined so far:

```
flippers:
  left_flipper:
    main_coil: c_flipper_left
    activation_switch: s_left_flipper
```

(continues on next page)

(continued from previous page)

```
  right_flipper:
    main_coil: c_flipper_right
    activation_switch: s_right_flipper
```

4. Enabling your flippers

By default, MPF only enables flippers when a game is in progress. So if this is a first-time config and you haven't configured your ball devices and start button and everything, you can't actually start a game yet, which means you can't test your flippers.

Fortunately we can get around that by configuring your flippers to just automatically enable themselves when MPF starts. To do this, add the following entry to each of your flippers in your config file:

```
enable_events: machine_reset_phase_3
```

So now the `flippers:` section of your config file should look like this:

```
flippers:
  left_flipper:
    main_coil: c_flipper_left
    activation_switch: s_left_flipper
    enable_events: machine_reset_phase_3
  right_flipper:
    main_coil: c_flipper_right
    activation_switch: s_right_flipper
    enable_events: machine_reset_phase_3
```

5. Configure your control system hardware

At this point your flipper configuration is technically complete, though there are two other important things you may have to do first:

If you're using physical hardware, you may need an additional section in your machine config for your control system. (For example, FAST Pinball and Open Pinball Project controllers require a one-time port configuration, etc.) See the *control system documentation* for details.

6. Adjust your flipper power

As a safety precaution, MPF uses very low (10ms) default pulse times for coils. In most cases, 10ms will not be enough power to physically move the flippers when you hit the button. (You might hear them click or buzz without actually seeing them move.)

So check out the documentation in the coils section for instructions on how to adjust the *pulse power* and the *hold power* for the coils you're using for your flippers.

Here's the complete config

Here's the complete machine config file (or sections of the machine config file) we created in this How To guide:

```
#config_version=5
switches:
  s_left_flipper:
    number: 1
    tags: left_flipper
  s_right_flipper:
    number: 2
    tags: right_flipper
coils:
  c_flipper_left:
    number: 0
    allow_enable: true
    default_hold_power: 0.125
  c_flipper_right:
    number: 1
    allow_enable: true
    default_hold_power: 0.125
flippers:
  left_flipper:
    main_coil: c_flipper_left
    activation_switch: s_left_flipper
    enable_events: machine_reset_phase_3
  right_flipper:
    main_coil: c_flipper_right
    activation_switch: s_right_flipper
    enable_events: machine_reset_phase_3
```

How to temporarily disable flippers

Help us to write it

```
switches:
  s_left_flipper:
    number: 1
    tags: left_flipper
  s_right_flipper:
    number: 2
    tags: right_flipper
coils:
  c_flipper_left:
    number: 0
    allow_enable: true
    default_hold_power: 0.125
  c_flipper_right:
    number: 1
    allow_enable: true
    default_hold_power: 0.125
flippers:
```

(continues on next page)

(continued from previous page)

```
  flipper_left:
    main_coil: c_flipper_left
    activation_switch: s_left_flipper
    hold_coil:
    enable_events: ball_started, flipper_on
    disable_events: ball_will_end, flipper_off
  flipper_right:
    main_coil: c_flipper_right
    activation_switch: s_right_flipper
    hold_coil:
    enable_events: ball_started, flipper_on
    disable_events: ball_will_end, flipper_off
##! mode: flipper_mode
mode:
  priority: 1000
event_player:
  mode_flipper_mode_started: flippers_on
  timer_flippers_disabled_started: flippers_off
  timer_flippers_disabled_complete: flippers_on
timers:
  flippers_button_active_left:
    control_events:
      - event: s_flipper_left_active
        action: restart
      - event: s_flipper_left_inactive
        action: stop
    start_value: 0
    end_value: 10
    direction: up
    tick_interval: 1s
  flippers_button_active_right:
    control_events:
      - event: s_flipper_right_active
        action: restart
      - event: s_flipper_right_inactive
        action: stop
    start_value: 0
    end_value: 10
    direction: up
    tick_interval: 1s
  flippers_disabled:
    control_events:
      - event: timer_flippers_button_active_left_complete
        action: start
      - event: timer_flippers_button_active_right_complete
        action: start
      - event: timer_flippers_disabled_complete
        action: reset
    start_value: 0
    end_value: 3
    direction: up
    tick_interval: 1s
```

How to enable "secondary playfield" flippers

Related Config File Sections
flippers:

Secondary or upper flippers (e.g. on an upper playfield) are enabled and defined just like normal lower flipper.

This is an example:

```
flippers:
  lower_left:
    main_coil: c_flipper_lower_left_main
    activation_switch: s_flipper_left
    label: Left Main Flipper
  lower_right:
    main_coil: c_flipper_lower_right_main
    activation_switch: s_flipper_right
    label: Right Main Flipper
  upper_left:
    main_coil: flipperUpLMain
    activation_switch: flipperUpL
    enable_events: ball_started, enable_upper_flippers
    disable_events: ball_will_end, service_mode_entered, disable_upper_flippers
    label: Upper Left Flipper
  upper_right:
    main_coil: flipperUpRMain
    activation_switch: flipperUpR
    enable_events: ball_started, enable_upper_flippers
    disable_events: ball_will_end, service_mode_entered, disable_upper_flippers
    label: Upper Right Flipper
```

Additionally, we defined `disable_upper_flippers` as event to disable the upper flippers and `enable_upper_flippers` to re-enable them. This might be useful if you want to disable flippers in some mode. If you do not want them to be enabled by default remove `ball_started` from `enable_events`.

How to enable "weak flippers" (novelty mode)

Some machines have modes which reduce the flipper power. This can be implemented in two ways. Either by reducing `pulse_power` or by reducing `pulse_ms` (some platforms only support the latter).

This is an example:

```
switches:
  s_left_flipper:
    number: 1
  s_right_flipper:
    number: 2
coils:
  c_flipper_left:
    number: 0
    default_pulse_ms: 30
    default_pulse_power: 0.5
```

(continues on next page)

(continued from previous page)

```
    default_hold_power: 0.125
  c_flipper_right:
    number: 1
    default_pulse_ms: 30
    default_pulse_power: 0.5
    default_hold_power: 0.125
flippers:
  left:
    main_coil: c_flipper_left
    activation_switch: s_left_flipper
    enable_events: normal_flippers_enable, ball_started
    disable_events: weak_flipper_enable, ball_will_end, service_mode_entered
  right:
    main_coil: c_flipper_right
    activation_switch: s_right_flipper
    enable_events: normal_flippers_enable, ball_started
    disable_events: weak_flipper_enable, ball_will_end, service_mode_entered
  left_weak:
    main_coil: c_flipper_left
    main_coil_overwrite:
      pulse_power: 0.3     # alternatively you can use pulse_ms: 20 here
    activation_switch: s_left_flipper
    enable_events: weak_flipper_enable
    disable_events: normal_flippers_enable
  right_weak:
    main_coil: c_flipper_right
    main_coil_overwrite:
      pulse_power: 0.3     # alternatively you can use pulse_ms: 20 here
    activation_switch: s_right_flipper
    enable_events: weak_flipper_enable
    disable_events: normal_flippers_enable
```

We define two sets of flippers: Normal and weak flippers. Post `weak_flipper_enable` in your mode to enable weak flippers. Later post `normal_flippers_enable` to reenable normal flippers. You can also use your mode start/stop events here.

Related Events

None

Flipper end-of-stroke (EOS) switches

Here's the thing about EOS switches in a modern pinball machine: they're optional.

To be very clear, EOS switches are only optional if the software is written to not use them. You can't just walk up to an existing game and cut the wires to the EOS switches or you'll probably burn up your coils. (I say "probably" because some games will detect that the EOS switch wasn't hit when it should have been and cut the power anyway.)

If you wanted to program a game without EOS switches, you could do that. The way you'd do that is to flip from "power" to "hold" mode after a predefined time (like 30ms), rather than waiting for an EOS switch to be engaged.

Why would you want to use the "pulse timing" method versus the "EOS switch" method for the flipper power stroke? There are a few reasons:

- You can control the "strength" of the flippers in software, rather than with hardware. This means you can fine tuning the flipper feel for your game without having to swap coils or adjust voltages.
- You can allow operators to change flipper strength via a service menu item, compensating for mismatched coils, coil age, machine slope, etc. However, this could also be implemented using PWMed pulses with full stroke and EOS cut-off.
- Your software can change the strength as part of a game feature. (For example, Wizard of Oz has a "weak flippers" mode which makes the shots harder.) This can also be implemented with EOS by reducing the pulse time below the EOS (tpyical) cut-off time.

Having said this, there's still a reason you might want to use the EOS switches today—-the EOS switch can be used to detect if a fast-moving ball has hit the flipper so hard that it broke through the hold power and caused the flipper bat to fall down. The idea is you'd use the EOS switch to reactivate the power winding (or to reapply full power if you're using the pulse method) until the EOS switch is activated again, and then you'd go back to holding the coil.

Whether you actually want to do this is a matter of opinion. Finding the proper strength for your hold power—-especially if you're using the pulse method—-is a balance between applying enough power to keep the flipper bat up without using so much power that your coil overheats. Some argue that if you get this balance right, your hold power should be enough to stand up to a fast ball hitting an upheld flipper. The other thing to consider with this is that even if a fast-moving ball does knock the flipper bat down, there's no agreement on whether automatically re-applying full power to raise the bat is the right thing to do. Some have argued that that's confusing to the player, and that if the flipper bat does fall down when the player is not expecting it, that the player should choose to re-engage it by releasing and reapplying the flipper button.

Even if you don't use EOS switches for action purposes in your game, chances are your flipper mechanisms have them. Assuming you have enough switch inputs available, we like the idea of wiring up your EOS switches anyway and just audit logging whether an EOS switch is deactivated while its associated flipper button is still active. Doing so means you capture the number of times a ball inadvertently moves a flipper bat, and you can make power adjustments to your hold phase accordingly. It also lets the machine know if the flippers are broken.

How does the machine know when the flipper is "up"?

You might notice that both of the options for not burning up the flipper coils when they're held up require that the machine "knows" when the coil is up in order to switch over to hold mode. So how exactly does a machine know this?

Many flippers in pinball machines today have an "end of stroke" (or "EOS") switch for each flipper. This switch was located under the playfield near the flipper coil, and it is physically activated by the flipper mechanism once it has rotated fully into the "up" position. In the old days (like in EM machines), the flipper coils all used the dual winding (i.e. "Option 1" from above) approach, and the EOS switch was a normally-closed switch connected in series with the flipper cabinet button which activated the power winding. So when the flipper button was pressed, both the power and hold windings were activated, and then when the flipper was all the way up it would open the EOS switch, cutting off power to the power winding. The hold winding remains energized until the player releases the flipper button.

When EOS switches are used in modern machines, they're typically connected into into the game like any other switch, so the CPU can process the EOS activation and disable the power winding or start pulsing the power.

Option 2: The flipper has one winding, and the game lowers the power once the flipper is up

The other type of flipper uses a normal coil with just a single winding. When the flipper button is pressed, the machine fires the flipper coil with normal full power. Then once the flipper makes it to the “up” position, the game starts pulsing the power really quickly. (So fast that it doesn’t move the flipper back down, but with enough “spaces” between the pulses that the coil doesn’t burn up.)

Then when the flipper button is released, the power is cut to the altogether. In case you’re wondering why the machine pulses the power, it’s because the pinball machine doesn’t have the ability to actually change the voltage and current that is supplied to the coil. That’s fine, though, because what actually causes a coil to burn is the heat generated from the current flowing through it. So a coil which is pulsed on then off every millisecond would only have a “duty cycle” of 50%, thereby generating far less heat and not burning up. (The 1ms on / 1ms off is just an example for this illustration. In a real machine it might be 1 on / 10 off, or 2/18, or 1/6—the exact pulse ratio depends on the coil type and the amount of voltage used.)

This single-winding coil is less common. Stern used to do it though in their current SPIKE system they’ve moved back to dual-wound flippers.

Design Decision 2: Pulse timings or EOS switch to indicate “up” position?

Next you have to figure out how you’re machine will know when to switch to the low power hold mode. (How it switches depends on Design Decision 1, where it either cuts off the high power winding, or switches over from the solid pulse to the quick on/off modulated pulses.) If you use pulse timings then it switches over after a certain number of milliseconds. If you use the EOS switch then it activates full power until the EOS switch is activated. Our view is that using the EOS switch to switch over to low-power hold mode is far less flexible than configuring specific initial pulse times. We like that this allows game designers and operators to precisely configure flipper power, and certainly this is a much more modern approach than physically swapping out flipper coils to increase or decrease power. Then again, if you’re old school and want to fire that flipper with full power until that EOS switch is activated, fine, go for it.

Design Decision 3: Will you use EOS switches to notify the game that a ball has “broken through” the hold?

Modern machines use pulse-width modulation (PWM) to keep flipper bats up because most coils will instantly burst into flames if you enable them at 48V which are typically used in today’s machines. PWM uses a so called duty-cycle which determines how much energy moves into the flippers. More energy strictly results in more power but that energy also turns into heat. Unfortunately, the resistance in copper wires in the coil increases with the temperature and, consequently, the less current and energy will flow through the coil. As a result the coil will become weaker of time when it heats up. Since we do not know the temperature in software this cannot easily be compensated as runtime (and the coil would probably become even hotter and burn if we would try).

Finding the right spot where the coil is strong enough, knockdowns do not happen and the temperature stays low enough is not generally easy. Parts age over time, environment temperature differs between location and even the voltage might fluctuate. We have seen overheating of coils in some machines by newer manufacturers.

So what can we do about this? We can detect when the EOS switch opens while the flipper button is active and repulse the flipper coil. Ideally, this should happen inside the pinball hardware but this is not supported by all hardware platform in MPF. For all remaining platforms, we mitigate this in software in MPF. This introduces a few milliseconds of delay but it should be fast enough that the player does not notice it.

This is how you can enable it in MPF:

```
flippers:
  single_wound_flipper:
    main_coil: c_flipper_single_main
    activation_switch: s_flipper_single
    eos_switch: s_flipper_single_eos
    use_eos: true
    repulse_on_eos_open: true
    eos_active_ms_before_repulse: 500
  dual_wound_flipper:
    main_coil: c_flipper_dual_wound_main
    hold_coil: c_flipper_dual_wound_hold
    activation_switch: s_flipper_dual_wound
    eos_switch: s_flipper_dual_wound_eos
    use_eos: true
    repulse_on_eos_open: true
    eos_active_ms_before_repulse: 500
```

To prevent repeated activations MPF will wait `eos_active_ms_before_repulse` ms before a repulse can happen. There are certain races between hardware rules and this mechanism which MPF tries to handle (but we might have missed cases - let us know if you find any rough edges or weird behaviour with this).

In general, this should allow you to reduce PWM power by a lot and instead use repulses in the rare case of knockdowns. This should work with all platforms and will use hardware rules if your platform supports them.

Kickbacks

Related Config File Sections
kickbacks:
ball_saves:

- *Monitorable Properties*
- *Related Events*

A kickback mechanism is a type of *autofire coil* that kicks the ball back into play, typically located in an outlane. It is often paired with a *ball_save* to compensate for missed kickbacks.

TODO: Add a picture of a kickback

This is an example:

```
switches:
  s_kickback:
    number: 5
coils:
  c_kickback:
    number: 7
    default_pulse_ms: 15
kickbacks:
  ac_kickback:
    coil: c_kickback
    switch: s_kickback
ball_saves:
  kickback_ball_save:
    active_time: 5s
    enable_events: kickback_ac_kickback_fired
    auto_launch: true
    balls_to_save: 1
```

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for kickbacks is `device.kickbacks.<name>`.

enabled Boolean (true/false) which shows whether this kickback is enabled.

Related Events

- *kickback_(name)_fired*

Lights

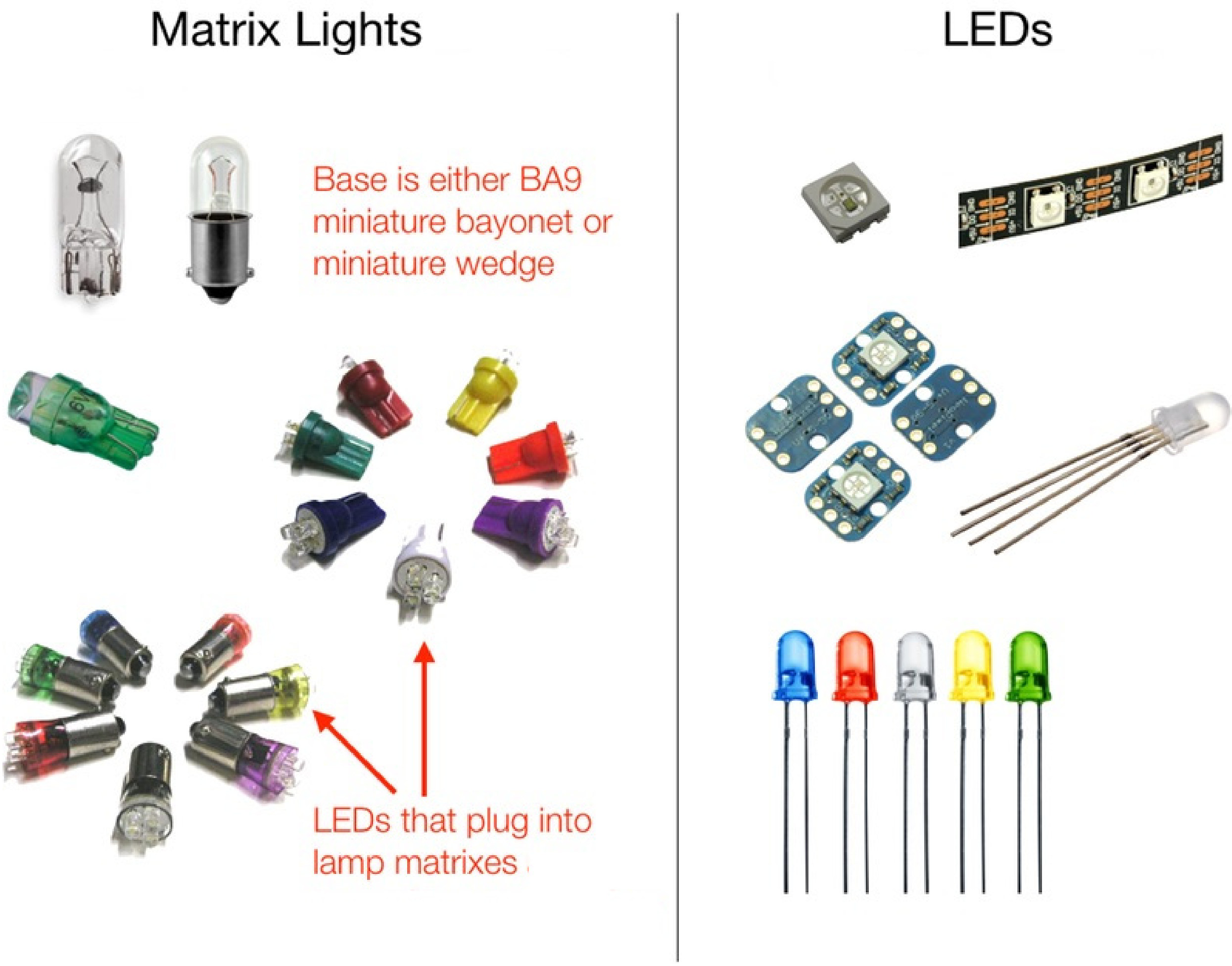
Related Config File Sections
lights:
light_settings:
light_player:

- *Fully working Example 1 - basics*
- *Fully working Example 2 - light_stripes*
- *Monitorable Properties*
- *Related How To guides*
- *Related Events*

In MPF 0.50 all LEDs, matrix lights and GIs are configured as *lights:*. See *"Lights" versus "LEDs" (Some LEDs are lights?!?)* for details.

There are multiple types of lights (read those for specific details):

- *LEDs*
- *GI (general illumination)*
- *Matrix Lights (Bulbs)*
- *Flashers*
- *Coils as Lights*



This is an example of for a light:

```
lights:
  my_led:
    number: 7   # the exact number format depends on your platform
```

For WS2812 LEDs use `type: grb` (WS2811 does not need this):

```
lights:
  my_ws2812_led:
    number: 23  # the exact number format depends on your platform
    type: grb
```

You can also map individual color channels:

```
lights:
  rgb_led:
    type: rgb
    channels:
      red:
        number: 9-29     # the exact number format depends on your platform
      green:
        number: 9-30
      blue:
        number: 9-31
      white:
        number: 9-32
```

Starting with MPF 0.54 there is a new syntax to chain lights:

```
lights:
  led_0:
    start_channel: 0-0    # the exact number format depends on your platform
    subtype: led
    type: rgb    # will use red: 0-0, green: 0-1, blue: 0-2
  led_1:
    previous: led_0
    subtype: led
    type: rgbw   # will use red: 0-3, green: 0-4, blue: 0-5, white: 0-6
  led_2:
    previous: led_1
    subtype: led
    type: rgbw   # will use red: 0-7, green: 0-8, blue: 0-9, white: 0-10
```

If your light is connected to a driver use this example:

```
coils:
  light_connected_to_a_driver:
    number: 42          # number depends on your platform
    allow_enable: true  # this will allow 100% enable without pwm
lights:
  light_on_a_driver:
    number: light_connected_to_a_driver  # map this light to a driver
    platform: drivers
```

Fully working Example 1 - basics

Let's bring above informaton together and learn by example. Though the following example is a fully working minimal set for the Cobra controller, it is as well helpful to understand the concpet more if you use a different set of hardware. For this example to work physically, the Cobra board needs to have 5V power supply and a Neopixel strip connected to NEO0. No need for a high voltage power supply like you need for coils. The example has been built for a WS2811 strip, but can be used as well for a WS2812 strips and others. This `config.yaml` is the only configuration file you need in your project. The config file is fully valid for the Cobra board connected to a Linux PC running MPF. If you have a Cobra board but run Windows or macOS you have to change the `ports`. If you run a completely different hardware you have to adapt the `hardware` section.

```
#config_version=5

hardware:               # change in case you don't use OPP
   platform: opp
   driverboards: gen2

opp:
   ports: /dev/ttyACM0, /dev/ttyACM1 # change if your Cobra board uses different ports

lights:
   led_strip_0_led_1:
      number: 0-0-1 # the exact number format depends on your platform
      subtype: led
      type: rgb
      tags: group1
   led_strip_0_led_2:
      number: 0-0-2 # the exact number format depends on your platform
      subtype: led
      type: rgb
      tags: group1
   led_strip_0_led_3:
      previous: led_strip_0_led_2
      subtype: led
      type: rgb
      tags: group1

light_player:
   led_code:
      led_strip_0_led_1: DFFF00
   led_off:
      led_strip_0_led_1: off
      led_strip_0_led_2: off
      led_strip_0_led_3: off
   led_name:
      led_strip_0_led_1: LightSalmon
   group_light:
      group1: green
   led_fade:
      led_strip_0_led_1:
        color: slateblue
        fade: 2000ms

keyboard:
   1:
     event: led_off
   2:
     event: led_code
   3:
     event: led_name
   4:
     event: group_light
   5:
     event: led_fade
```

When you run this configuration, you can use the keys 1 - 5 to set certain lights. Each key submits one event. That event is being used in the `light_player` section. Each event in the `light_player` section is assigned to an LED (or group of LEDs), these LEDs are defined in the lights section of the config file. In the `light_player` section each LED is assigned a color. For the colors you can use:

- `off`: to switch an led off
- hex code: e.g. `DFFF00` to define each color channel, here red=DF, green=FF, blue=00. Note that the code has no leading # since that would be a comment in your config file
- html name: e.g. `LightSalmon`, you can check the available names here https://htmlcolorcodes.com/color-names/ Capitalization doesn't matter in the config file, e.g. `LightSalmon` or `lightsalmon` are equally good

In the `light_player` section you can either define the color as value of the specified LED, which turns that LED immediately to the given color. Or you can specifiy a `color` and a `fade` value, then the color will transition to the new value in the specified time. In the config file this is configured for key 5, see `led_fade` in the `light_player` section.

In the `light_player` section after each event you can specify one or multiple lights. In the section `led_off` both LEDs are specified, hence both are turned off when the event `led_off` is sent.

Note that the defined lights have tags, here the tag value is `group1`. In the `light_player` section you can either address a single LED by its name or you can use a group name to address all LEDs in that group. When you press key 4 then LED 1, 2, and 3 are switched to green. A few notes on above example:

- It is kept as simple as possible to learn by example.
- Keep in mind that numbering starts with 0, so LED 1 and 2 in above config are your 2nd and 3rd LED of the strip
- If you use a WS2812 strip then the green and red channel are swapped. Which means that if you see a red light when pressing button 4, then you have a WS2812 strip. In order to get this fixed change the type value in the config file from `rgb` to `grb`.
- After you run that example and understand how it works, then change the type of `led_strip_0_led_2` to ggg. Now run the setup again and press key 4. The first LED will still show green, but the second LED will show white. That is because you told the configuration that that LED has only green channels so it turns all of them on when you want to show green, but in fact the other channels show red and blue. Depending on what you do, this might be helpful to know.
- The above example uses NEO0 of the Cobra controller, if you want to use NEO1 you have to change the number value in the lights section of your config file, the first 0 has to be a 1 in this case.
- Note that in the definition of `led_strip_0_led_3` the hardware addess is not specified (unlike led_strip_0_1 and led_strip_0_2. Instead what is specified is what the previous LED is. That is handy in case you need to add a new LED somewhere in your chain. Instead of changing all hardware addresses you can just change the one `previous` tag.

Fully working Example 2 - light_stripes

From a hardware perspective the same remarks as in the example above are true. This example will show a fully working example using the parameter `light_stripes` (yes written with an e). The adavantage of this paramater is that you are able to define a full serial LED light strip with a few lines of config. See as well the corresponding config file section *light_stripes:*

```
#config_version=5

hardware:
 platform: opp
 driverboards: gen2

opp:
 ports: /dev/ttyACM0, /dev/ttyACM1

light_stripes: #yes there is a spelling mistake, make the same mistake
 led_strip_0:
    number_start: 0
    count: 50
    number_template: 0-0-{}
    light_template:
       type: rgb
       tags: strip0

light_player:
 full_strip_on:
    strip0: DFFF00
 led_off:
    led_strip_0: off

keyboard:
 1:
    event: led_off
 2:
    event: full_strip_on
```

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for lights is `device.lights.<name>`.

brightness The numeric value of the brightness of this light, from 0-255.

color The current color.

Related How To guides

- *Tutorial step 17: Add lights (or LEDs)*

Related Events

None

LEDs

Related Config File Sections
lights:
light_settings:
light_stripes:
light_rings:
light_player:

MPF can control LEDs, including single-channel (single color) and full RGB LEDs. (You can control the order too, so you can control RGB, BRG, etc.) You can set default fade rates and control strips and rings of LEDs.

In general there are two ways to wire LEDs in a pinball machine. Either parallel or serial. With serial LEDs you got a chain of LEDs which are connected to a controller board on one side. In contrast, with parallel LEDs every LED has its own wire(s) to the controller. While parallel LEDs are more robust in general they also require much more wiring. Which kind of LEDs you're using usually depends on what is supported in your platform (some support both).

Serial LEDs

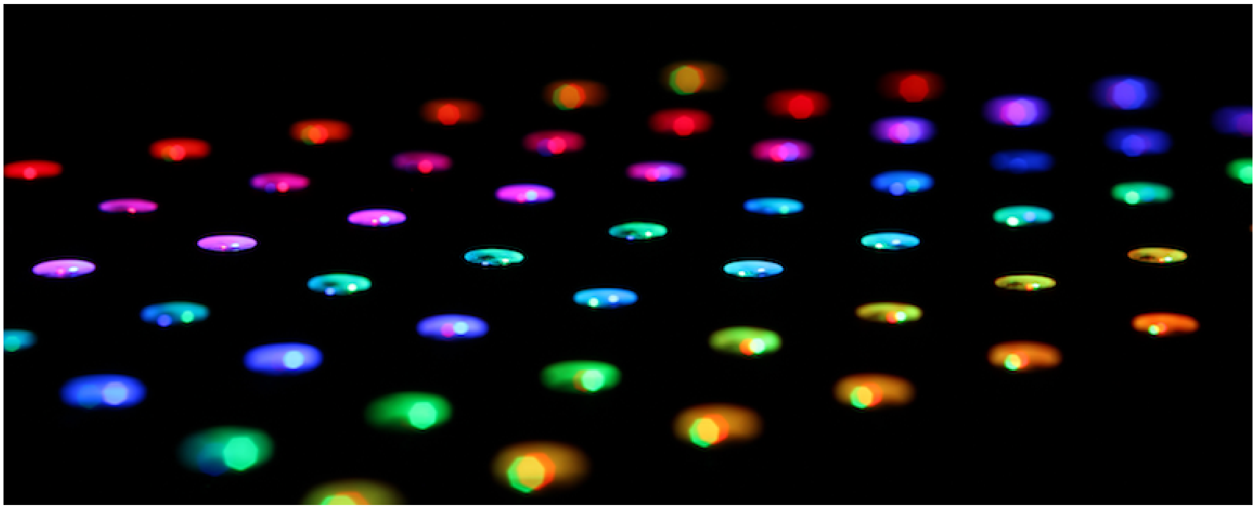
With serial LEDs the order of colors is usually fixed. For instance, in WS2811 LEDs (a common serial LED controller embedded inside the LED), the first channel is red, the second green and third is blue (RGB order). Newer WS2812 LEDs have GRB order (green, red and blue). Some LEDs also contain an additional white channel and thereby have four channels (either RGBW or GRBW order). Other serial LEDs contain three white LEDs (WWW order). If nothing is specified MPF assumes RGB order so you need to specify it for any LED with a different channel order.

Overview video about serial LEDs:

https://youtu.be/Q9BG9T7Kj4A

Hardware

There are two common types of serial LEDs: WS281x and LPD880x. (See *WS2811 and WS2812 LEDs in Pinball* for more details about WS2811/WS2812 in pinball.) Those LEDs are chained which means that the controller only connects to the first LED. The first LED will connect to the second. The second to the third and so on.



Both types are spec'd for 4.5V to 5.5V operations and you should make sure that the voltage does not drop below 4.5V inside the chain at full brightness. Otherwise, your colors will be off and the LEDs might start to flicker. We recommend you to turn on all your LEDs and measure this. In most cases it is helpful to run your *power supply* at 5.5V instead of 5V to give your LEDs some headroom.

Additionally, make sure to run separate ground lines for serial LEDs from your PSU. We recommend you to connect the ground at the PSU and not below the playfield because coils will create a lot of spike in the ground line otherwise. However, make sure that you connect your grounds or you will be in danger!

Video about wiring of lights:

https://youtu.be/C9GzkMduEKY

Config in MPF

You can define serial LEDS in MPF as *lights:*:

```
lights:
  my_ws2811:
    number: 0         # first LED in chain (with three channels)
    type: rgb
  my_ws2812:
    number: 1         # second LED in chain (with three channels)
    type: grb
  my_serial_white_leds:
    number: 2         # third LED in chain (with three channels)
    type: www
```

The numbering depends on your platform. Internally the first LED will map to the first three LEDs in the chain (because one LED contains three interal LEDs). The second will map to LED four to six and so on.

The config above is equivalent to the following (again numbers may be different per platform):

```
lights:
  my_ws2811:
    channels:
      red:
        - number: 0-0
      green:
        - number: 0-1
      blue:
        - number: 0-2
  my_ws2812:
    channels:
      red:
        - number: 1-1
      green:
        - number: 1-0
      blue:
        - number: 1-2
```

RGBW LEDs are special in most serial LED controllers since the controller assumes that every LED has exactly three channels. Therefore, you have to assign the channels directly:

```
lights:
  my_rgbw_serial_led:
    channels:
      red:
        - number: 3-0
      green:
        - number: 3-1
      blue:
        - number: 3-2
      white:
        - number: 4-0
  my_ws2812_after_rgbw:
    channels:
      red:
        - number: 4-1
      green:
        - number: 4-2
      blue:
        - number: 5-0
```

The RGBW shifts all the channels by one internally. As you can see this can quickly become confusing so it might be wise to run RGBW LEDs (or any non-three-channel LEDs) as a separate chain.

Starting with MPF 0.54 there is a new syntax to chain lights:

```
lights:
  led_0:
    start_channel: 0-0    # the exact number format depends on your platform
    subtype: led
    type: rgb    # will use red: 0-0, green: 0-1, blue: 0-2
  led_1:
```

(continues on next page)

(continued from previous page)

```
    previous: led_0
    subtype: led
    type: rgbw   # will use red: 0-3, green: 0-4, blue: 0-5, white: 0-6
  led_2:
    previous: led_1
    subtype: led
    type: rgbw   # will use red: 0-7, green: 0-8, blue: 0-9, white: 0-10
```

Parallel LEDs

TODO: Add a picture of a parallel RGB LED

With parallel LEDs you usually got a bit more flexibility with your channel assignments. You can decide to make an LED with only a red channel for example. MPF cannot guess your hardware layout in most platforms. Therefore your have to explicitly tell MPF your channel layout:

```
lights:
  my_red_only_insert:
    channels:
      red:
        - number: 0
  my_rgb_insert:
    channels:
      red:
        - number: 1
      green:
        - number: 3
      blue:
        - number: 2
  my_white_light:
    channels:
      white:
        - number: 4
```

You can also have multiple channels per color (if you do not want to make them different lights):

```
lights:
  multi_white_channels:
    channels:
      white:
        - number: 5
        - number: 6
        - number: 7
```

With parallel LED you can also use `start_channel` to define the color (starting from MPF 0.54):

```
lights:
  my_red_only_insert:
    start_channel: 0    # the exact number format depends on your platform
    type: r    # will use red: 0
  my_rgb_insert:
    start_channel: 1    # the exact number format depends on your platform
```

(continues on next page)

(continued from previous page)

```
    type: rbg   # will use red: 1, green: 3, blue: 2
  my_white_light:
    previous: my_rgb_insert     # you can also chain those if you want
    type: w   # will use white: 4
```

Serial vs Parallel LEDs

There is a controversy if serial LEDs are feasible below the playfield or not. In general, serial LEDs require much less wiring which make them much cheaper during assembly. However, if one LED fails within a chain all subsequent ones will likely also fail until the broken LED is changed. Interference will happen below a playfield and might disturb the colors. In practise the refresh rate of serial LEDs are so high that you will not notice any incorrect colors. It is unclear how interference affects the reliability of the controller chips of serial LEDs. Jersey Jack Pinball tried serial LEDs in the Wizard of Oz (WoZ) and ran into a lot of reliability issues. Finally, they reverted back to parallel LEDs (one I2C driver chip per chain). One of the problems they had was interferences in the ground line which is why we recommend a separate power supply for serial LEDs and a separate ground line (but still common ground; see *the voltages and power guide* for details).

For production runs you should probably be careful with serial LEDs. At least test extensively. However, you might take some risks in a homebrew machine because serial LEDs are quite cheap and easy to replace once broken. In practise they seem to work just fine for all homebrew machines we know.

Can I used RGB LEDs below colored inserts?

There is no point to use RGB LEDs below colored inserts. That simply does not work physically. Those colored inserts act as filter and any other color simply shall not pass.

We recommend white LEDs below colored inserts. Then define them as red or whatever color your insert is. If you use parallel LEDs below colored inserts just buy plain white ones. For serial LEDs you can buy bulk WS2811 PCBs from china and connect white LEDs to any of the channels.

Which LED Types Are Supported in MPF?

MPF supports any white, single-color or multi-color LED. This includes RGB, RGBW or any other combination you can imagine. The `type` parameter just reads the channels and maps them without thinking too much of it. For instance you can use GRBW LEDs with a green, red, blue and white channel. Similarly, RRBRGWBGWWR or even more crazy combinations work fine.

Currently, MPF support red, blue, gree and white channels. White it calculated as the minimum brightness of all channels. If you need other channels such as orange let us know in the forum.

Color Correction

If you are using RGB LEDs, they might not be perfectly white when you turn them on. They might be pinkish or blueish instead depending on the brand of the LED. To a certain extend this is normal/expected and you can compensate for it by configuring *color_correction profiles in light_settings*.

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for LEDs is `device.lights.<name>`.

- *color*
- *corrected_color*

Related How To Guides
Tutorial step 17: Add lights (or LEDs)

Related Events
None

"Lights" versus "LEDs" (Some LEDs are lights?!?)

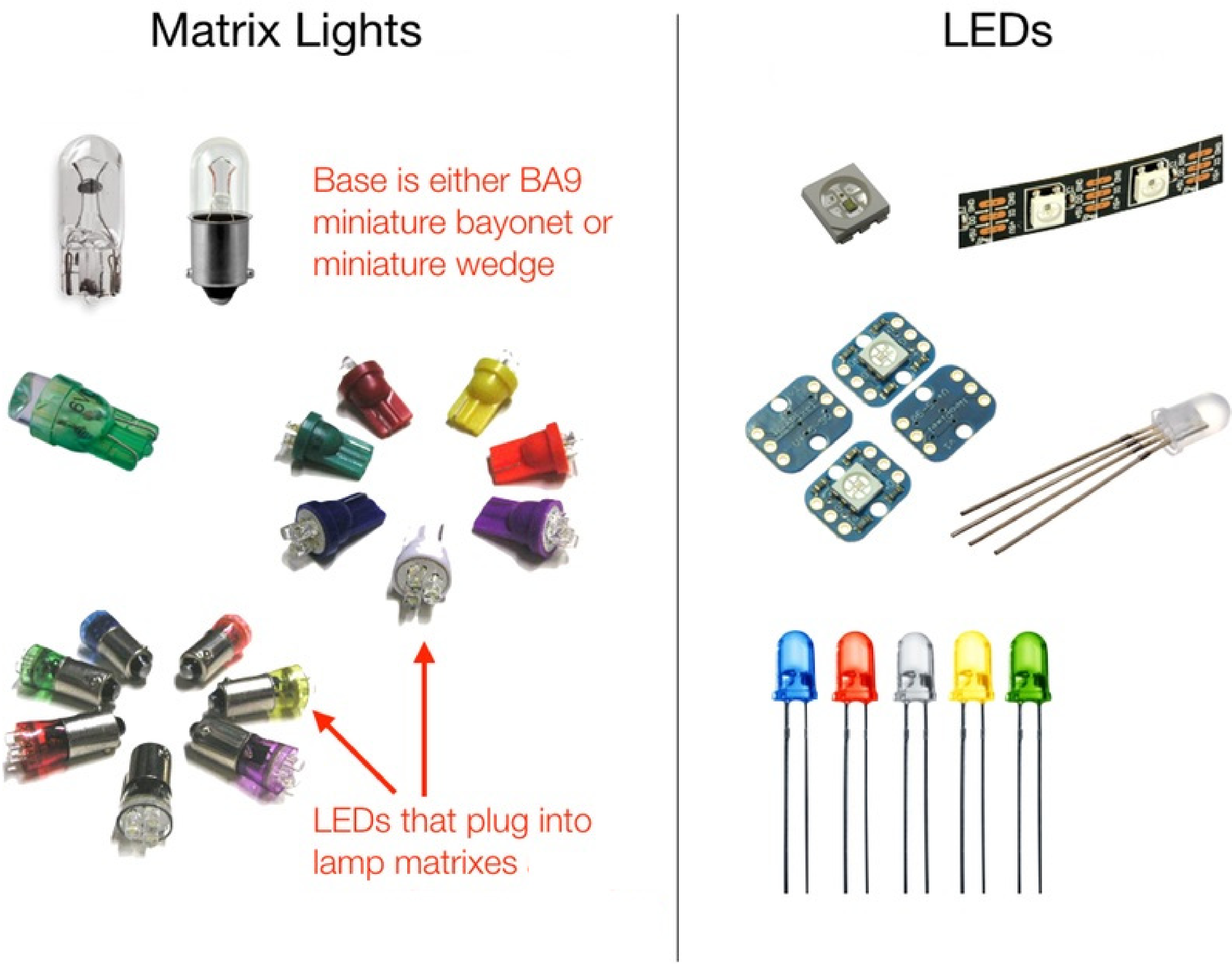
In MPF 0.33 and earlier not all LEDs had to be configured as LEDs. This changed in 0.50+ where all lights, GIs and matrix_lights were unified as lights. The distinction is now only the *subtype* in the *lights config*.

Taking a step back. There are two types of lighting systems in pinball machines: lamp matrices and direct-connected LEDs. All commercial pinball machines from about 1979 through 2012 (give or take) used lamp matrices (typically with 8 rows and 8 columns of lights). Historically these were used with incandescent light bulbs, (#44, #555, etc.).

However, in more recent years various manufacturers have released LED "replacement" bulbs that fit the old-style sockets but that are actually LEDs. If your machine uses a lamp matrix, then you will add your lights (whether they're LEDs or incandescent) as *lights* with subtype matrix in your machine config. You'll do this even if you have LED bulbs in your lamp matrix.

Alternately, if you have directly-controlled LEDs (i.e. no lamp matrix), whether single color or RGB, then you'll configure them as *subtype* led in the *lights:* section of your config.

The following diagram shows the different types. An easy way to tell is if your lights or LEDs have mini bayonet or mini wedge bases, they're *Matrix Lights*, and everything else is *LEDs*:



Note that it's possible that you'll have both matrix lights and direct connected LEDs in the same machine. For example, maybe you're writing code for an existing WPC machine and you'll use the existing matrix lights as they are while also adding new direct connected LEDs for some new toys.

GI (general illumination)

Related Config File Sections
lights:
light_player:
coils:

- *Hardware*
- *Config*
- *Monitorable Properties*
- *Related How To guides*
- *Related Events*

MPF includes support for GI (general illumination) light strings which are common in existing Williams and Stern machines. You can specify GI strings which you can then enable, disable, or (if the hardware supports it) dim. Typically, there are one to four GI strings.

Note: In MPF 0.50 GIs became *lights:* with `subtype` `gi`. They behave like any other lights in MPF.

Hardware

TODO: Add a picture of a GI string TODO: Add a picture of GI LEDs

GI Strings are actually kind of complex. Many of them are AC (even in WPC machines), and some Williams WPC machines include triacs (kind of like a transistor for AC) and "zero cross" AC waveform detection circuits so they can sync their dimming commands with the AC current wave. Later Williams WPC machines split their GI into non-dimmable (which used still used AC) and switched their dimmable to DC. Some machines also have "enable" relays that must be activated first before certain GI strings will work. In general those bulbs are the same models as used for inserts (#44 and #47 for EM and early SS; #444/#555/#249 for later SS; #906 for later machines).

GI string might also be *connected to a driver* and not part of a light matrix. In recent machines LEDs are used but still driven in strings.

Video about wiring of lights:

https://youtu.be/C9GzkMduEKY

Config

MPF hides all this complexity from you. You just define your GI strings in your machine *lights:* section and then you can enable, disable, and dim the dimmable ones as you wish.

This is an example for a *light* with `subtype: gi`:

```
lights:
  gi_string_left:
    number: 3    # number depends on your platform
    subtype: gi
```

In modern machines (such as Spike) your GIs might just be handled as lights. The details depend on your hardware platform and are outlined in the platform documentation.

This is an example for a *light* in Spike:

```
lights:
  gi_string_left:
    number: 3    # number depends on your platform
    subtype: led  # might be matrix in some platforms
```

In some cases GIs are connected to normal drivers on your driver board (e.g. on a PD-16 on the P3-Roc). If that is the case you should configure them as *coils*. Then add them as *light* with `platform: drivers`:

```
coils:
  gi_string_left:
    number: A1-B1-3    # number depends on your platform
    allow_enable: true  # this will allow 100% enable without pwm
lights:
  gi_string_left:
    number: gi_string_left  # map this light to a driver
    platform: drivers
```

Alternatively, you could also use *coil_player* but this gives you the convinience of being able to use GIs in normal light shows.

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for lights is `device.lights.<name>`.

color The color of this string. If you set it to brightness values all color channels will have the same value. Brightness 100 (of 255) will be hex 64 and color 646464.

Related How To guides

See the documentation of your platform on how to configure GIs.

Platform related How To
P/P3-Roc leds
P/P3-Roc matrix light
FAST leds
FAST matrix light
OPP leds
OPP matrix light

Related Events

None

Matrix Lights (Bulbs)

Related Config File Sections
lights:

- *Hardware*
- *Config*
- *Monitorable Properties*

Some hardware platforms support lamps in a matrix. Those lights are usually bulbs and single color (white). Each of them is assigned to a unique column/row combination and driven sequentially by the platform.

Video about wiring of lights:

https://youtu.be/C9GzkMduEKY

Hardware

TODO: Add a picture of bulbs in a matrix

There are various types of bulbs in pinball machines depending on when they were made. In EM and early SS #44 and #47 bulbs were used. Later SS machines used #555 bulbs (or the vibration resistant variant #444 or the #259 substitute). Those bulbs are rated at 6.3V but typically driven at 12V in a matrix or using AC voltages in older machines. Newer machines (pre LED) use #906 bulbs which are rated at 13.5V.

Config

Details differ by platform but the syntax for the number of such a light is usually *column:row* or *column:row* (see your platform for details). The config looks like this:

```
lights:
  my_matrix_light:
    number: 2:10      # or 2/10
```

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for LEDs is `device.lights.<name>`.

- *color*
- *corrected_color*

Related How To Guides
Tutorial step 17: Add lights (or LEDs)

Related Events
None

Flashers

Related Config File Sections
coils:
coil_player:
lights:
light_player:
flasher_player:

MPF includes support for flashers, which are essentially just really bright lights that are controlled via high-power driver transistors instead of low-power lighting circuitry.

MPF's flasher devices are only used in older machines (WPC, Stern SAM, System 11) since modern LED-based machines typically use regular LED devices (or combinations of them) as flashers. (So basically a "flasher" in MPF is any single-color light that's connected to a driver output rather than a light output.

Hardware

#89 and #906 bulbs are commonly used as flashers in pinball machines. Those are rated at 13V but typically driven at higher voltages for only a very short amount of time. Turning them on permanently will burn quickly in most machines.

Video about wiring of lights:

https://youtu.be/C9GzkMduEKY

Config

Starting with MPF 0.50 flashers and lights have been unified. Depending on your platform flashers might be *lights:* or *coils:*. In most cases they are configured as *coil*:

```
coils:
  flasher_coil_4:
    number: 4
    allow_enable: true
```

Then add them as *light*:

```
lights:
  flasher_4:
    number: flasher_coil_4
    platform: drivers
```

Now you can use them in *flasher_player:* (or also in *light_player:* if you want to enable the flasher permanently).

```
flasher_player:
  flash:
    flasher_01: 100ms
```

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for lights is `device.lights.<name>`.

color

The color of this string. If you set it to brightness values all color channels will have the same value. Brightness 100 (of 255) will be hex 64 and color 646464.

Related How To guides

See the documentation of your platform on how to configure GIs.

Platform related How To
P/P3-Roc leds
P/P3-Roc matrix light
FAST leds
FAST matrix light
OPP leds
OPP matrix light

Related Events
None

Coils as Lights

Related Config File Sections
lights:
light_player:
coils:
coil_player:

Sometimes you will find lights on a (coil) driver. There are various reasons for this and MPF supports it. You can either use *coil_player:* to control those lights but it will be different from normal lights (and light shows). Alternatively, you can map the coils to a light (recommended). To map a coil as light you can use the following config:

```
coils:
  your_light_coil:
    number: 42                # number depends on your platform
    allow_enable: true       # this will allow 100% enable without pwm
lights:
  your_light_on_a_coil:
    number: your_light_coil     # map this light to a driver
    platform: drivers
```

This is sometimes done for *GI (general illumination)* and *Flashers*.

WS2811 and WS2812 LEDs in Pinball

The most common serial LEDs use WS2811 or WS2812 controllers. Both controllers differ in the order of the channels. The WS2811 controller uses RGB and WS2812 uses GRB order. Unfortunately, some controller (i.e. Fadecandy) expect WS2812 by default and shuffle pixels internally for you.

Overview video about serial LEDs:

https://youtu.be/Q9BG9T7Kj4A

Config

MPF tries to make this right for you and your can configure those LEDs as follows:

```
lights:
  my_ws2811:
    number: 0         # first LED in chain (with three channels) - exact number format depends on your
↪platform
    type: rgb
  my_ws2812:
    number: 1         # second LED in chain (with three channels)
    type: grb
```

There are also RGBW LEDs which are compatible which usually use RGBW as order. They can be used like this:

```
lights:
  my_rgbw_serial_led:
    channels:
      red:
        - number: 3-0
      green:
        - number: 3-1
      blue:
        - number: 3-2
      white:
        - number: 4-0
```

Starting with MPF 0.54 there is a new syntax to chain LEDs:

```
lights:
  led_0:
    start_channel: 0-0    # the exact number format depends on your platform
    subtype: led
    type: rgb    # will use red: 0-0, green: 0-1, blue: 0-2
  led_1:
    previous: led_0
    subtype: led
    type: rgbw   # will use red: 0-3, green: 0-4, blue: 0-5, white: 0-6
  led_2:
    previous: led_1
    subtype: led
    type: rgbw   # will use red: 0-7, green: 0-8, blue: 0-9, white: 0-10
```

Hardware

Each pixel connects to the next pixel with three wires: VDD, signal, GND. According to the datasheet VDD should be 4.5V to 5.5V. This means two things: First, you can safely dial to power supply up to 5.5V for serial LEDs and we recommend you to do so. Secondly, you should make sure that at maximum brighness (i.e. maximum power consumption) the voltage should not drop below 4.5V. We urge you to actually test this by dialing the brightness up the maximum and measuring the voltage at the middle and the end of your chain.

What if the voltage drops below 4.5V? From our experience serial WS2811/WS2812 work down to 3.5V but the whitepoint shifts to more blueish and the chain becomes much more susceptible to noise and flickering. If you see the voltage below 4.5V we strongly suggest that you fix your power setup.

First step should be to connect power from both sides to your chains. That does not harm and practically halfs the length or your chain. Afterwards, measure the voltage in the middle of the chain. If that still does not help try using thicker wires or dialing up your power supply (especially if the voltage is also dropping at the beginning of your chain).

If all this does not help try shorter chains. Most controllers support multiple chains and you should take advantage of that.

Do not underestimate the currents which are needed to drive LEDs. As a rule of thumb you can calculate 60mA times the number of LEDs. If you got 100 LEDs that make it 6A. 300 LEDs result in 18A which at 5V are 90W. Size your power supply accordingly. Also remember that the voltage drop in your wires is resistance times current `R * I` so size the wire between your PSU and your lights accordingly. You can check this using the Voltage Drop Calculator. Also note that standard .156 molex connectors are only rated for 7A and you do not want your board look like WPC boards with burned connectors.

Video about electronics basics:

https://youtu.be/8ByqYkYKnFc

Video about wiring of lights:

https://youtu.be/C9GzkMduEKY

Types of LEDs

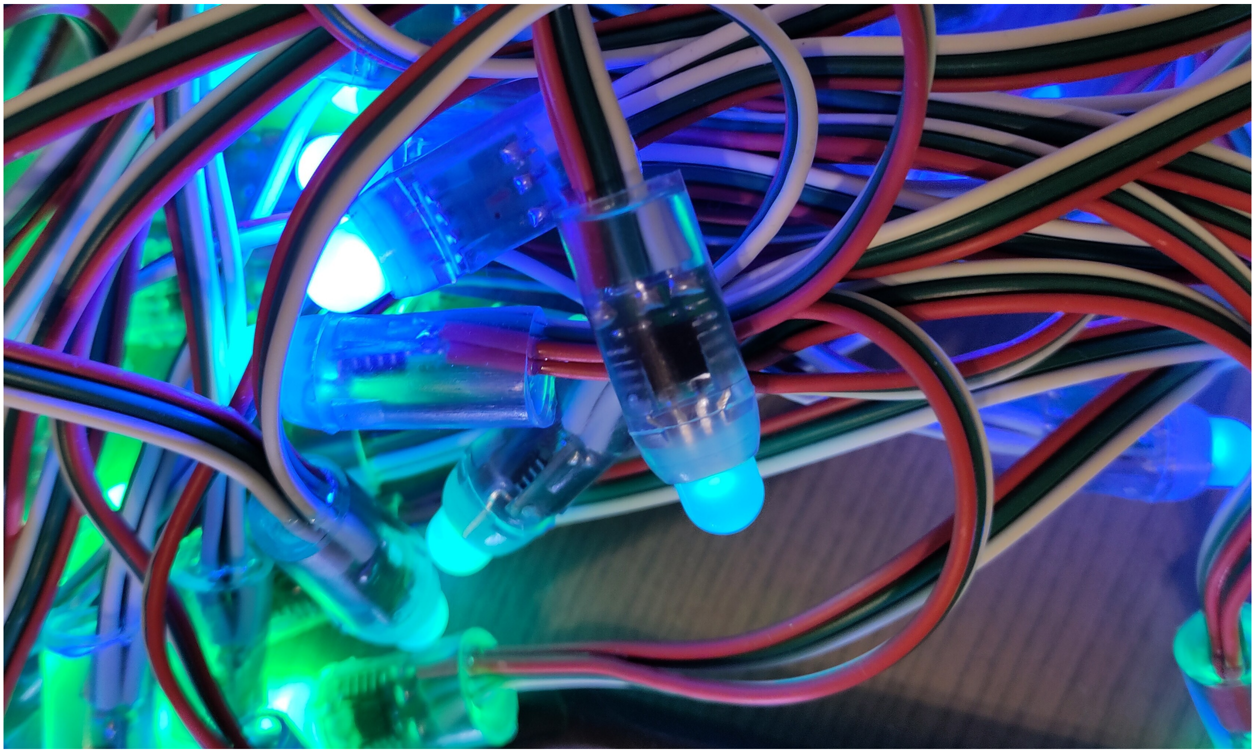
Single Chip LEDs

Those while small LEDs are in a 5050 package and often used on PCBs. All those light rings, stripes or any PCBs are most likely WS2812. FAST pinball sells PCBs which can be mounted below an insert and there are numerous other designs to buy around the internet.

TODO: Add a picture of WS2812 PCB

Christmas Lights

"Christmas Light" chains are very well known. You can buy them for around 15 bucks with 50 lights. Those work well for GIs but you can also use them below inserts if you print or bend holders for this.



Those LEDs are WS2811 in most of the cases. You typically see the controller as a separate chip and a RGB LED soldered to it.

Bulk WS2811

You can actually buy WS2811 in bulk from China. They are usually used to build christmas light chain but you can solder almost any LED to them. This could be flashers, custom playfield lights, segment displays or any other light you want to control. The chip will provide around 18.5mA per LED at full brightness.

TODO: Add a picture of a WS2811 PCB

Some of those controllers also support 12V power supply. The datasheet is inconsistent here. Absolute maximum rating are 6-7V but they also talk about 12V and 24V. So take care about that voltage when buying those PCBs.

WS2814 or SK6812

There is not much known about the inner working of those chips. But they work similar to the chips above but at `18.5mA * 4 = 74mA` total power.

TODO: Add a picture of a RGBW WS2814 and/or SK6812

WS2813

Those chips are similar to the WS2812 chips but they got an additional fallback input which connects to the output of the second last LED. If the previous LED in the chain breaks the chain will continue to work which is very convenient.

TODO: Add a picture of a WS2813

There exist four versions: * A and B run at `18mA * 3 = 53mA` (similar to WS2812) * C and D are low power version und run at `5mA * 3 = 15mA`

WS2815

WS2815 is a 5050 chip similar to WS2813 but it runs at 12V instead of 5V which is pretty interesting for pinball machines. This allows longer chains and thinner wires which is a huge improvement. Additionally, it has the handy fallback pin of WS2813 so one broken chip will not bring down the whole chain.

TODO: Add a picture of a WS2815

Loops / Orbits / Ramps

Related Config File Sections
sequence_shots:
sound_player:

Ramps, loops or orbits usually contain two switches. One at the entry and one to signal success. To detect only shots where both switches were hit in order you can use *sequence_shots*.

TODO: Add a picture of an orbit

```
switches:
  s_ramp_entry:
    number: 1
  s_ramp_success:
    number: 2
sequence_shots:
  ramp:
    switch_sequence: s_ramp_entry, s_ramp_success
    sequence_timeout: 3s
```

Additionally, most machines usually play a sound once the entry is hit to signal the player that he hit the ramp and another sound on success to indicate that the ball made it. You can use *sound_player:* to achieve that. In this example you would use the events `s_ramp_entry_active` and `ramp_hit` to play the sounds:

```
sound_player:
  s_ramp_entry_active: indicate_ramp
  s_ramp_success: indicate_ramp_success
```

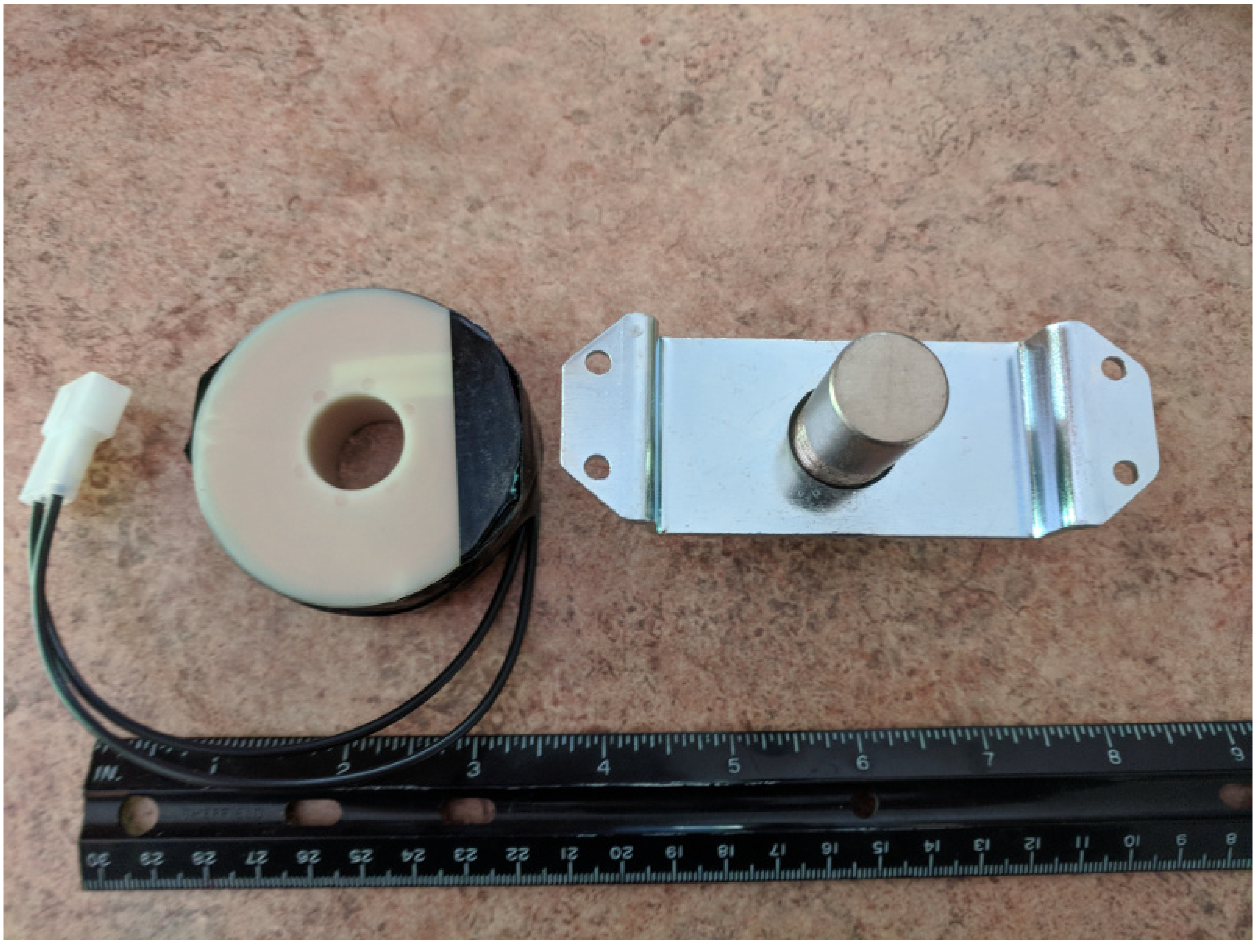
Magnets

Related Config File Sections
magnets:

MPF supports the ability to control precise timing for magnets which you can use to grab and release balls. It also includes the ability to set timings to “fling” a ball by grabbing, releasing, then pulsing the magnet again.


511-5065-ND


IN.
1
2
3
4
5



Video about magnets:

https://youtu.be/XGnrfO3eJD0

Hardware

Magnets are quite strong *single wound coils* and everything in the coils section also applies to them. Especially, the Strength and Current calculations apply to them. Expect a resistance in the range of 2 to 10 ohms for a magnet coil.

Connecting Magnets

Please refer to the *Connecting Coils section for single wound coils*.

If you do not have a diode on your magnet we recommend to add one. Magnets are strong coils and they can easily fry your driver board otherwise.

Magnets often got a thermal fuse soldered inline to the connectors. Those should not limit you in any way.

Part Numbers

Assemblies:

- PBL-100-0007-00 (with 511-5065-ND coil)

Coils:

- 20-10197
- 20-9247
- 511-5065-ND
- 90-5064-02
- A-15685

Dedicated driver boards:

You can use a board such as `520-5068-01` to connect up to three drivers to four logic level outputs (3 inputs + 1 clock). The board contains FETs with flyback diode and a logic buffer for further protection.

Config

This is an example:

```
coils:
  magnet_coil:
    number:
    default_pulse_ms: 100
    default_hold_power: 0.375

switches:
  grab_switch:
    number:

magnets:
  magnet:
    magnet_coil: magnet_coil
    grab_switch: grab_switch
    release_ball_events: magnet_release
    fling_ball_events: magnet_fling
```

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for magnets is `device.magnets.<name>`.

active Boolean (true/false) as to whether this magnet is actively on and in the powered state.

enabled Boolean (true/false) which shows whether this ball hold is enabled.

Related How To guides

How to use the Stern Magnet Processor Board

Related Events

- *magnet_(name)_grabbing_ball*
- *magnet_(name)_grabbed_ball*
- *magnet_(name)_releasing_ball*
- *magnet_(name)_released_ball*
- *magnet_(name)_flinging_ball*
- *magnet_(name)_flinged_ball*

How to use the Stern Magnet Processor Board

Related Config File Sections
digital_outputs:
switches:
shows:
show_player:

Stern uses a Magnet Processor Board (part number #520-6801-00) in Metallica for the coffin under the hammer. This board is special as it can detect the ball a ball near the magnet. In addition in can grab and hold the ball.

Connecting the Board

TODO: Add an image of the magnet processor board.

Supply Voltage

The itself magnet runs on 50VDC and the board needs a logic supply of 5VDC. Both of the ground rails seem do be isolated from each other and are not connected on the MPB itself. However, you need to connect those two GNDs to maintain common ground (see *common ground* for details).

Zero Crossing

The MPB has a 13VAC input on J1-8 which seems to be unused and is not needed for its operation.

Inputs and Outputs

There are 3 logic level (5V) inputs and 1 logic level output on the MPB. The inputs are called DATA6 (J1-5), DATA7 (J1-6) and STROBE (J1-7) on the MPB and are used to control the mode of operations of the MPB. The single output of the MPB is called Sw Drive (J1-4). This changes from low to high with ball detection. To enable the output to pass sufficient current to power an opto isolator (recommended) or to use it in a switch matrix, enable the output with +5V at Sw Return (J1-3).

Controller States

So let's go over the four states of the controller:

OFF:

Signals: D6 = low, D7 = low, pulse Strobe

The magnet does nothing.

GRAB:

Signals: D6 = high, D7 = low, pulse Strobe

The magnet activates for a second and then deactivates. This grabs the ball from centimeters away. GRAB is a one time event. The controller does not GRAB again unless it passes through another state or OFF- state first.

DETECT:

Signals: D6 = high, D7 = high, pulse Strobe

The magnet is pulsed with 4µs peaks to detect the ball. Ball detection requires a small distance of seperation between the magnet core and the ball. 3.5 mm of wood appears to be a compromize between detection and holding strength. The green LED on the MPB lights up when the ball is detected and the output pin 4 goes high.

HOLD(+DETECT):

Signals: D6 = low, D7 = high, pulse Strobe

The magnet is pulsed with three 4µs peaks (presumably) for detection and a longer pulse (presumably) for holding the ball on the magnet. The ball is held firmly, but the detection does not always work. Sometimes it was very reliable, sometimes there was no detection at all. This might require some more analysis.

Config

To use the magnet in MPF you can use the following config as starting point. Shows that pulse the input pins are run as a sequence of events, not as loops. The board will maintain its mode until instructed to change. Caution: if the board is in a hold mode and MPF crashes the board will remain in the hold mode until power is lost or MPF instructs it to enter the OFF- state.

```
digital_outputs:
  magnet_strobe:
    number: 1    # number depends on your platform
    type: driver
    enable_events: magnet_strobe_on
    disable_events: shutdown, magnet_strobe_off
```

(continues on next page)

(continued from previous page)

```
  magnet_d6:
    number: 2    # number depends on your platform
    type: driver
    enable_events: magnet_d6_on
    disable_events: shutdown, magnet_d6_off
  magnet_d7:
    number: 3    # number depends on your platform
    type: driver
    enable_events: magnet_d7_on
    disable_events: shutdown, magnet_d7_off

switches:
  s_detect:
    number: 1     # number depends on your platform

shows:
  magnet_state_off:
    - time: 0
      events:
        - magnet_d6_off
        - magnet_d7_off
    - time: 20ms
      events:
        - magnet_strobe_on
    - time: 30ms
      events:
        - magnet_strobe_off
    - time: 50ms
      events:
        - magnet_d6_off
        - magnet_d7_off
  magnet_state_detect:
    - time: 0
      events:
        - magnet_d6_on
        - magnet_d7_on
    - time: 20ms
      events:
        - magnet_strobe_on
    - time: 30ms
      events:
        - magnet_strobe_off
    - time: 50ms
      events:
        - magnet_d6_off
        - magnet_d7_off
  magnet_state_grab:
    - time: 0
      events:
        - magnet_d6_on
        - magnet_d7_off
    - time: 10ms
      events:
        - magnet_strobe_on
```

(continues on next page)

(continued from previous page)

```
  - time: 20ms
    events:
      - magnet_strobe_off
  - time: 50ms
    events:
      - magnet_d6_off
      - magnet_d7_off
magnet_state_hold:
  - time: 0
    events:
      - magnet_d6_off
      - magnet_d7_on
  - time: 20ms
    events:
      - magnet_strobe_on
  - time: 50ms
    events:
      - magnet_strobe_off
  - time: 70ms
    events:
      - magnet_d6_off
      - magnet_d7_off
```

You can then turn the controller into `detect` in a mode by posting the `magnet_state_detect` event. Then add an event_player based on `s_detect_active` to turn the controller into the `grab` state. Finally, after a few seconds turn it into the `hold` state and check the state of `s_detect` to see if the grab succeeded.

TODO: Add some example config for this logic.

Motors

Related Config File Sections
motors:
digital_outputs:
switches:

Hardware

TODO: Add a picture of a kickback

Help us to write it

Config

In this example we configure a motorized drop target bank which can move up and down with two position switches.

```
switches:
  s_position_up:
    number:
  s_position_down:
    number:

digital_outputs:
  c_motor_run:
    number:
    type: driver

motors:
  motorized_drop_target_bank:
    motor_left_output: c_motor_run
    position_switches: !!omap
      - up: s_position_up
      - down: s_position_down
    reset_position: down
    go_to_position:
      go_up: up
      go_down: down
```

The motor can run continuously and drives a camshaft which moves the bank up and down. MPF will figure the position using two position switches *s_position_up* and *s_position_down*. To enable the motor we use a digitial_output *c_motor_run* which maps to a driver. On reset the bank moves down and can afterwards be commanded using the events *go_up* and *go_down*.

The following is an example to drive the slimer in Stern Ghostbusters:

```
switches:
  s_slimer_home:
    number: 8-1
  s_slimer_away:
    number: 8-2

digital_outputs:
  c_slimer_motor_forward:
    number: 8-3
    type: light
  c_slimer_motor_backward:
    number: 8-4
    type: light

motors:
  ghostbusters_slimer:
    motor_left_output: c_slimer_motor_forward
    motor_right_output: c_slimer_motor_backward
    position_switches: !!omap
      - home: s_slimer_home
      - away: s_slimer_away
    reset_position: home
    go_to_position:
      slimer_home: home
      slimer_away: away
```

The slimer motor can move in two directions using two digital_outputs *c_slimer_motor_forward* and *c_slimer_motor_backward* which map to lights in Spike. The switches *s_slimer_home* and *s_slimer_away* are used by to determine the current position. To command the slimer use the events *slimer_home* or *slimer_away*.

Related Events

- *motor_(name)_reached_(position)*

Playfields

Related Config File Sections
playfields:
playfield_transfers:

- *Monitorable Properties*
- *Related Events*
- *Other playfield concepts*

Believe it or not, the playfield in MPF is technically a *ball device*. This is needed since MPF wants to know where all the balls are at all times, so it needs to know which balls are "in" the playfield device.

TODO: Add a picture of a playfield

The playfield is also responsible for tracking balls that "disappeared" from it without going into other devices—-a process which kicks off the *ball search*. The default playfield ball device (called *playfield*) is created automatically based on settings in the `mpfconfig.yaml` default configuration file. Most machines only have one playfield, though if you have a mini-playfield or a head-to-head machine then you can configure additional playfield devices.

Ball tracking and ball search is performed per playfield in MPF. Therefore, most devices in MPF belong to one playfield and mark it as active when they see a ball. You should configure the exact playfield for every device as soon as you have more than one playfield in your machine. Otherwise, MPF will complain about unexpected balls (e.g. you will see *unexpected_ball_on_(name)* events), ball search might at the wrong time and ball tracking might go haywire. To transfer balls you can use *playfield transfer* or *ball devices*. A ball device might capture from one playfield and eject to another.

Playfields are configured in the *playfields: section* of the configuration file.

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for playfields is `device.playfields.<name>`.

available_balls Balls which will be available eventually. If a ball is requested it will be included in available_balls but not in balls until it arrives.

balls The number of balls on the playfield.

Related Events

- *(name)_ball_count_change*
- *(name)_active*
- *sw_(name)_active*
- *unexpected_ball_on_(name)*

Other playfield concepts

How MPF tracks the number of balls on a playfield

In MPF, the "playfield" is technically a ball device, just like anything else that holds a ball (the trough, the plunger lane, a VUK, etc.). Any balls that are loose and rolling around the playfield can be considered to be "in" the playfield ball device.

Most ball devices in MPF have either (1) switches that a ball sitting in the device activates while sitting there (configured as `ball_switches:` in MPF), or (2) a switch that is momentarily activated when a ball rolls over it on its way in. (Configured as an `entrance_switch:` in MPF.)

But a playfield has none of these.

However, there are many switches in a pinball machine which are only hit by a ball that's on the playfield, and MPF uses these switches to know whether there's a ball on the playfield.

playfield_active switch tags

In MPF, you add a tag called `playfield_active` to the list of tags for every switch which is hit by a ball that's active on the playfield.

You do this in the `switches:` section of your machine config, like this:

```
switches:
  s_trough1:
    number:
  s_trough2:
    number:
  s_plunger_lane:
    number:
  s_standup_1:
    number:
    tags: playfield_active
  s_upper_right_rollover:
    number:
    tags: playfield_active
  s_ramp_enter:
    number:
    tags: playfield_active
  s_ramp_made:
    number:
    tags: playfield_active
```

Note that not every switch has the `playfield_active` tag, rather, it's just used for the switches that are hit when a ball is on the playfield.

Note that all switches which can be hit by a ball on the playfield are tagged, even if they're ramp switches since a ball rolling around a ramp is a ball on the playfield.

Tracking new balls added to the playfield

MPF also uses the playfield_active tags to know whether a ball has successfully been ejected from a ball device to the playfield.

If a ball device ejects to a playfield that has no balls on it, then the first time a switch tagged with `playfield_active` is hit, MPF knows the ball successfully made it out of the device and onto the playfield. Ball devices also have eject timeouts which will be used to confirm that a ball was ejected to the playfield if the timeout expires and the ball has not fallen back into the device that ejected it, which is useful since it's possible for the ball to make it out of the device but then not to hit a switch right away.

The playfield_active tagged switches are only used to confirm a ball ejects to the playfield if there are no current balls on the playfield when the device ejects a ball to it. If there is a ball (or multiple balls) on the playfield when a device ejects a ball to the playfield, then MPF doesn't know whether a hit to a playfield_active switch is from one of the current balls or the new ball, so in that case it always falls back to using the eject timeout to confirm that the ball successfully made it out.

These switches are used for ball search

MPF's *ball search* functionality uses the `playfield_active` switches to know whether a ball is stuck. (Basically every activation of one of these switches resets the ball search timer, and if that timer runs out and the player is not holding in a flipper button, then the ball search starts.)

So it's important to add the `playfield_active` tag to every switch that can be hit by a ball on the playfield.

Tagging switches with multiple playfields

If you have more than one playfield, then the "playfield_active" switch tag name should be adjusted to match the name of your actual playfield. For example, if you have a playfield called "upper_playfield", then the switches which are hit by a ball on the upper playfield should be tagged `upper_playfield_active`.

'Playfield' balls versus 'balls in play'

One important concept for ball tracking to understand is that there's a difference between the number of balls on a playfield and the "balls in play".

Most of the time, the number of balls rolling around the playfield is the same as the number of balls in play. However this is not always the case.

For example, when the machine tilts, the player's ball is "dead" and the number of balls in play is set to zero. But of course when that happens, there are still balls loose on the playfield which MPF has to track to make sure they all drain without getting stuck.

Also, if you have more than one playfield (like with an upper or lower playfield), then the number of balls on the individual playfields will be lower than the total number of balls in play.

Another time these two values are different is when the player shoots the ball into a lock. At that moment the playfield has no balls (and the lock has one), though there's technically still a ball in play.

Playfield transfer

Related Config File Sections
playfields:
playfield_transfers:

MPF Device

If you want to track balls across multiple playfields you can use a *playfield_transfer* device to move a ball from one playfield to another. This is mostly useful in head2head games. However, you can also use it to track balls on a mini-playfield. In some cases you can also use a *ball_device* which captures from one playfield and ejects to another playfield to achieve the same result.

Related Events

- *playfield_transfer_(playfield_transfer)_ball_transferred*

Plungers & Ball Launch Devices

Related Config File Sections
ball_devices:

A Plunger is a type of ball device. MPF supports mechanical (traditional "spring" plungers), coil-fired plungers, and combo auto/manual plungers.

Here are the options:

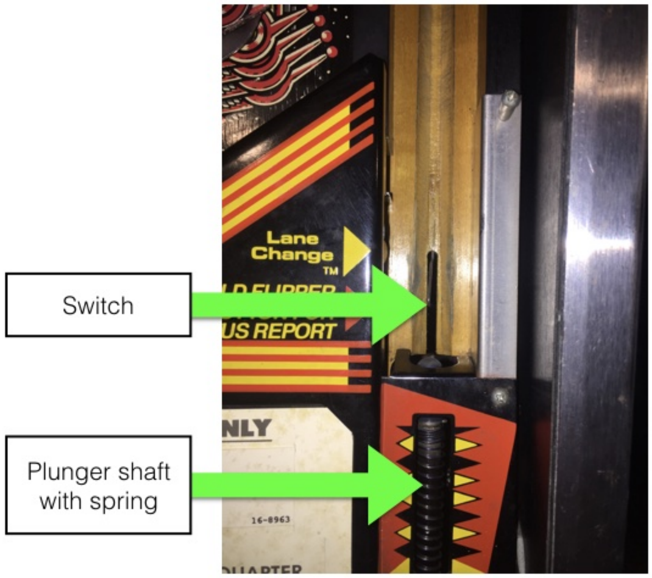
- *Mechanical (spring) plungers*
- *Plunger lanes with no ball switch*
- *Coil-fired plungers / ball launchers*
- *Combo (mechanical + coil-fired) plungers*

Since there are so many different options, you need to first identify which type of plunger or ball launch system your machine has. So look at the following pictures to match up what you have, and then follow the specific links to see how to configure MPF to use it in your machine.

Option 1: Spring plunger with ball switch

The most "traditional" style plunger is a spring-powered mechanical plunger lane. In modern machines, there's a switch at the bottom of the plunger lane which is activated by a ball sitting in the plunger lane waiting to be plunged.

Here's an example of this from a Pin*Bot machine:

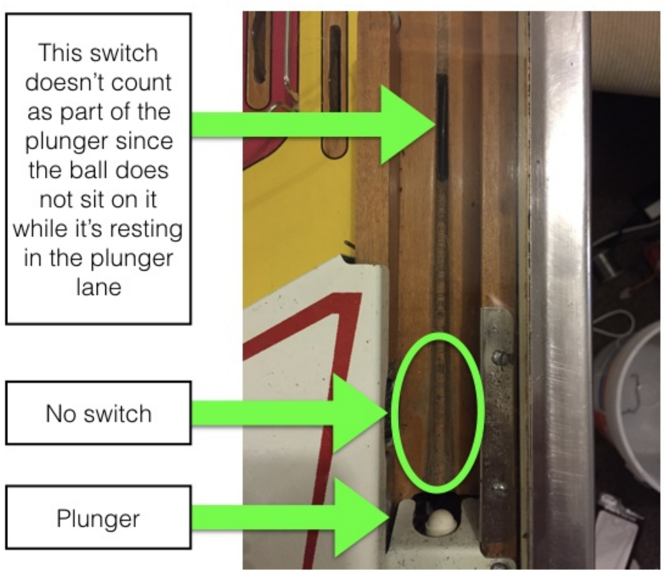


If you have this type of spring-powered plunger with a switch that's active when a ball is sitting in it ready to be plunged, follow the *Mechanical (spring) plungers* guide to configure it in MPF.

Option 2: Spring plunger with no ball switch

Older pinball machines (typically those that only have one ball) have what appear to be traditional plungers like in Option 1, but if you look closely, you'll notice that there is no switch which is active when the ball is sitting in the plunger lane.

Here's an example of this from Gottlieb Big Shot:

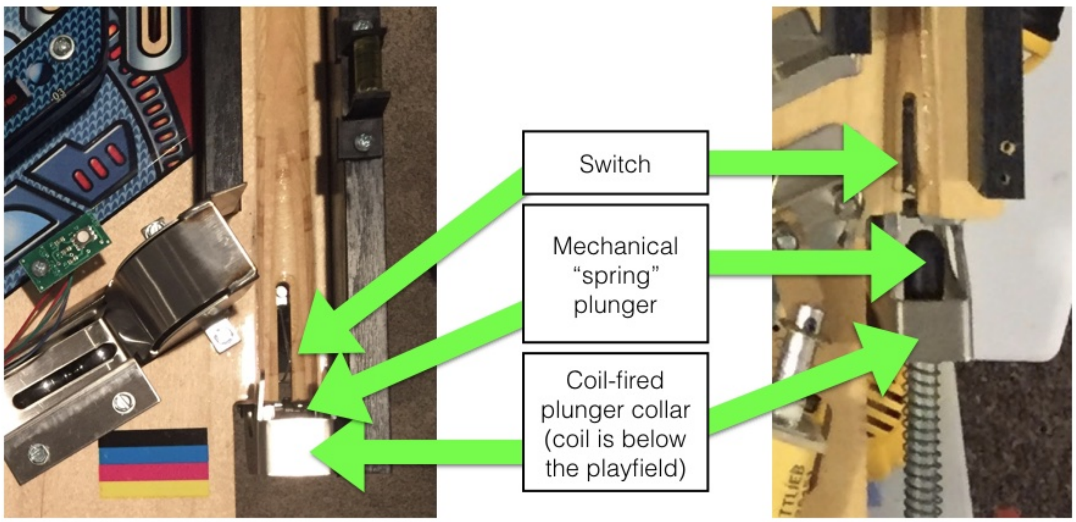


If you have this type of spring-powered plunger with **no** switch that's active when a ball is sitting in it ready to be plunged, follow the *Plunger lanes with no ball switch* guide to configure it in MPF.

Option 3: Combo spring plunger with coil-fired autolauncher

Many modern machines have a combination-style plunger which combines a mechanical spring-powered plunger with an autolauncher coil. These types of plungers allow game to decide whether the player should manually pull back on the plunger handle to launch the ball with spring power or whether the game should pulse a coil to eject the ball into play.

Here are two examples of slightly different versions of these, the left from a Stern Star Trek Premium, and the right from a Gottlieb Brooks 'n Dunn machine:

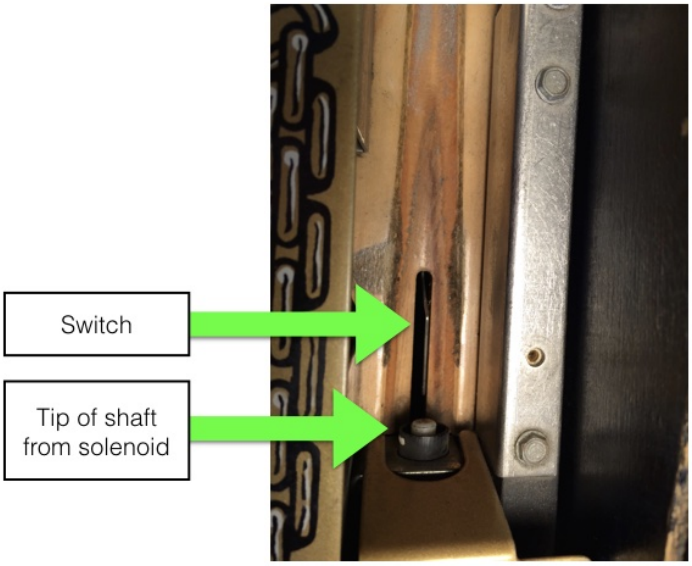


If you have this type of auto/manual combo plunger, follow the *Combo (mechanical + coil-fired) plungers* guide to configure it in MPF.

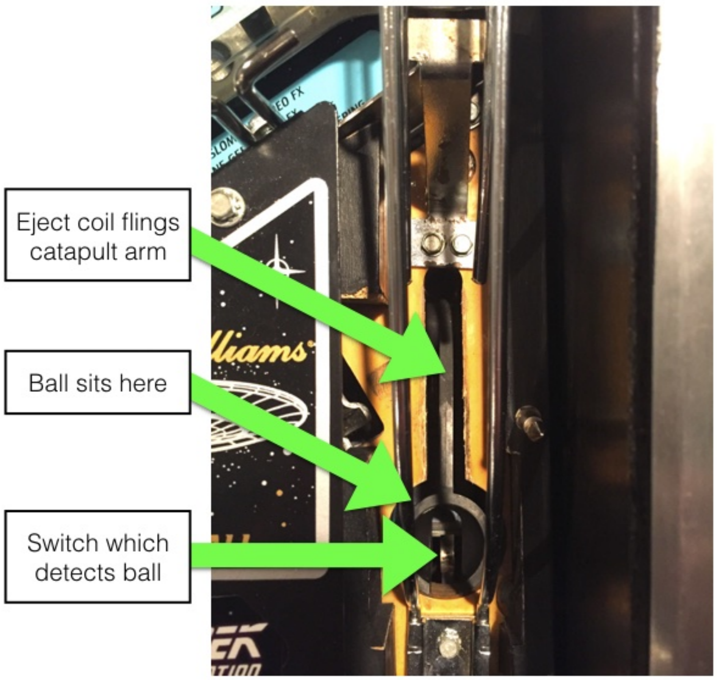
Option 4: Coil-fired plunger (no mechanical spring option)

The final plunger option is the fully automatic coil-fired option that has no mechanical spring-based option.

There are a few different physical forms of this. Here's a typical example from Judge Dredd where a coil shaft with a plastic tip is pulsed to launch the ball directly:



And here's an example from Williams Star Trek: The Next Generation which uses a catapult-style mechanism in order to launch the ball into play.



Note that both of these options are "identical" as far as MPF is concerned. They both have switches which are active when a ball is able to be launched, they both pulse coils to launch the ball, and neither one has a manual plunge option.

If you have this type of coil-powered plunger, follow the *Coil-fired plungers / ball launchers* guide to configure it in MPF.

Related How To Guides
Tutorial step 8: Add your plunger lane
Troubleshooting P-Roc/P3-Roc

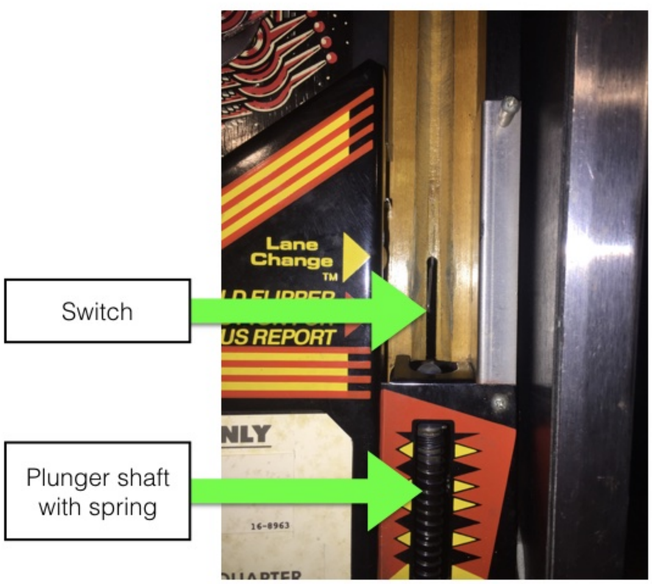
Related Events
balldevice_ball_missing
balldevice_balls_available
balldevice_(name)_ball_missing
balldevice_captured_from_(captures_from)
balldevice_(name)_ball_eject_attempt
balldevice_(name)_ball_eject_failed
balldevice_(name)_ball_eject_success
balldevice_(name)_ejecting_ball

Mechanical (spring) plungers

Related Config File Sections
ball_devices:
playfields:

This guide shows you how to configure a traditional mechanical spring plunger with MPF.

This guide is for use with a plunger lane that has a switch in the lane which is activated by a ball waiting to be plunged, like this:



If you have a mechanical spring plunger but you do NOT have a switch there, then follow the *Plunger lanes with no ball switch* guide instead.

If you have a mechanical spring plunger that also has an "auto launch" coil fired option, then follow the *Combo (mechanical + coil-fired) plungers* guide instead.

1. Add the switch

The first step is to add your plunger lane switches to the `switches:` section of your machine config file. Here's an example:

```
switches:
  s_plunger_lane:
    number: 2-6
```

Note that we configured this switches as number `2-6`, but you should use the actual switch numbers for your control system that the switches are connected to. (See *How to configure "number:" settings* for instructions for each type of control system.)

Be sure to set the `type: NC` if this switch is an opto and to configure the other switch settings as needed.

2. Add your plunger ball device

Remember a *ball device* is anything in your pinball machine that holds a ball (even if it's just for a short time). So your plunger lane is a ball device.

In this case, you can add an entry for your plunger to the `ball_devices:` section of your machine-wide config, and then create sub entries for the ball switch.

Here's an example. Note that in this case, we've left out the other ball devices (such as your trough and/or drain):

```
ball_devices:
  bd_plunger:
    ball_switches: s_plunger_lane
    mechanical_eject: true
```

In the example above, we named the plunger device *bd_plunger*, but if course you can name it whatever you want. You might use *bd_right_plunger* and *bd_left_plunger* for a game like *Red & Ted's Road Show* that has plunger lanes on both sides.

Note that the `ball_switches:` entry will just be a single switch, which is fine. Since there's only one switch listed in the `ball_switches:` section, that will tell MPF that this device can hold one ball.

3. Add the mechanical eject setting

Most ball devices in MPF have a coil which MPF pulses to eject a ball from the device. But in the case of a mechanical spring-powered plunger, there is no coil to eject the ball.

In this case, you have to tell MPF that this device has a mechanical eject option, which basically lets MPF know that the ball might suddenly disappear from this device, and when that happens, and eject attempt has been made.

To do that, add `mechanical_eject: true` to your plunger device, like this:

```
ball_devices:
  bd_plunger:
    ball_switches: s_plunger_lane
    mechanical_eject: true
```

4. Configure the eject confirmation, target & timeouts

Next you need to configure some settings that will let your plunger know whether ball launch events were successful.

The first setting is called `eject_targets:`. (You may remember this from when you *configured your trough or drain device*.) This setting is a list of one (or more, if there's a diverter) ball devices that your plunger lane ejects into.

In probably 99% of cases, the plunger device only ejects to the playfield. In that case you do *not* need to configure your `eject_targets:` because the playfield is the default setting.

However, if your plunger lane ejects to some other device (maybe another launcher or a subway or something) other than the playfield, then you'd configure that here.

Next up is the `confirm_eject_type:` which is how MPF knows that a ball really made it out of the plunger and won't fall back in.

In most cases, the default setting of "target" is fine (because that means that MPF just watches for the target device (from above) to get a ball, and when it does, it assumes the eject from this device was successful.

However, plunger lanes that eject to the playfield sometimes have a switch that's activated when the ball leaves the plunger. You can use this switch with a few caveats:

- If this switch has been hit, it means the ball is out for sure, and it's not possible for it to roll back.
- This switch must always be hit, e.g. the ball can't sneak around it.
- No other balls should be able to hit this switch while they're in play.

What this means is that this switch is pretty limited and almost never used.

Finally, you need to configure the `eject_timeouts:` which is a time setting for how long MPF will wait to confirm the eject. If a ball re-enters that device before the timeout happens, then MPF assumes the eject failed and will try it again.

For the `eject_timeouts:`, you want to figure out what the MAXIMUM time is that a ball could be ejected from the plunger but still not make it all the way out and then fall back into the plunger. You'll have to play with this setting in your machine, but in most machines it's probably around 3s.

Here are some examples of these settings in action.

First, for a typical coil-fired plunger lane / catapult that ejects the ball directly to the playfield: (This is probably 99% of all cases)

```
ball_devices:
  bd_plunger:
    # ...
    eject_timeouts: 3s
```

Next, for a coil-fired plunger that has a switch at the exit of the plunger lane that is only hit if the ball has made it out of the plunger and cannot be hit by a random ball on the playfield:

```
ball_devices:
  bd_plunger:
    # ...
    confirm_eject_type: switch
```

(continues on next page)

(continued from previous page)

```
    confirm_eject_switch: s_plunger_lane_exit
    eject_timeouts: 3s
```

Next, if your plunger lane ejects into another ball device (a cannon, in this case):

```
ball_devices:
  bd_plunger:
    # ...
    eject_targets: bd_cannon
    eject_timeouts: 2s
```

5. Set your trough/drain device eject_targets

Once you have your plunger device set up, you need to go back to your trough or ball drain device and add the new plunger to your trough's `eject_targets:`, like this:

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough_jam
    eject_coil: c_trough_eject
    tags: trough, home, drain
    jam_switch: s_trough_jam
    eject_coil_jam_pulse: 15ms
    eject_targets: bd_plunger
```

Of course you'd add the name that you gave your plunger device, which could be something like "bd_catapult" or whatever you called it.

Also, if you have a two-stage drain (like a System 11 machine), you'd add this to the second device (the one that feeds the plunger).

6. Add the plunger as default_source_device

Next you need to add your plunger lane ball device as `default_source_device` to your playfield to tell MPF that this ball device is used to add a new ball into play.

To do that, add your new plunger ball device as `default_source_device` in the default `playfield`, like this:

```
playfields:
  playfield:
    default_source_device: bd_plunger
    tags: default
```

7. Tag your playfield switches

Since the plunger lane ejects balls to the playfield, it's important that you have your playfield switches tagged properly since that's how MPF knows that a ball is loose on the playfield.

See the *How MPF tracks the number of balls on a playfield* documentation for details.

Complete config example

Here's a complete machine config with a "standard" coil-fired plunger that ejects the ball directly to the playfield. Note that this config does not include the switches and coils for the trough.

This config is what probably 99% of machines with coil-fired plungers will use:

```
switches:
  s_plunger_lane:
    number: 2-6
  s_launch_button:
    number: 1-5
  s_trough1:
    number: 3-1
  s_trough2:
    number: 3-2
  s_trough3:
    number: 3-3
  s_trough4:
    number: 3-4
  s_trough_jam:
    number: 3-5
coils:
  c_trough_eject:
    number: 3-1
    default_pulse_ms: 20
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough_jam
    eject_coil: c_trough_eject
    tags: trough, home, drain
    jam_switch: s_trough_jam
    eject_coil_jam_pulse: 15ms
    eject_targets: bd_plunger
  bd_plunger:
    ball_switches: s_plunger_lane
    mechanical_eject: true
    eject_timeouts: 3s
playfields:
  playfield:
    default_source_device: bd_plunger
    tags: default
```

What if it doesn't work?

Have a look at our *troubleshooting guide for ball_devices*.

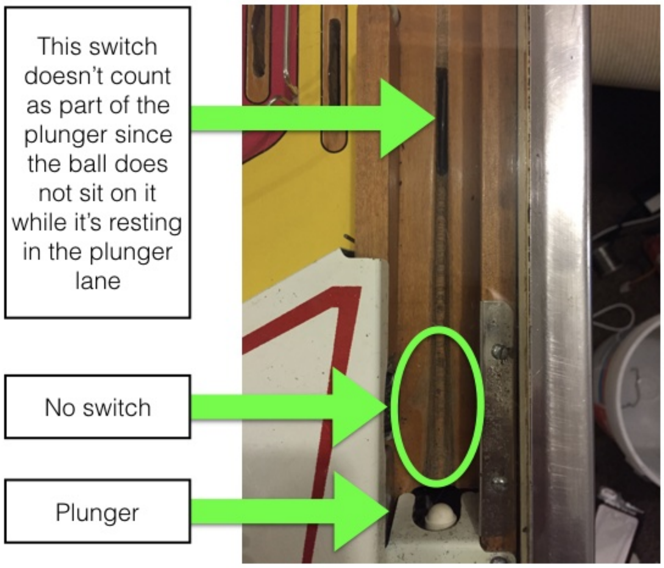
Plunger lanes with no ball switch

Related Config File Sections
ball_devices:
playfields:

Modern pinball machines have a switch in the plunger lane that tells the software that a ball is sitting in the plunger lane waiting to be plunged.

This document describes how you configure MPF to work with plunger lanes when the plunger lane has no switch which is active when a ball is sitting at the plunger. (This is common is older single-ball machines, including many EM and early solid state machines.)

Here's an example from a Gottlieb Big Shot




```
#config_version=5

playfields:
    playfield:
        default_source_device: trough
        tags: default

coils:
    trough_eject:
        number:

switches:
    s_trough_1:
        number:
    s_trough_2:
        number:
    s_trough_3:
        number:
    s_trough_4:
        number:
    s_trough_jam:
        number:
    s_playfield:
        number:
        tags: playfield_active
```

(continues on next page)

(continued from previous page)

```
ball_devices:
    trough:
        eject_coil: trough_eject
        ball_switches: s_trough_1, s_trough_2, s_trough_3, s_trough_4
        debug: true
        tags: trough, drain, home
```

1. Configure your trough / ball drain

MPF's plunger lanes work hand-in-hand with the trough / ball drain devices. So if you haven't configured that yet, go back and *do that now*, then come back here and configure your plunger.

2. Understand that your plunger is *not* a ball device

Most pinball machines have a switch in the plunger lane which is used to tell MPF that there's a ball in the plunger waiting to be plunged.

However, this How To guide is for plunger lanes with no ball switch. (If your plunger lane has a ball switch, then follow the *Mechanical (spring) plungers* guide instead.)

In machines where the plunger lane does not have a ball switch, that means that MPF has no idea whether a ball is in the plunger lane. That's totally fine, and MPF can support that no problem. However, in this case, *you do not configure your plunger lane as a ball device*!

Instead the plunger lane area is considered part playfield, so a ball in the plunger lane that's not sitting on a switch is just like any other area of the playfield where the ball might be rolling around while it's not on a switch.

3. Add the trough as default_source_device

Normally you would use your plunger device as source device for your playfield. But since your plunger lane with no switch is not a ball device, that means we have to go back to the trough ball device and use it as source device. Therefore, you need to add your trough ball device as `default_source_device` to your playfield to tell MPF that this ball device is used to add a new ball into play.

To do that, add your trough device as `default_source_device` in the default `playfield`, like this:

```
playfields:
  playfield:
    default_source_device: bd_trough
    tags: default
```

Then when MPF needs to add a live ball into play, it will eject a ball from the trough and you're all set!

4. What happens if MPF starts with a ball in the plunger?

One of the downsides to not having a switch in the plunger lane is that MPF has no way of knowing if there's a ball in there. Throughout the ordinary course of operation, this is fine, because MPF "knows" that the trough ejected a ball, and it "knows" when the ball is on the playfield, so if the trough has ejected a ball and that ball hasn't yet entered the playfield, MPF can "assume" that ball is in the plunger lane.

However, what happens if MPF boots up from scratch and there's a ball in the plunger lane? In that case, the ball is not activating any switches, so MPF really has no idea if the ball is in the plunger line (which is fine) or if the ball is stuck somewhere on the playfield (which is not fine).

Todo: This does not work yet. Let us know in the forum if you need it.

5. Configuring the ball save timer

Be sure to set your ball save start event based on a tag from your switches tagged with `playfield_active` rather than *ball_starting* or your trough eject confirmation, since you don't want the timer to start running when the ball is sitting in the plunger lane.

See the *Ball Saves* documentation for details.

What if it doesn't work?

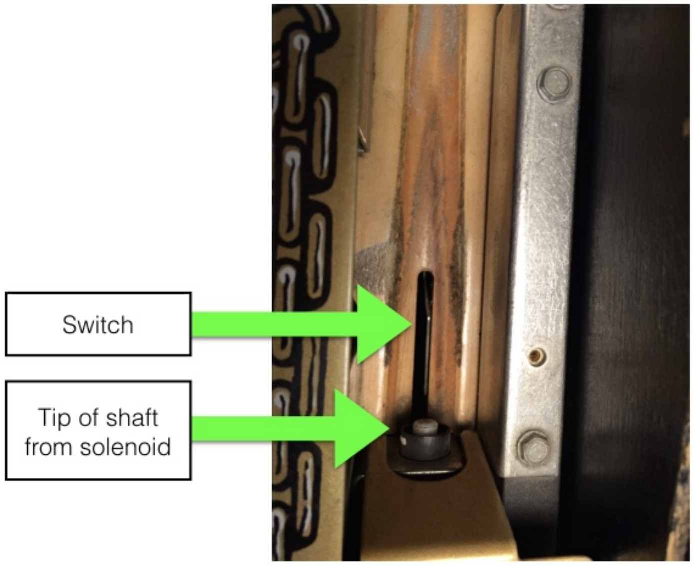
Have a look at our *troubleshooting guide for ball_devices*.

Coil-fired plungers / ball launchers

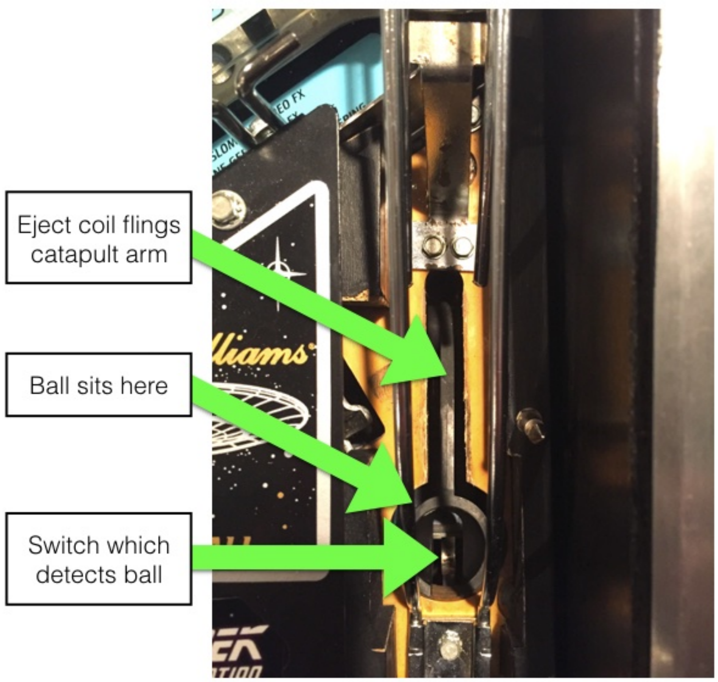
Related Config File Sections
ball_devices:
playfields:

Many modern pinball machines use some kind of "launch" button to launch the ball into play.

Sometimes these look more-or-less like traditional plunger lanes, except there's a solenoid instead of a spring-powered plunger, like this:



Other times these are more like "catapult" devices with a coil attached to the arm to launch the ball into play:



Note that if you have a coil-fired ball launcher that's combined with a spring plunger (giving the option for manual spring launches or machine-controlled auto launches, stop here and follow the *Combo (mechanical + coil-fired) plungers* guide instead.

1. Add the switches

The first step is to add your plunger's switches to the `switches:` section of your machine config file. Create an entry in the `switches:` section for both the switch in the device that's active when a ball is sitting in the plunger ready to be launched, and also create the entry for the switch connected to the button the player hits to launch the ball.

Here's an example:

```
switches:
  s_plunger_lane:
    number: 2-6
  s_launch_button:
    number: 1-5
```

Note that we configured this switches with numbers `2-6` and `1-5`, but you should use the actual switch numbers for your control system that the switches are connected to. (See *How to configure "number:" settings* for instructions for each type of control system.)

Be sure to set the `type: NC` if either of these switches is an opto and to configure the other switch settings as needed.

2. Add the coil

Next, create an entry in your `coils:` section of your machine config file for your plunger's eject coil. Again, the name doesn't matter. We'll call this *c_plunger* and enter it like this:

```
coils:
  c_plunger:
    number: 2-1
    default_pulse_ms: 20
```

Again, the `number:` entries in your config will vary depending on your actual hardware, and again, you can pick whatever name you want for your coil.

You'll also note that we went ahead and entered a `default_pulse_ms:` value of 20 which will override the default pulse time of 10ms. It's hard to say at this point what value you'll actually need. You can always adjust this at any time. You can play with the exact values in a bit once we finish getting everything set up.

3. Add your plunger / launcher ball device

Remember a *ball device* is anything in your pinball machine that holds a ball (even if it's just for a short time). So your plunger lane / ball launcher is a ball device.

In this case, you can add an entry for your plunger to the `ball_devices:` section of your machine-wide config, and then create sub entries for the ball switch and eject coil.

Here's an example. Note that in this case, we've left out the other ball devices (such as your trough and/or drain):

```
ball_devices:
  bd_plunger:
    ball_switches: s_plunger_lane
    eject_coil: c_plunger
```

In the example above, we named the plunger device *bd_plunger*, but if course you can name it whatever you want. You might use *bd_catapult* for a catapult-style launcher, or *bd_right_plunger* and *bd_left_plunger* for a game like Judge Dredd that has plunger lanes on both sides.

Note that the `ball_switches:` entry will just be a single switch. It's the switch that's active when a ball is sitting in the plunger waiting to be launched. (This is NOT the switch the player hits to launch the ball.)

Since there's only one switch listed in the `ball_switches:` section, that will tell MPF that this device can hold one ball.

4. Configure the launch switch

Next you need to configure the plunger lane so it launches the ball when the player hits the launch button. In MPF terms, this is technically the plunger "ejecting" the ball, so we use a setting called `player_controlled_eject_event:` which you add to your plunger.

At this point, you might be wondering why we configure a player controlled eject "event". Why is it an "event" and not a "switch"?

This is due to MPF's flexibility to support the myriad of different types of machines in the world.

For example, some machines launch the ball when a player hits a button. Others launch it when the player *releases* a button. Still others play a little show then launch. Etc.

So we decided, "Hey, we have this great events system in MPF, so let's just use that."

Remember that by default, there are "active" events that are posted when a switch becomes active, and "inactive" events that are posted when a switch that was active becomes inactive.

4.1 Launching the ball when a player hits the launch button

Assuming the switch tied to the launch button (or gun trigger or fishing rod button or whatever you have) is called *s_launch_button*, then that means an event called *s_launch_button_active* will be posted as soon as that switch is hit. In that case, you'd configure your plunger like this:

```
ball_devices:
  bd_plunger:
    ball_switches: s_plunger_lane
    eject_coil: c_plunger
    player_controlled_eject_event: s_launch_button_active
```

Pretty straightforward.

4.2 Launching the ball when a player releases the launch button

If you want to launch the ball into play when the player *releases* the launch button, then just use that switch's inactive event:

```
ball_devices:
  bd_plunger:
    ball_switches: s_plunger_lane
```

(continues on next page)

(continued from previous page)

```
    eject_coil: c_plunger
    player_controlled_eject_event: s_launch_button_inactive
```

Note that whenever the `player_controlled_eject_event:` is used, MPF has to specifically enable the ability for that event to eject a ball. In other words, you don't have to worry about the player hitting that switch to launch extra balls into play, and it's fine if that event is posted in other places in your game.

5. Configure the eject confirmation, target & timeouts

Next you need to configure some settings that will let your plunger know whether ball launch events were successful.

The first setting is called `eject_targets:`. (You may remember this from when you *configured your trough or drain device*.) This setting is a list of one (or more, if there's a diverter) ball devices that your plunger lane ejects into.

In probably 99% of cases, the plunger device only ejects to the playfield. In that case you do *not* need to configure your `eject_targets:` because the playfield is the default setting.

However, if your plunger lane ejects to some other device (maybe another launcher or a subway or something) other than the playfield, then you'd configure that here.

Next up is the `confirm_eject_type:` which is how MPF knows that a ball really made it out of the plunger and won't fall back in.

In most cases, the default setting of "target" is fine (because that means that MPF just watches for the target device (from above) to get a ball, and when it does, it assumes the eject from this device was successful.

However, plunger lanes that eject to the playfield sometimes have a switch that's activated when the ball leaves the plunger. You can use this switch with a few caveats:

- If this switch has been hit, it means the ball is out for sure, and it's not possible for it to roll back.
- This switch must always be hit, e.g. the ball can't sneak around it.
- No other balls should be able to hit this switch while they're in play.

What this means is that this switch is pretty limited and almost never used.

Finally, you need to configure the `eject_timeouts:` which is a time setting for how long MPF will wait to confirm the eject. If a ball re-enters that device before the timeout happens, then MPF assumes the eject failed and will try it again.

For the `eject_timeouts:`, you want to figure out what the MAXIMUM time is that a ball could be ejected from the plunger but still not make it all the way out and then fall back into the plunger. You'll have to play with this setting in your machine, but in most machines it's probably around 3s.

Here are some examples of these settings in action.

First, for a typical coil-fired plunger lane / catapult that ejects the ball directly to the playfield: (This is probably 99% of all cases)

```
ball_devices:
  bd_plunger:
```

(continues on next page)

(continued from previous page)

```
    # ...
    eject_timeouts: 3s
```

Next, for a coil-fired plunger that has a switch at the exit of the plunger lane that is only hit if the ball has made it out of the plunger and cannot be hit by a random ball on the playfield:

```
ball_devices:
  bd_plunger:
    # ...
    confirm_eject_type: switch
    confirm_eject_switch: s_plunger_lane_exit
    eject_timeouts: 3s
```

Next, if your plunger lane ejects into another ball device (a cannon, in this case):

```
ball_devices:
  bd_plunger:
    # ...
    eject_targets: bd_cannon
    eject_timeouts: 2s
```

6. Set your trough/drain device eject_targets

Once you have your plunger device set up, you need to go back to your trough or ball drain device and add the new plunger to your trough's `eject_targets:`, like this:

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough_jam
    eject_coil: c_trough_eject
    tags: trough, home, drain
    jam_switch: s_trough_jam
    eject_coil_jam_pulse: 15ms
    eject_targets: bd_plunger
```

Of course you'd add the name that you gave your plunger device, which could be something like "bd_catapult" or whatever you called it.

Also, if you have a two-stage drain (like a System 11 machine), you'd add this to the second device (the one that feeds the plunger).

7. Add the plunger as default_source_device

Next you need to your plunger lane ball device `default_source_device` to your playfield to tell MPF that this ball device is used to add a new ball into play.

To do that, add your new plunger ball device as `default_source_device` in the default `playfield`, like this:

```
playfields:
  playfield:
```

(continues on next page)

(continued from previous page)

```
    default_source_device: bd_plunger
    tags: default
```

8. Tag your playfield switches

Since the plunger lane ejects balls to the playfield, it's important that you have your playfield switches tagged properly since that's how MPF knows that a ball is loose on the playfield.

See the *How MPF tracks the number of balls on a playfield* documentation for details.

Complete config example

Here's a complete machine config with a "standard" coil-fired plunger that ejects the ball directly to the playfield. Note that this config does not include the switches and coils for the trough.

This config is what probably 99% of machines with coil-fired plungers will use:

```
switches:
  s_plunger_lane:
    number: 2-6
  s_launch_button:
    number: 1-5
  s_trough1:
    number: 3-1
  s_trough2:
    number: 3-2
  s_trough3:
    number: 3-3
  s_trough4:
    number: 3-4
  s_trough_jam:
    number: 3-5
coils:
  c_plunger:
    number: 2-1
    default_pulse_ms: 20
  c_trough_eject:
    number: 3-1
    default_pulse_ms: 20
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough_jam
    eject_coil: c_trough_eject
    tags: trough, home, drain
    jam_switch: s_trough_jam
    eject_coil_jam_pulse: 15ms
    eject_targets: bd_plunger
  bd_plunger:
    ball_switches: s_plunger_lane
    eject_coil: c_plunger
    player_controlled_eject_event: s_launch_button_active
    eject_timeouts: 3s
```

(continues on next page)

(continued from previous page)

```
playfields:
  playfield:
    default_source_device: bd_plunger
    tags: default
```

What if it doesn't work?

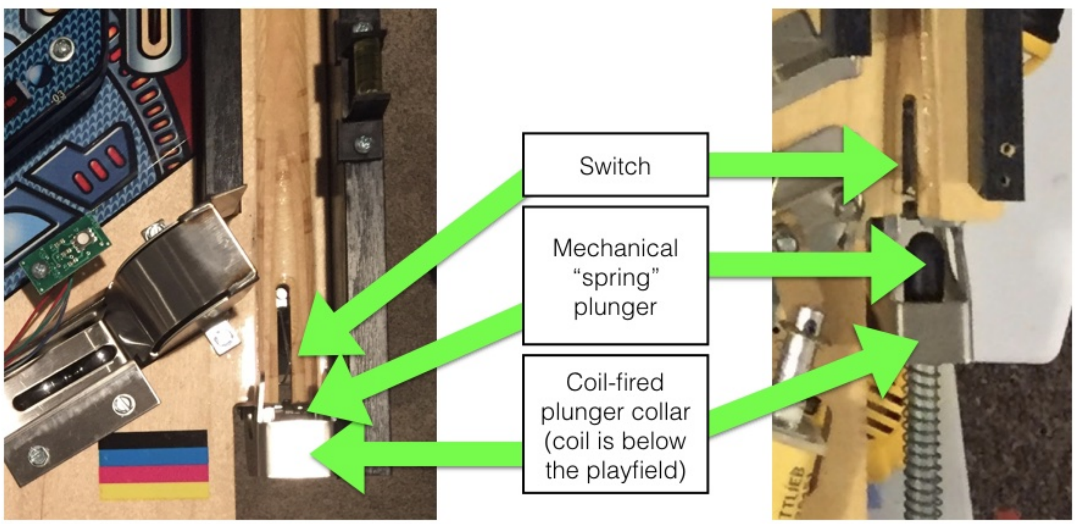
Have a look at our *troubleshooting guide for ball_devices*.

Combo (mechanical + coil-fired) plungers

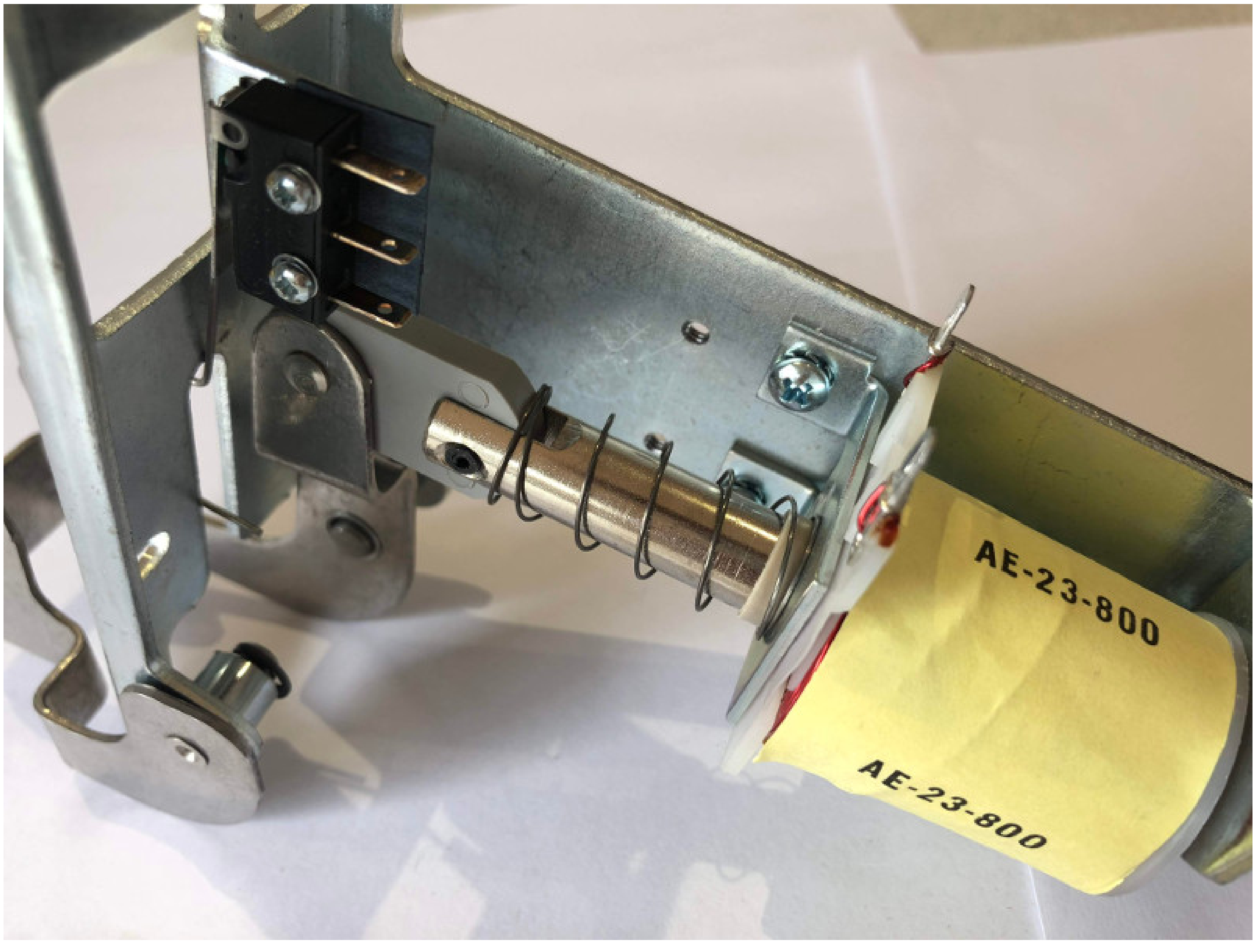
Related Config File Sections
ball_devices:
playfields:

This guide explains how to configure a "combo" plunger lane which has both a mechanical spring-powered plunger as well as a coil-fired auto plunge option.

Here's an example of this:




AE-23-800



If you have a purely mechanical plunger with no autolaunch option, follow the *Mechanical (spring) plungers* guide instead. If you have a standard coil-fired plunger or launch device with no mechanical spring plunger, follow the *Coil-fired plungers / ball launchers* guide instead.

Note: If you're reading through this guide and comparing it to the guide for the coil-fired plunger lane, you'll find that they're almost identical, except that this guide adds the `mechanical_eject: true` setting to the plunger.

1. Add the switches

The first step is to add your plunger's switches to the `switches:` section of your machine config file. Create an entry in the `switches:` section for the switch which is in the plunger lane that's activated by a ball waiting to be plunged.

You might also have a button which the player can hit to launch balls into play. Some machines have this (Like *Stern Star Trek* with the button on the apron), while others only let the player launch the ball with spring plunger and they use the coil for ball save and multiballs only.

So add one (or both, if you have a launch button) to your machine config if you haven't done so already:

```
switches:
  s_plunger_lane:
    number: 2-6
  s_launch_button:
    number: 1-5
```

Note that we configured this switches with numbers `2-6` and `1-5`, but you should use the actual switch numbers for your control system that the switches are connected to. (See *How to configure "number:" settings* for instructions for each type of control system.)

Be sure to set the `type: NC` if either of these switches is an opto and to configure the other switch settings as needed.

2. Add the coil

Next, create an entry in your `coils:` section of your machine config file for your plunger lane's eject coil. Again, the name doesn't matter. We'll call this *c_plunger* and enter it like this:

```
coils:
  c_plunger:
    number: 2-1
    default_pulse_ms: 20
```

Again, the `number:` entries in your config will vary depending on your actual hardware, and again, you can pick whatever name you want for your coil.

You'll also note that we went ahead and entered a `default_pulse_ms:` value of 20 which will override the default pulse time of 10ms. It's hard to say at this point what value you'll actually need. You can always adjust this at any time. You can play with the exact values in a bit once we finish getting everything set up.

3. Add your plunger / launcher ball device

Remember a *ball device* is anything in your pinball machine that holds a ball (even if it's just for a short time). So your plunger lane / ball launcher is a ball device.

In this case, you can add an entry for your plunger to the `ball_devices:` section of your machine-wide config, and then create sub entries for the ball switch and eject coil.

Here's an example. Note that in this case, we've left out the other ball devices (such as your trough and/or drain):

```
ball_devices:
  bd_plunger:
    ball_switches: s_plunger_lane
    eject_coil: c_plunger
```

In the example above, we named the plunger device *bd_plunger*, but if course you can name it whatever you want. You might use *bd_catapult* for a catapult-style launcher, or *bd_right_plunger* and *bd_left_plunger* for a game like Judge Dredd that has plunger lanes on both sides.

Note that the `ball_switches:` entry will just be a single switch. It's the switch that's active when a ball is sitting in the plunger waiting to be launched. (This is NOT the switch the player hits to launch the ball if you have one of those.)

Since there's only one switch listed in the `ball_switches:` section, that will tell MPF that this device can hold one ball.

4. Add the mechanical eject setting

Since your plunger ball device has an option for the player to manually plunge the ball with the spring rod, we need to give MPF a "heads up" that a ball sitting in the plunger lane might suddenly disappear, and that when that happens, that means the player has attempted to eject the ball from this device.

To do that, add `mechanical_eject: true` to your plunger device, like this:

```
ball_devices:
  bd_plunger:
    ball_switches: s_plunger_lane
    eject_coil: c_plunger
    mechanical_eject: true
```

5. (Optional) Configure the launch switch

If your machine also has a launch button which you'd like to (optionally) use for the player to hit to launch the ball into play with the plunger lane's eject coil, then you can add a setting called `player_controlled_eject_event:`.

At this point, you might be wondering why we configure a player controlled eject "event". Why is it an "event" and not a "switch"?

This is due to MPF's flexibility to support the myriad of different types of machines in the world.

For example, some machines launch the ball when a player hits a button. Others launch it when the player *releases* a button. Still others play a little show then launch. Etc.

So we decided, "Hey, we have this great events system in MPF, so let's just use that."

Remember that by default, there are "active" events that are posted when a switch becomes active, and "inactive" events that are posted when a switch that was active becomes inactive.

5.1 Launching the ball when a player hits the launch button

Assuming the switch tied to the launch button (or gun trigger or fishing rod button or whatever you have) is called *s_launch_button*, then that means an event called *s_launch_button_active* will be posted as soon as that switch is hit. In that case, you'd configure your plunger like this:

```
ball_devices:
  bd_plunger:
    ball_switches: s_plunger_lane
    eject_coil: c_plunger
    mechanical_eject: true
    player_controlled_eject_event: s_launch_button_active
```

Pretty straightforward.

5.2 Launching the ball when a player releases the launch button

If you want to launch the ball into play when the player *releases* the launch button, then just use that switch's inactive event:

```
ball_devices:
  bd_plunger:
    ball_switches: s_plunger_lane
    eject_coil: c_plunger
    mechanical_eject: true
    player_controlled_eject_event: s_launch_button_inactive
```

Note that whenever the `player_controlled_eject_event:` is used, MPF has to specifically enable the ability for that event to eject a ball. In other words, you don't have to worry about the player hitting that switch to launch extra balls into play, and it's fine if that event is posted in other places in your game.

6. Configure the eject confirmation, target & timeouts

Next you need to configure some settings that will let your plunger know whether ball launch events were successful.

The first setting is called `eject_targets:`. (You may remember this from when you *configured your trough or drain device*.) This setting is a list of one (or more, if there's a diverter) ball devices that your plunger lane ejects into.

In probably 99% of cases, the plunger device only ejects to the playfield. In that case you do *not* need to configure your `eject_targets:` because the playfield is the default setting.

However, if your plunger lane ejects to some other device (maybe another launcher or a subway or something) other than the playfield, then you'd configure that here.

Next up is the `confirm_eject_type:` which is how MPF knows that a ball really made it out of the plunger and won't fall back in.

In most cases, the default setting of "target" is fine (because that means that MPF just watches for the target device (from above) to get a ball, and when it does, it assumes the eject from this device was successful.

However, plunger lanes that eject to the playfield sometimes have a switch that's activated when the ball leaves the plunger. You can use this switch with a few caveats:

- If this switch has been hit, it means the ball is out for sure, and it's not possible for it to roll back.
- This switch must always be hit, e.g. the ball can't sneak around it.
- No other balls should be able to hit this switch while they're in play.

What this means is that this switch is pretty limited and almost never used.

Finally, you need to configure the `eject_timeouts:` which is a time setting for how long MPF will wait to confirm the eject. If a ball re-enters that device before the timeout happens, then MPF assumes the eject failed and will try it again.

For the `eject_timeouts:`, you want to figure out what the MAXIMUM time is that a ball could be ejected from the plunger but still not make it all the way out and then fall back into the plunger. You'll have to play with this setting in your machine, but in most machines it's probably around 3s.

Here are some examples of these settings in action.

First, for a typical coil-fired plunger lane / catapult that ejects the ball directly to the playfield: (This is probably 99% of all cases)

```
ball_devices:
  bd_plunger:
    # ...
    eject_timeouts: 3s
```

Next, for a coil-fired plunger that has a switch at the exit of the plunger lane that is only hit if the ball has made it out of the plunger and cannot be hit by a random ball on the playfield:

```
ball_devices:
  bd_plunger:
    # ...
    confirm_eject_type: switch
    confirm_eject_switch: s_plunger_lane_exit
    eject_timeouts: 3s
```

Next, if your plunger lane ejects into another ball device (a cannon, in this case):

```
ball_devices:
  bd_plunger:
    # ...
    eject_targets: bd_cannon
    eject_timeouts: 2s
```

7. Set your trough/drain device eject_targets

Once you have your plunger device set up, you need to go back to your trough or ball drain device and add the new plunger to your trough's `eject_targets:`, like this:

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough_jam
    eject_coil: c_trough_eject
    tags: trough, home, drain
    jam_switch: s_trough_jam
    eject_coil_jam_pulse: 15ms
    eject_targets: bd_plunger
```

Of course you'd add the name that you gave your plunger device, which could be something like "bd_catapult" or whatever you called it.

Also, if you have a two-stage drain (like a System 11 machine), you'd add this to the second device (the one that feeds the plunger).

8. Add the plunger as a default_source_device

Next you need to add your plunger lane ball device `default_source_device` to your playfield to tell MPF that this ball device is used to add a new ball into play.

To do that, add your new plunger ball device as `default_source_device` in the default `playfield`, like this:

```
playfields:
  playfield:
    default_source_device: bd_plunger
    tags: default
```

9. Tag your playfield switches

Since the plunger lane ejects balls to the playfield, it's important that you have your playfield switches tagged properly since that's how MPF knows that a ball is loose on the playfield.

See the *How MPF tracks the number of balls on a playfield* documentation for details.

Complete config example

Here's a complete machine config with a "standard" coil-fired plunger that ejects the ball directly to the playfield. Note that this config does not include the switches and coils for the trough.

This config is what probably 99% of machines with coil-fired plungers will use:

```
switches:
  s_plunger_lane:
    number: 2-6
  s_launch_button:
    number: 1-5
  s_trough1:
    number: 3-1
  s_trough2:
    number: 3-2
  s_trough3:
    number: 3-3
  s_trough4:
    number: 3-4
  s_trough_jam:
    number: 3-5
coils:
  c_plunger:
    number: 2-1
    default_pulse_ms: 20
  c_trough_eject:
    number: 3-1
    default_pulse_ms: 20
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough_jam
    eject_coil: c_trough_eject
    tags: trough, home, drain
    jam_switch: s_trough_jam
    eject_coil_jam_pulse: 15ms
    eject_targets: bd_plunger
  bd_plunger:
```

(continues on next page)

(continued from previous page)

```
    ball_switches: s_plunger_lane
    eject_coil: c_plunger
    mechanical_eject: true
    player_controlled_eject_event: s_launch_button_active
    eject_timeouts: 3s
playfields:
  playfield:
    default_source_device: bd_plunger
    tags: default
```

What if it doesn't work?

Have a look at our *troubleshooting guide for ball_devices*.

Pop Bumpers

Related Config File Sections
autofire_coils:
switches:
lights:
coils:

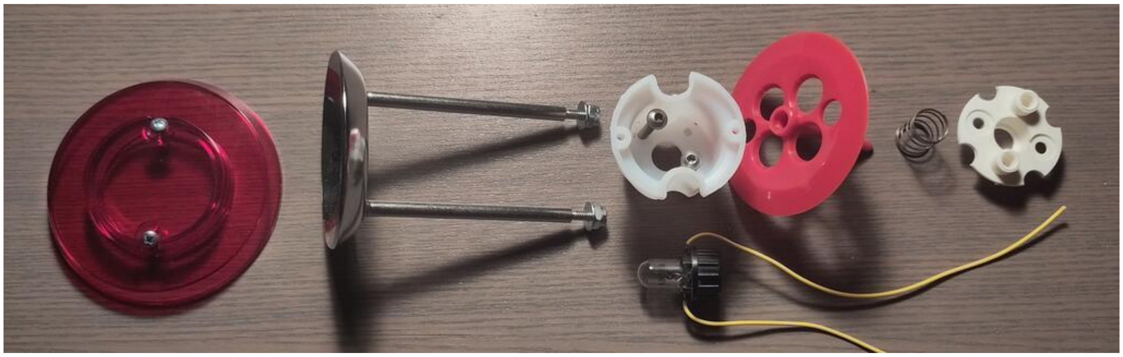
Popbumpers are configured as *autofire_coils* in MPF.

Hardware

Pop bumpers are made of three elements relevant for MPF:

- A *blade switch* to notice balls
- A #444 or #249 *bulb* for light shows
- A *coil* to push the ball away.

There are of course other parts, like the pop bumper body, all parts need to match in size and dimension of course. The complete assembly consists of parts above the playfield and some parts below the playfield. Above the playfield the following parts need to be assembled in the following order:

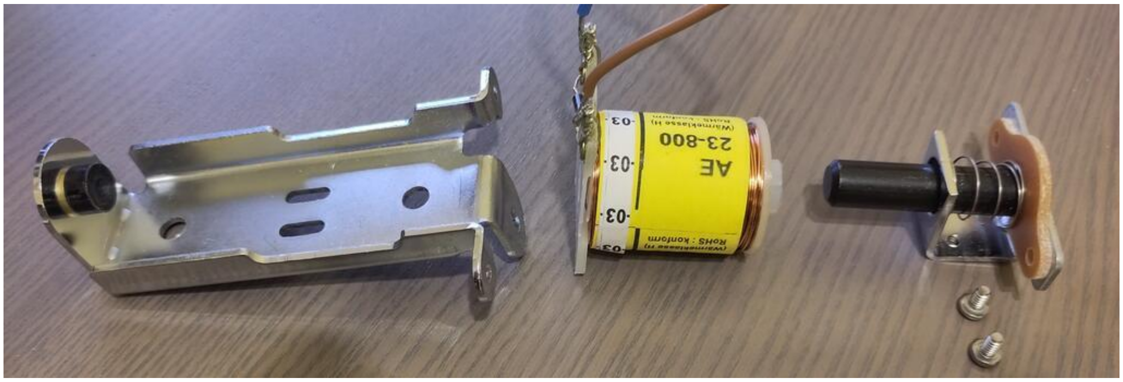


A few notes which might help to assemble the parts above the playfield:

- The nuts on the rod for the metal ring don't use a metric threading. To fasten them a 8mm wrench is good, but don't try to use a metric nut.
- The two screws to mount the cap are of size 3,0 x 12 mm, use pan head screws. When you screw the parts multiple times together turn the screw first counter clockwise to use exactly the same thread cut by the screw from the previous assembly. Otherwise the body will be worn out after a while.

- The two screws to mount the body to the board are of size 3,5 x 25mm, use pan head screws.
- You need a 17mm Fostner drill to drill hole for the base.
- You need a 8mm drill for the metal ring rods.

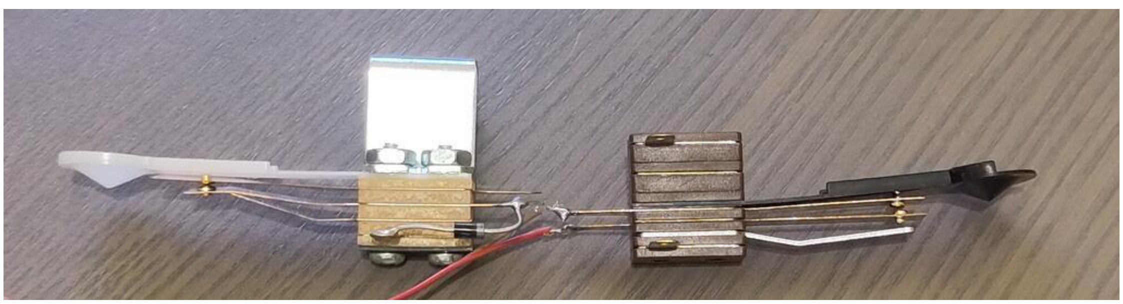
Below the playfield the parts needed look like this:



A few notes which might help to assemble the parts below the playfield:

- The coil specification depends on the voltage you use and the strength you would like to have. Probably a trial and error approach is needed to find the coil you want.
- The piece with the spring on the right hand side consists of 5 parts: yoke metal, yoke fiber, spring, metal bracket (plunger bracket), plunger
- The screws are again not metric
- If your coil has a fly back diode make sure to connect it the right way around. The ring of the diode has to point to positive voltage. You might not have a diode and you might not need one (if your controller board has that already built in)

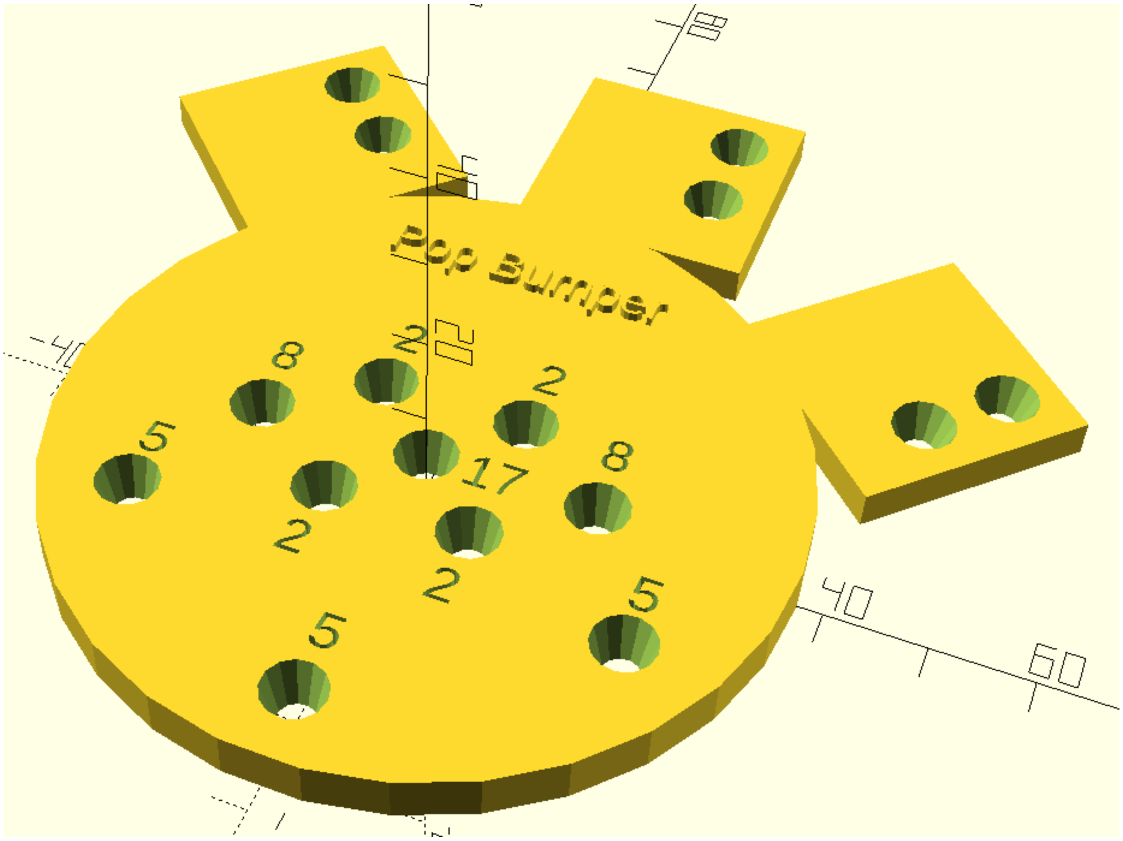
Last but not least below the playfield you need a leaf switch:



The leaf switch is closed by the pin of the pop bumper skirt when the skirt pressed down by the ball. It is important to adjust it very precisely. Again a few notes:

- A diode on the switch is needed if you use a matrix input, for direct inputs no diode is needed
- You need the switch and a mounting bracket (like the left leaf switch on the picture above)
- The switches come in different heights, make sure to get the height you need. (compare the two variations on the picture)
- Try to get a translucent leaf that helps to adjust it later since the skirt's pin can be seen from below

When mounting the pop bumpers you have to drill quite a few holes for home brew machine, having a stencil might be helpful.



The above stencil has been created with openSCAD. Feel free to use the `scad file` or the `stl file` for your own purposes.

Part numbers:

- Older one part plastic bumpers: 500-5227-00, AS-2999 (Turbo bumpers)
- Modern bumpers: 515-6459-04/A-9415 and B-9414

Config

This is an example:

```
switches:
  s_popbumper_left:
    number: 7                # depends on your platform
coils:
  c_popbumper_left:
    number: 4                # depends on your platform
    default_pulse_ms: 23     # tune this for your machine
lights:
  l_popbumper_left:
    number: 13               # depends on your platform
    subtype: matrix          # might be differnt
autofire_coils:
```

(continues on next page)

(continued from previous page)

```
  ac_popbumper_left:
    coil: c_popbumper_left
    switch: s_popbumper_left
```

Adjust `default_pulse_ms` and `default_pulse_power` in your coil to control the strength and sound of your popbumpers.

Related How To Guides
Mechanical Switches
Matrix Lights (Bulbs)
Coils (Solenoids)
Autofire Coils

How to Configure Score Reels

Related Config File Sections
score_reels:
score_reel_groups:

Multiple score reels are grouped to show the player score. Score reels detect certain position using switches (usually 0)

TODO: Add a picture of score reels

This is an example:

```
lights:
  light_p1:
    number:
    tags: player1
  light_p2:
    number:
    tags: player2
switches:
  score_1p_10k_0:
    number:
  score_1p_1k_0:
    number:
  score_1p_100_0:
    number:
  score_1p_10_0:
    number:
  score_2p_10k_0:
    number:
  score_2p_1k_0:
    number:
  score_2p_100_0:
    number:
  score_2p_10_0:
    number:
```

(continues on next page)

(continued from previous page)

```
coils:
  player1_10k:
    number:
  player1_1k:
    number:
  player1_100:
    number:
  player1_10:
    number:
  player2_10k:
    number:
  player2_1k:
    number:
  player2_100:
    number:
  player2_10:
    number:
  chime1:
    number:
  chime2:
    number:
  chime3:
    number:
score_reels:
  score_1p_10k:
    coil_inc: player1_10k
    switch_0: score_1p_10k_0
    limit_hi: 9
    limit_lo: 0
  score_1p_1k:
    coil_inc: player1_1k
    switch_0: score_1p_1k_0
    limit_hi: 9
    limit_lo: 0
  score_1p_100:
    coil_inc: player1_100
    switch_0: score_1p_100_0
    limit_hi: 9
    limit_lo: 0
  score_1p_10:
    coil_inc: player1_10
    switch_0: score_1p_10_0
    limit_hi: 9
    limit_lo: 0
  score_2p_10k:
    coil_inc: player2_10k
    switch_0: score_2p_10k_0
    limit_hi: 9
    limit_lo: 0
  score_2p_1k:
    coil_inc: player2_1k
    switch_0: score_2p_1k_0
    limit_hi: 9
    limit_lo: 0
```

(continues on next page)

(continued from previous page)

```
  score_2p_100:
    coil_inc: player2_100
    switch_0: score_2p_100_0
    limit_hi: 9
    limit_lo: 0
  score_2p_10:
    coil_inc: player2_10
    switch_0: score_2p_10_0
    limit_hi: 9
    limit_lo: 0
score_reel_groups:
  player1:
    reels: score_1p_10k, score_1p_1k, score_1p_100, score_1p_10, None
    tags: player1
    chimes: None, chime1, chime2, chime3, None
    lights_tag: player1
  player2:
    reels: score_2p_10k, score_2p_1k, score_2p_100, score_2p_10, None
    tags: player2
    chimes: None, chime1, chime2, chime3, None
    lights_tag: player2
```

Related Events

- *reel_(name)_advanced*

Scoops / Vertical up Kickers (VUKs) / Saucer holes

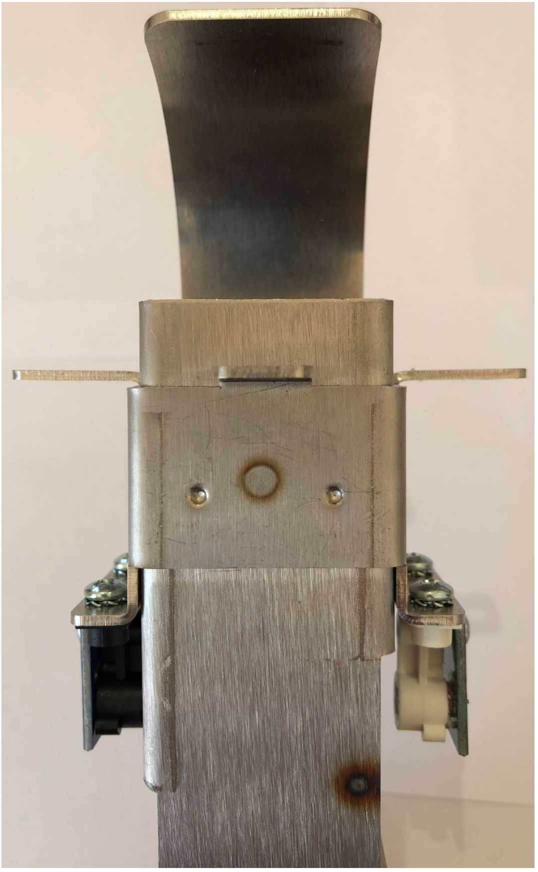
Related Config File Sections
ball_devices:
switches:
coils:

- *Electronical details*
- *Config*

Scoops usually capture balls from a playfield (sometimes via a subway) and eject them back to the playfield after a short while. Saucer holes work like scoops but the ball stays visible all the time and they are sometimes used as a lock. Similarly, vertical up kickers (VUKs) capture from the playfield but they eject onto a ramp or a upper playfield.

Electronical details

Electronically, all of those mechs consist of a switch or opto and a coil to eject the ball.




AE-23-800

Connect your switch according to *Mechanical Switches* or *How to configure opto switches* depending on its type. Then connect your coil according to *Coils (Solenoids)*.

Config

In MPF, you configure them as *ball devices* since they can count balls and choose to keep or eject it.

This is an example:

```
switches:
  s_scoop:
    number: 2
coils:
  c_scoop_eject:
    number: 4
    default_pulse_ms: 20
ball_devices:
  bd_scoop:
    ball_switches: s_scoop
    eject_coil: c_scoop_eject
    eject_timeouts: 1s
```

It is very common to delay the game when the ball is inside a scoop/VUK/saucer to show animations and play sounds. You can achieve this using a *queue_relay_player* in your mode (you might want to use *conditional events* to only trigger it when certain condition match):

```
switches:
  s_scoop:
    number: 2
coils:
  c_scoop_eject:
    number: 4
    default_pulse_ms: 20
ball_devices:
  bd_scoop:
    ball_switches: s_scoop
    eject_coil: c_scoop_eject
    eject_timeouts: 1s
##! mode: my_mode
# in your mode
queue_relay_player:
  balldevice_bd_scoop_ball_eject_attempt:
    post: start_mode_success_show
    wait_for: mode_success_show_ended
show_player:
  start_mode_success_show:
    success_show:
      loops: 0
      events_when_completed: mode_success_show_ended
shows:
  success_show:
    - duration: 10
      # add lights/sounds/slides here
```

When your mode is running the eject will be delayed by 10s (duration of your show). Add all your lights, shows and slides to this show. After the show ends it will eject normally.

The same can be achieved using a *ball_hold device*. If you want your saucer/VUK/scoop to lock a ball for a *multiball* use a *ball_lock device* instead (see *multiball* in the game design section for more details).

Servos

Related Config File Sections
servos:

A servo is device which can move to a certain position based on internal feedback. There is no need to add position switches and the servo will hold its position even if something pushes it aside. On the downside, there is no way to tell when the servo reached its position since it will not provide any position feedback to the software side.

This is an example:

```
servos:
  servo1:
```

(continues on next page)

(continued from previous page)

```
  servo_min: 0.1
  servo_max: 0.9
  positions:
    0.0: servo1_down
    0.8: servo1_up
  reset_position: 0.5
  reset_events: reset_servo1
  number: 1
servo2:
  positions:
    0.2: servo2_left
    1.0: servo2_home
  reset_position: 1.0
  reset_events: reset_servo2
  number: 2
```


Overview video about servos:

https://youtu.be/wA6KEODwQ5w

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for servos is `device.servos.<name>`.

position Value, stored in memory of what servo position should be, on a scale from 0.0 to 1.0.

Related How To guides
Programming Servo Sequences

Related Events

None

Programming Servo Sequences

Related Config File Sections
servos:
shows:
show_player:

You often want to move servos to different positions sequentially. For instance, an animated toy should open and close its mouth three times on a hit to a target. The target will post one event and you could use that to move the servo to one position. Servos do not prove position feedback so there is no way to trigger something on arrival at that position (unless you add additional switches). Instead you usually create a show which triggers a timed series of movements. The advantage of this solution is that you can easily integrate and synchonize it with sounds and lights.

The following example will move the servo six times when `my_toy_hit` is posted (three times to open and three times to close):

```
servos:
  my_toy:
    positions:
      0.0: open_mouth
      1.0: close_mouth
    reset_position: 1.0
    number: 1
shows:
  toy_hit:
    - duration: 1s
      events: open_mouth
    - duration: 2s
      events: close_mouth
show_player:
  my_toy_hit:
    toy_hit:
      loops: 2
```

To see how this can be used in a real machine we recommend this explanation video by the pinball amigos:

https://youtu.be/1QOOJNtsGxw

You might also enjoy our video about servos in general:

https://youtu.be/wA6KEODwQ5w

Shakers

Related Config File Sections
coils:
coil_player:

Shaker motors cause vibrations to give the player tactile feedback.

Hardware

TODO: Add a picture of a shaker

Todo: *Help us to write it*

Part numbers:

- Spooky: #100-0054-00
- Stern Spike: #502-5027-01

- Stern SAM: #502-5027-00
- Data East/Sega/Stern: #515-5893-01

Most shaker motors are not meant to be enabled without PWM. Depending on the voltage your PWM should have a duty cycle between 10% and 30%.

Config

This is an example on how to use a shaker using coil_player:

```
coils:
  c_shaker:
    number:
    default_pulse_ms: 1
    default_hold_power: 0.125    # keep this low

##! mode: your_mode
coil_player:
  enable_shaker_event:
    c_shaker: enable
  disable_shaker_event:
    c_shaker: disable
```

Alternatively, you can use it inside a show:

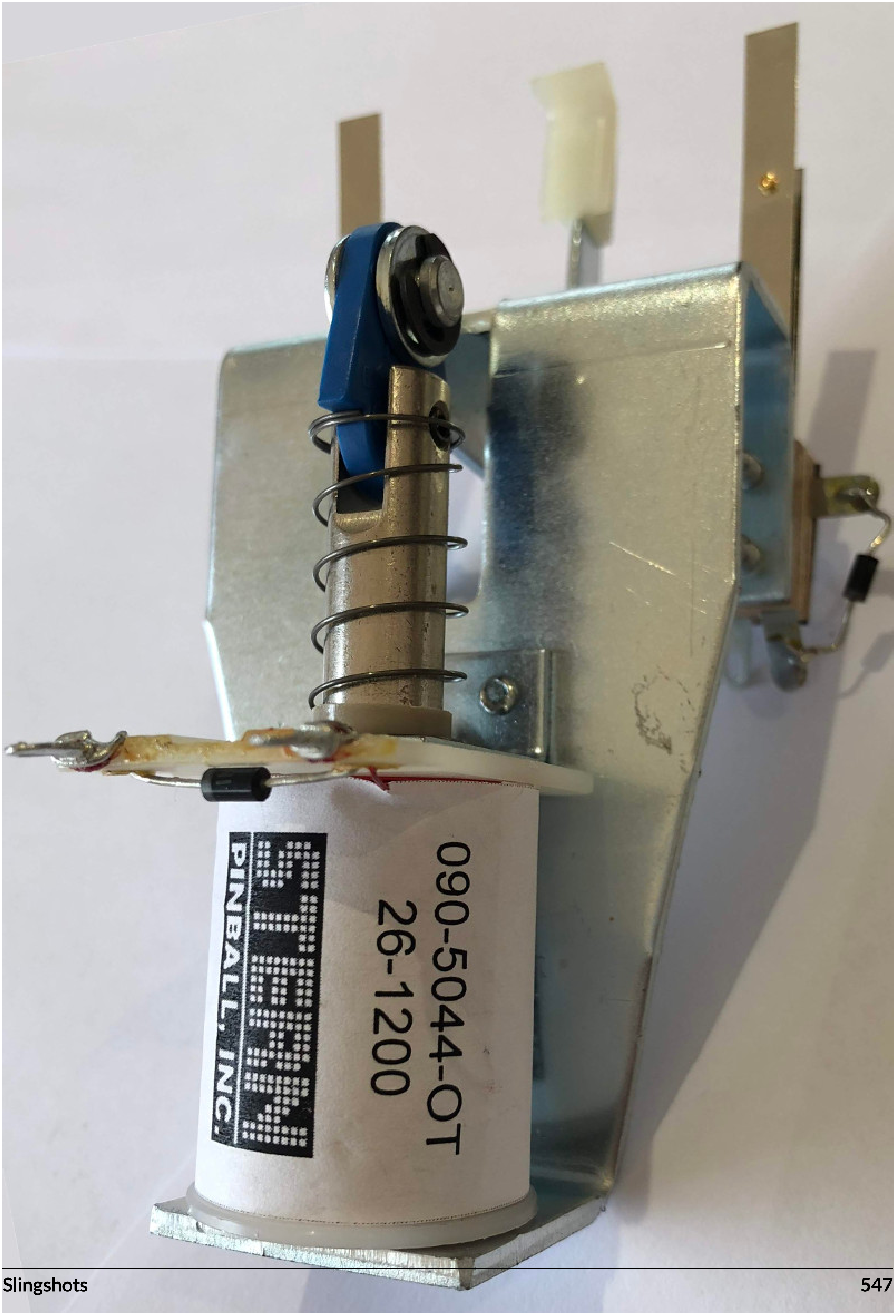
```
coils:
  c_shaker:
    number:
    default_pulse_ms: 1
    default_hold_power: 0.125    # keep this low

##! mode: your_mode
shows:
  my_show_with_shaker:
    - duration: 1s
      coils:
        c_shaker: enable
      # add some slides, lights or sounds here
    - duration: 1s
      coils:
        c_shaker: disable
      # add some more slides, lights or sounds here

show_player:
  play_show_with_shaker:
    my_show_with_shaker:
      loops: -1
```

Slingshots

Related Config File Sections
autofire_coils:

Slingshots are configured as *autofire_coils* in MPF.

Hardware


090-5044-OT
26-1200
STERN
PINBALL, INC.

A sling shot usually consists of two *blade switches* and one *coil*. Those switches are wired in parallel because it does not matter which switch was closed to fire to slingshot. Connect one side of each switch to ground and the other side of both switches to the same input.

Part numbers:

- Data East/Sega/Stern: #500-5849-00
- Spooky/American Pinball/Suncoast: PBL-5849-01

Config

This is an example:

```
switches:
  s_sling_left:
    number: 5
coils:
  c_sling_left:
    number: 7
    default_pulse_ms: 15
autofire_coils:
  ac_slingshot_left:
    coil: c_sling_left
    switch: s_sling_left
```

Adjust `default_pulse_ms` and `default_pulse_power` in your coil to control the strength and sound of your slingshots.

Spinners

Related Config File Sections
switches:
spinners:

Spinners are rotating metal plates which close a switch once per rotation.

Hardware

Part numbers:

- Stern: #511-5113-00 or #100-0014-00

Config

In MPF spinners are configured as follows:

```
switches:
  s_my_spinner:
    number: 42    # number depends on your platform

spinners:
  basic_spinner:
    switch: s_my_spinner
    active_ms: 500
```

It is very common to count the rotations of your spinner per player. You can either use a player variable or a counter for that. This is an example:

```
switches:
  s_my_spinner:
```

(continues on next page)

(continued from previous page)

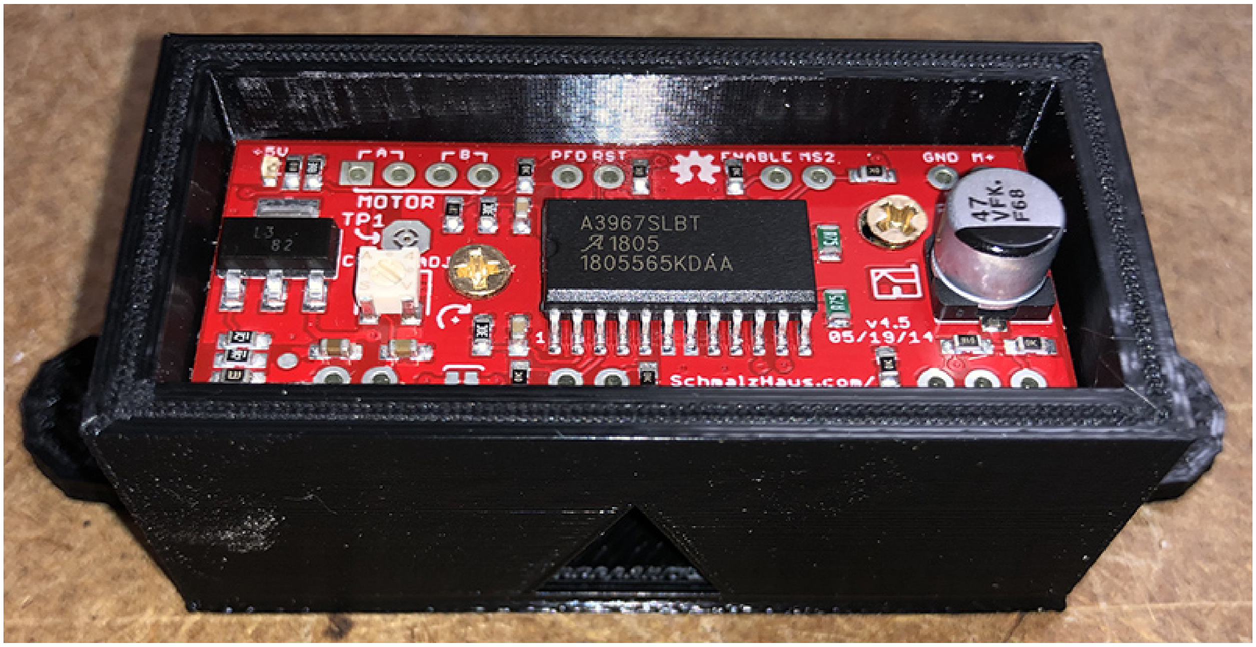
```
    number: 42    # number depends on your platform

spinners:
  basic_spinner:
    switch: s_my_spinner
    active_ms: 500
##! mode: my_mode
# in your base mode add 1 for every rotation to a player variable which you can use in slides
variable_player:
  s_my_spinner_active:
    spinner_rotations: 1
# in a game mode the player needs to spin the spinner 10 times
counters:
  spinner_rotations:
    count_events: spinner_basic_spinner_hit
    count_complete_value: 10
    events_when_complete: mode_finished
```

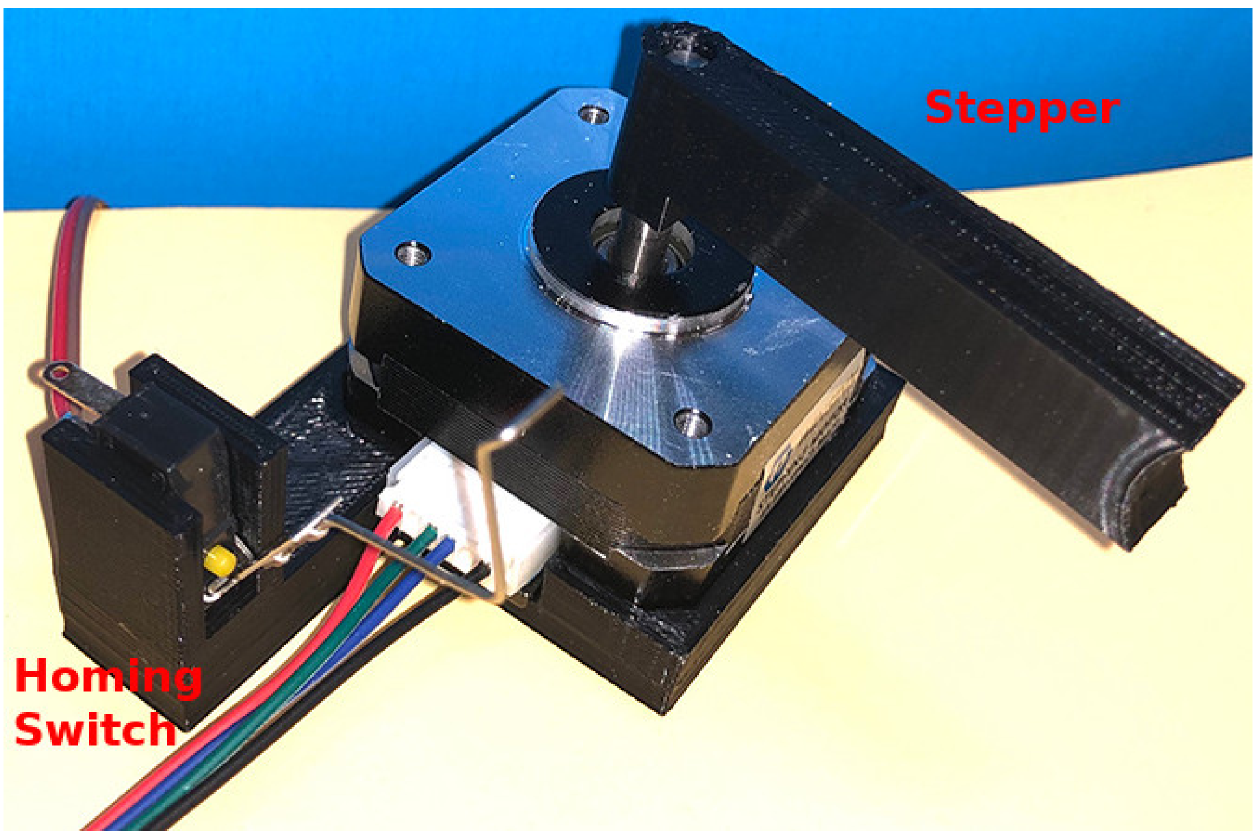
Related Events

- *spinner_(name)_hit*
- *spinner_(name)_inactive*
- *spinner_(name)_idle*
- *spinner_(name)_active*
- *spinner_(name)_(label)_hit*
- *spinner_(name)_(label)_active*

Stepper Motors

Related Config File Sections
steppers:





Stepper motors offer digitally controlled precise movement of mechanisms. They require a separate driver board that interfaces with the host computer by USB or through the pinball machine controller. Steppers have a unique design with two or more sets of coils which when energized sequentially turn the armature a set distance, typically 1.8 degrees.

Overview video about steppers:

https://youtu.be/YaRNBU0OHGc

Steppers vs Servos

It is useful to compare stepper motors to servo motors. While in many cases they can be used interchangeably, each has advantages and disadvantages. The principle advantage of steppers is precision. If used within their torque window, steppers can reproducibly count thousand of steps, reverse them, and land back at the starting position. Generally steppers are faster than servo motors which transmit torque through a gear assembly. Disadvantages of steppers include less torque than offered by servo motors and requiring a driver controller. Also, unlike servos, steppers do not include a feedback mechanism to report the rotational angle of the armature. This deficit requires that a stepper use a homing mechanism (typically a switch) to inform software when the assembly is at an extreme of linear or rotational position. Lastly, steppers are subject to rotational drift when not energized, whereas servos maintain position in their off state.

Stepper controller boards require a minimum of two digital inputs, one for rotational direction and one to trigger a rotational step. Usually one or more additional inputs are also used to control the power state of the driver board and/or motor coils. Some driver boards also allow programming of microstepping to command rotation at less than that of a full step.

MPF abstracts the nitty gritty of stepper control allowing steppers to be used with a minimum of YAML programming. On startup, an event is issued to rotate the motor to a home position. Once homed, further events can be issued which rotate the motor an arbitriary number of steps in either direction as required by the application.

See *Servos* for more details.

Example config

```
#config_version=5
switches:
  s_home:
    number:
steppers:
  stepper1:
    number: 1              # depends on your hardware
    homing_mode: switch
    homing_switch: s_home
    named_positions:
      10: move_to_position_1
      20: move_to_position_2
      50: move_to_position_3
```

When you post `move_to_position_1` the stepper will move to the position `10`. Similarly, it will move to `20` when you post `move_to_position_2` and to `50` when `move_to_position_3` is posted. It will track its current position internally.

Switches

Related Config File Sections
switches:
switch_overwrites:

MPF's *switch* device represents a switch in a pinball machine. This device is used for switches, including cabinet buttons, rollovers, targets, optos, trough switches, DIP switches, etc.

There are two switch types most commonly seen in pinball machines (read those for details):

- *Mechanical switches*
- *Optical switches*

And an additional two types used in a handful of machines (read those for details):

- *Proximity switches*
- *Reed switches*

Typical switch applications in pinball machines are:

- *Rollover/lane switches*
- *Standup targets*
- *Spinners*
- *Flipper buttons* and *Flipper end-of-stroke switches*
- As part of a mech such as *Drop targets*, *Popbumpers* or *Ball Devices*/*Troughs*
- *Service and door switches*

MPF supports all types of switches found in all generations of pinball machines, including matrix switches, direct switches, Fliptronics switches, switches connected to I/O boards, etc.

Switches only have two states: *active* and *inactive*. (We don't say "open" or "closed" because sometimes switches are normally-closed which mean they're actually active when they're open.) In MPF, you configure your switches in the `switches:` section of your machine configuration file, including options (like whether the switch is "active" when it's in the open state or the closed state.)

You can also configure *debounce settings* for each switch, which controls how MPF responds to switch events. Saying that a switch has to be "debounced" means that the pinball controller makes sure the switch is actually in its current state for a few milliseconds before it send the switch event to MPF. This can be useful to filter out unwanted or phantom switch events which might happen due to electrical interference or other little weird things.

Most switches in pinball machines are debounced except for the ones that you absolutely want to fire instantly, like flipper switches and the switches attached to automatically fired coils like slingshots and pop bumpers.

This is an example:

```
switches:
  my_switch:
    number: 42    # number from your hardware platform
```

Switch Concepts

- *Debouncing in Pinball Machines*
- *Switch Controller*

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for switches is `device.switches.<name>`.

state Numeric value which represents the logic state of this switch. 0 is inactive, 1 is active.

recycle_jitter_count How many times this switch has activated within it's configured `ignore_window_ms:`. (These activations are ignored.)

Related How To guides

- *Tutorial step 3: Get flipping!*
- *How to configure opto switches*
- *Mechanical Switches*
- *Rollover Switches*
- *Service and Door Switches*
- *Start, Tournament and Launcher Buttons*

Related Events

- *(name)_active*
- *(name)_inactive*
- *(name)_active*
- *(name)_inactive*
- *sw_(tag)*
- *sw_(tag)_active*
- *sw_(tag)_inactive*
- *switch_(name)_active*

- *switch_(name)_inactive*

Debouncing in Pinball Machines

A pinball machine is a mechanical machine with a lot mechanical, electronic and electromagnetical interferences. This has to be mitigated on multiple levels to prevent unwanted effects:

1. Prevent too much communication between hardware platform and CPU. A lot of switch changes could easily overflow the communication bus or starve the CPU/controller.
2. Prevent too many switch events in the game. It is not uncommon to show slides or play sounds on a switch event. If this event occurs very often this may easily slow down your game.
3. Prevent coils from pulsing too often. If a coil pulses on a switch hit and the switch activates constantly it might essentially be stuck on for the whole time which in the best case would only blow a fuse and in the worst case might burn down the machine.

As you can see there are multiple types of debouncing. We will explain how to use those in the following:

Switch Debouncing at the Hardware Level

To prevent too much communication between you hardware platform and your CPU there is typically some switch debouncing at the hardware level. This is what most electronic engineers will first think about when taking about *debouncing*.

On the surface, switch *debounce* is pretty straightforward. Switches are mechanical things, computers are fast, and your pinball software wants to make sure a switch is actually in a new state before acting on a switch.

Pinball controllers set debounce in different ways. For example, some platforms (for instance, P-ROC, P3-ROC) say "a switch must be in a new state for 2 consecutive reads" to be considered debounced, while other platforms (e.g. FAST) focus on time-based durations rather than number of reads, saying, "a switch must be in a new state for X milliseconds before it's considered debounced." In practise, there is not much difference between those two.

When considering switch debounce, the switch usually is supposed to be active for the whole debounce time. So this could also be called "minimum active time". Usually this time is in the range of two to four milliseconds. The reason for that is that waiting for a minimum active time induces some lag to the switch event.

Still, switch debounce is often disabled for hardware rules (e.g. for pop bumpers or sling shots) to render them more responsive. However, this also them more susceptible to interferences or phantom hits. For that reason, in some platforms, even in that case a minimal debounce time is enforced (around one millisecond).

There is very little reason to increase switch debounce time to more than about four ms (see next section on what to do instead). Because if you set your debounce times too long, then you risk switch events being missed. (It would be annoying if a ball brushed a pop bumper and the bumper not didn't fire.)

By default, MPF will enable switch debounce in all *switches*. For autofires such as pop bumpers or sling shots it will be disabled. You can overwrite this using the `debounce` setting in your *switches*.

Preventing too many Switch Events in MPF

Depending on the type of switch you will see hits between five and fifty milliseconds. So any switch debounce time above that will miss switch hits. However, if you set your debounces too short, you risk getting multiple switch events for what should have been a single switch event. (Again it would be annoying if a ball hit a pop bumper and that bumper fired once, but you actually got back multiple switch events which led to multiple scores, multiple sound effects, etc.)

The solution to this is to combine switch debounce with a window to ignore multiple hits. There are two ways to implement this.

Ignore Window

The first and most used way is to define a period after registered hit which ignores all further hits. This setting is called `ignore_window_ms` in your *switch config*. For example, if you set `ignore_window_ms: 100`, then a switch is activated once, then again 50ms later, the second activation will be ignored. The timer is set based on the last switch hit that *activated* the switch, so if another switch hit came in 105ms after the first (which would be 55ms after the second), it will also count.

In most cases you can easily set `ignore_window_ms` to a few hundred milliseconds. This will not affect hardware rules. Use `recycle` on your *coil* instead.

This is what most javascript programmers understand when they hear debouncing. Kind of related but also a bit different from what EEs understand by it.

Throttling

There is another technique which is commonly used in the javascript works when working with computationally expensive callbacks which is called throttling. The goal here is similar but the implementation is differerent. Instead of having a window after each activation this defines a maximum number of calls per time unit. For instance a maximum of 10 calls per second. This would certainly also be possible in MPF but is currently not supported. We think this would be inferior to `ignore_window_ms` since it is more susceptible to bursts it might still cause temporary lags. However, we might add this later to prevent permanent problems with bad or bouncy switches.

Preventing Coil Overheating

When enabling coils you usually use PWM to control the maximum power. However, when pulsing coils they are often enabled without any PWM for a while. This works fine for a single activation but might cause problems when a switch is activated repeatedly (i.e. because of interferences). In that case, the coil would be permanently pulsed and, thereby, enabled all the time. That will hopefully only blow a fuse on that coil but might as well burn down the machine. To prevent this there is `recycle` on your *coil*. When set to true it will prevent any further pulse for a certain time after a pulse (similar to `ignore_window_ms` on the switch above). The duration depends on your platform and might also be configurable.

Understanding switch scanning loop speed

The other major factor which affects debounce involves the timing of how the switches are read.

In all modern pinball platforms, a switch changing state doesn't interrupt the controller. Instead, the controller reads the state of all switches at a certain interval.

But even this varies from platform-to-platform, and even based on whether you have matrix or direct switches. (More on this in a bit.)

The important thing, though, is that different controllers and different types of switches are checked at different intervals. That could be every millisecond, or every 1ms, or every 2ms. . . really it's up to the controller and switch type as they're all different. Scanning speed induces some delay and jitter to your debounce times. Refer to your platform documentation for details.

In most cases switch matrixes are scanned slighly slower than direct switches on a hardware platform. However, they are usually still fast enough not to cause any problems with missed switches.

Switch Controller

The *Switch Controller* is responsible for receiving all hardware switch state changes and translating them into MPF events which are broadcast out to all the other game modules. In other words, the switch controller is the only part of the game that actually receives notification of the physical switches-—it's the only thing that "talks to" the switch hardware. Everything else in the game just waits for the switch controller to tell it that a switch action happened, rather than all different parts of the game all talking to hardware.

Why do we force everything to talk to the switch controller instead of letting individual modules talk to the switches directly? Lots of reasons:

- The switch controller has the intelligence to know whether a switch is normally open (NO) or normally closed (NC), based on how each switch is configured in the machine configuration files. This means that all the game modules only have to listen for the *switch active* and *switch inactive* events, rather than each module needing the intelligence to transpose the switch states as needed.
- The switch controller can change the timing of switches, even applying software delays and debouncing to switches, and this is all hidden from the other MPF modules.
- The switch controller can "hide" physical switch activities from the game. This is most useful for broken switches that are firing like crazy. If the switch controller notices that a switch is going nuts, it can suppress those events, slow them down, or just ignore them altogether. That way you can just write your game code to say something like "when this switch is active, assign these points" and you don't have to worry about a bad switch giving all your players high scores! (This functionality is not yet complete)
- The switch controller can also reprogram the game logic around broken switches. So if it knows that a switch is broken, it can send the game switch events for the broken switch when some alternate switch is hit. This means that each of your game modules can automatically get the benefit of this intelligent switch substitution without you having to write anything special. (Again, how this substitution takes place and which switches can be substituted for others is all configurable in your config files.)
- Since the switch controller is the only interface into the game for switches, it can "inject" switch events from any source. For example, MPF includes functionality to simulate switch events with a computer keyboard (for testing and debugging), as well as switch events from a mobile phone or table. We also have a plug-in to read and playback switch events from log files from games that already ran, as well as the ability to write scripts that simulate games. All this is done by interfacing to the switch controller—-your actual game code doesn't know (or care) where the original switch events came from.

How to configure opto switches

Related Config File Sections
switches:

Optical switches (short optos) are common in pinball machines. They usually cover ranges up to 10cm and are used in places where normal *roll over switches* cannot be used (e.g. because a lane is too wide or on a ramp). Optos are also commonly used in ball troughs.

Electronical details

Electronically they consist of a sender and a receiver. The sender is usually connected to 5-12V power with a current limiting resistor in line (to limit the current to about 50-70 mA). Alternatively, a constant current driver may be used (more expensive but better for the sender). The receiver is usually connected to a direct input on a switch board (they cannot be used in a switch matrix without further logic PCBs). Most direct inputs have an internal pull up which will pull the level to VCC (usually around 10 kOhm). The opto receiver will then pull the current to GND when it receives light from the sender. Once the light beam is interrupted the receiver will stop conducting and the input will go up to VCC again. For this reason, the input will be closed when the beam is not interrupted and open when the opto is interrupted. This is exactly inverse than a normal switch and you have to configure your opto as *normally closed* (short `NC`) for that reason. If your opto is using an additional PCB for optos it might invert the signal and revert this effect (just try normally closed first and change it later - it will not break anything).

TODO: Add electronical drawing for sender and receiver.

Video about wiring optos:

https://youtu.be/7G8LFBbYNOY

Brightness and Current

The brightness of IR diodes (and diodes in general) depends only on the current flowing through the diode. Usually IR diodes (i.e. the famous QED123) drop about 1.7V forward voltage. However, this is not a constant and will fluctuate depending on manufacturer tolerances. That means that at about 1.7V current will start to flow through the diode and it will emit (IR) light. Unfortunately, just connecting it to 1.7V is not sufficient because the current is non-linear for LEDs. Below the forward voltage (i.e. 1.7V) the currency is 0 and above the forward voltage it increases exponentially. According to the specifications most IR LEDs should use run at 20mA. However, in pinball machines 50-70mA is used because it allows larger distances. Unfortunately, that is also the reason why transmitters often break in pinball machines because more current makes diodes age faster. 100mA is the absolute maximum rating for most LEDs which means that the part is not indented to be operated at this current for prolonged periods. Since the current increases exponentially above the forward voltage, the forward voltage has huge tolerances during manufacturing and power supplies have tolerances, too, it is absolutely necessary to limit the current instead of regulating the voltage. There are two approaches which will be described in the following.

Constant Current Drivers

Technically, the best solution to drive a IR sender is a constant current driver. However, it is quite costly compared to the next solution. In some cases a constant current driver might be embedded on your optos (see below at the Stern Spike optos). In addition, there are opto driver boards which contain constant current drivers. The main advantage of constant current drivers is that they are not affected by any fluctuation in the supply voltages or manufacturing tolerances/aging of the IR diode. Expect significant higher lifetimes of your transmitters with those drivers.

Common parts:

- Stern Spike Trough Boards
- FAST 4-Channel 12v Constant Current Opto Emitter Driver
- Stern Spike Opto Amplifier - 520-5239-01

Have a look at our PCB section of hardware.missionpinball.org for DIY designs. Have a look at *Switch/Opto Breakout Boards* for details about breakout boards.

Current Limiting Resistor

A very cheap and common solution is to use a resistor in line with your transmitter to limit the current. In practice, this will result in varying brightnesses depending on manufacturing tolerances of the resistor (10%) and the diode (unknown but high). Additionally, changes in the supply voltage will also affect the brightness. For this reason, it is wise to design your resistor a bit lower to account for some tolerances.

If your supply voltage is 5V you probably want a 56ohm or 68ohm resistor at 1/2watt in line with your sender (assuming a forward voltage of 1.7V and 50mA forward current). For 12V you need a 220ohm at 1 watt which will get very hot (do not use a standard 1/2watt resistor).

Common parts:

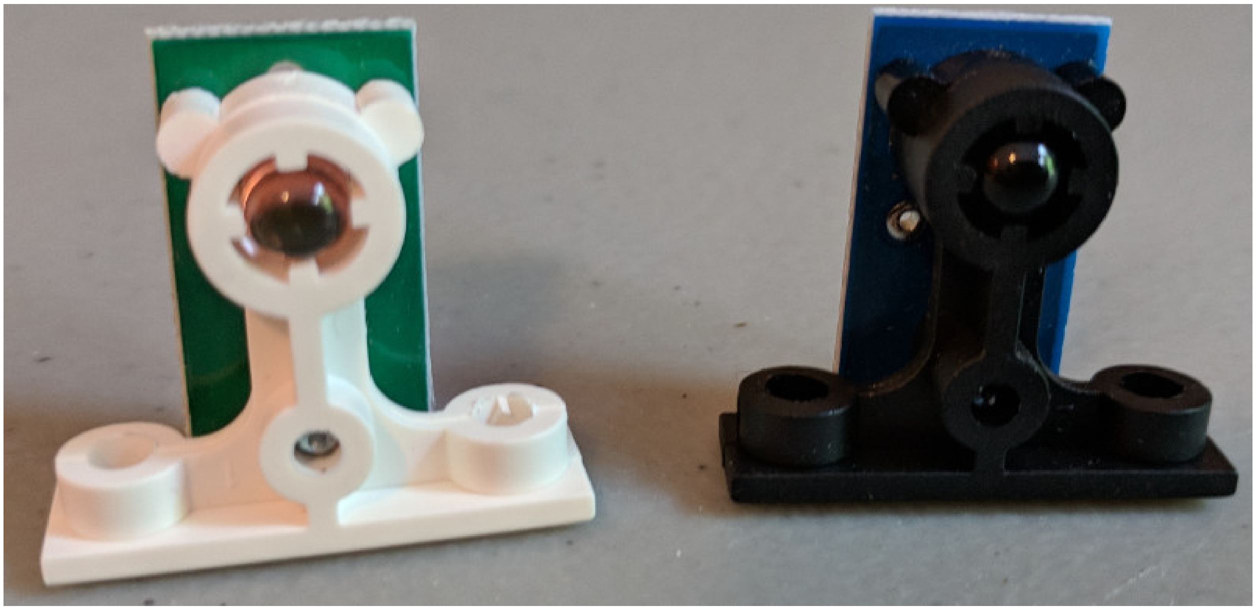
- 4x Opto board - #600-0256-00

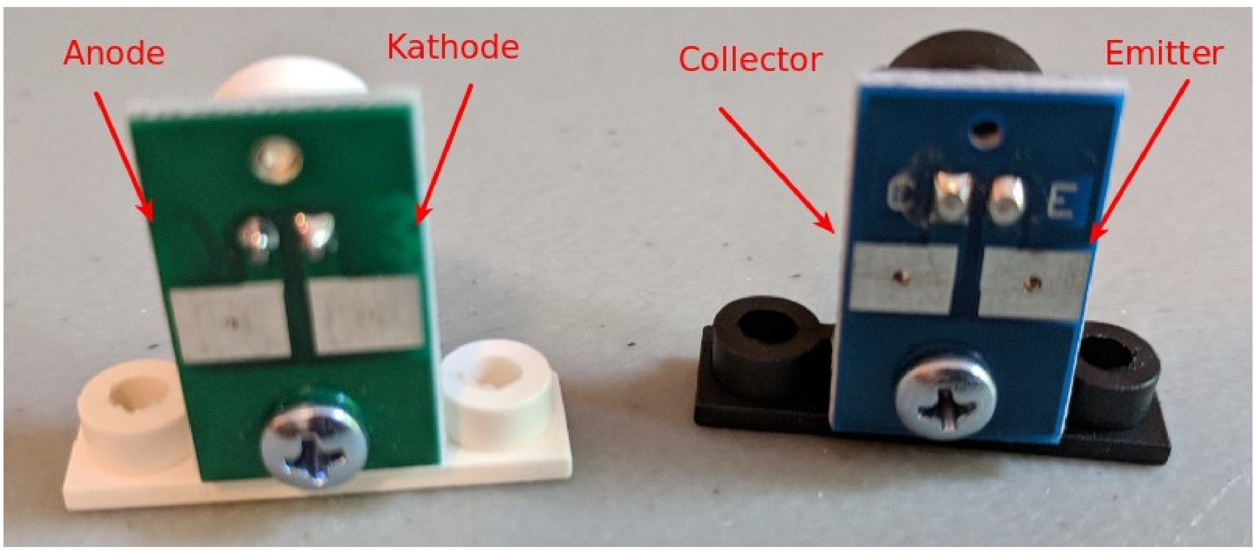
Have a look at our PCB section of hardware.missionpinball.org for DIY designs. Have a look at *Switch/Opto Breakout Boards* for details about breakout boards.

Common Parts in Pinball Machines

In the following we will describe some common parts and how to connect them.

Williams/Bally Optos





In most platforms with direct inputs you can directly connect a receiver to an input. You connect the collector to the input (`C`) and the emitter (`E`) to ground. Consult the documentation of your hardware platform for details.

For the transmitter connect the kathode (`K`) to ground and the anode (`A`) to a current limiting resistor. Connect the resistor to power. DO NOT omit the resistor to power without any current limiting or it will break/burn.

Part numbers:

- Transmitter: A-16908 or A-14231
- Receiver: A-16909 or A-14232

Diodes used (in case you need to replace them):

- Transmitter: QED123

- Receiver: QSD124 or QSD124A4R0 (Pinball part numbers: 5163-14114-00 or 5163-12732-00)

Data East/Sega/older Stern Optos

TODO: Add a picture of those transceivers

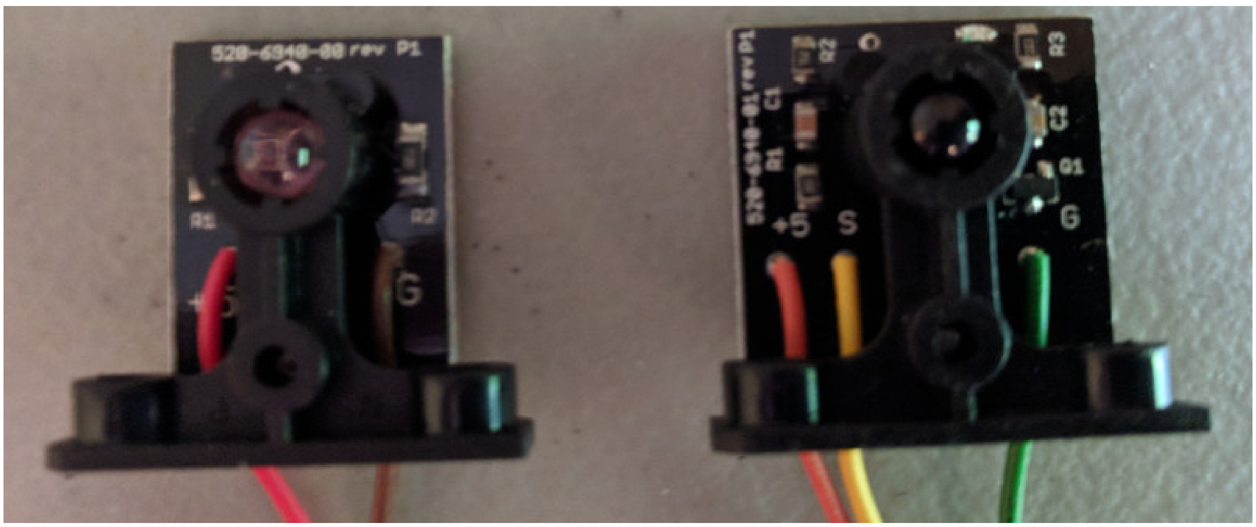
Data East/Sega and later Stern used a diode which can serve as either transmitter or receiver called "transceiver". The advantage of this solution is that you only need one type of parts. Electronically they work similar to Williams/Bally optos.

Part numbers:

- Transceiver: 500-6775-00/500-6775-01 or 500-6747-00

Stern Spike Optos

Labels on Stern Spike optos looks different but they work similarly:



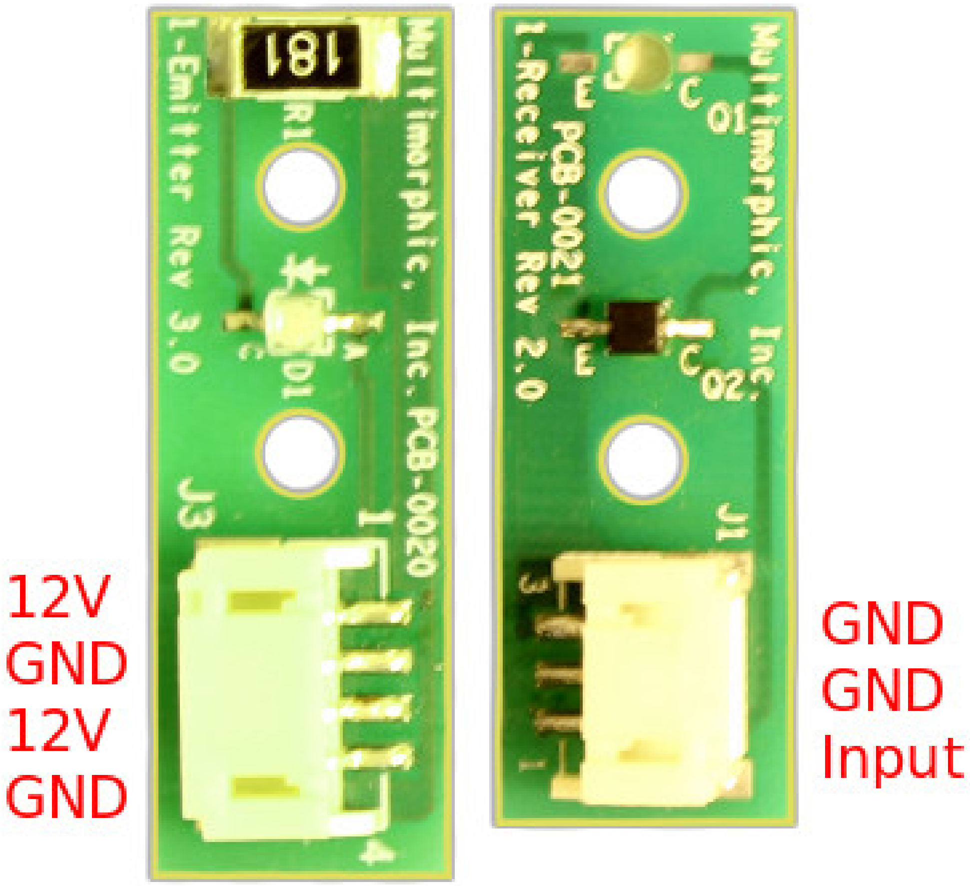
On the transmitter (left) connect `+5` to 5V and `G` to GND. A current limiting resistor is not required since it is embedded on the sender.

The receiver also connects `+5` to 5V and `G` to GND. Additionally, connect signal `S` to your input.

Part numbers:

- Transmitter: 520-6940-00/515-0215-00
- Receiver: 520-6940-01/515-0215-01

Multimorphic Optos:



Multimorphic produces and sells optos with a JST connector. The transmitter contains a current limiting resistor for 12V (you only have to connect one of the 12V and GND pins). You don't need an additional resistor but you are also bound to 12V. They might work at 5V but the range will be much lower. Though the surface mount resistor on the transmitter board is designed to run "hot," it still requires a surrounding air gap to dissipate heat. 3D printed parts mounted against this resistor will melt.

Part numbers:

- Transmitter: PCBA-0019-EO03, PCBA-0019-EI03, PCBA-0020-CI03, PCBA-0020-CO03
- Receiver: PCBA-0021-EI03, PCBA-0021-CI03, PCBA-0021-EO03, PCBA-0021-CO03

Config

You can configure a normally closed opto like this:

```
switches:
  trough1:
    number: 81    # number depends on your platform
    type: 'NC'    # normally closed
  orbit_opto:
    number: 23    # number depends on your platform
    type: 'NC'    # normally closed
```

See *switches:* for details about the config options.

Mechanical Switches

Related Config File Sections
switches:

Most switches in pinball machines are mechanical switches which are open by default and close a circuit when pushed.

Video about wiring switches:

https://youtu.be/7G8LFBbYNOY

There are two common types of mechanical switches:

Leaf switches/Blade switches

First, blade switches which are very cheap and reliable but cannot be used everywhere:


Ground
Input

Typically, those are use for *flipper buttons* and *flipper end of stroke switches*.

Part numbers:

- Stern Flipper Leaf Switch: 500-6889-01 or 500-6890-01
- Data East/Sega Flipper Leaf Switch: 180-5122-00
- Williams/Bally Flipper Leaf Switch: SW-10A-48 or SW-1010A-13
- Data East End of Stroke Switch: 180-5018-00
- Williams/Classic Stern/Bally End of Stroke Switch: SW-10A-50, ASW-A20-23, SW-1A-193

Additionally, those are used for *targets*:

Part numbers (Data East/Sega/Stern):

- 515-5966-xx
- 500-5835-xx
- 515-5124-xx
- 500-5232-xx
- 515-5162-xx
- 515-5967-xx
- 515-6027-xx

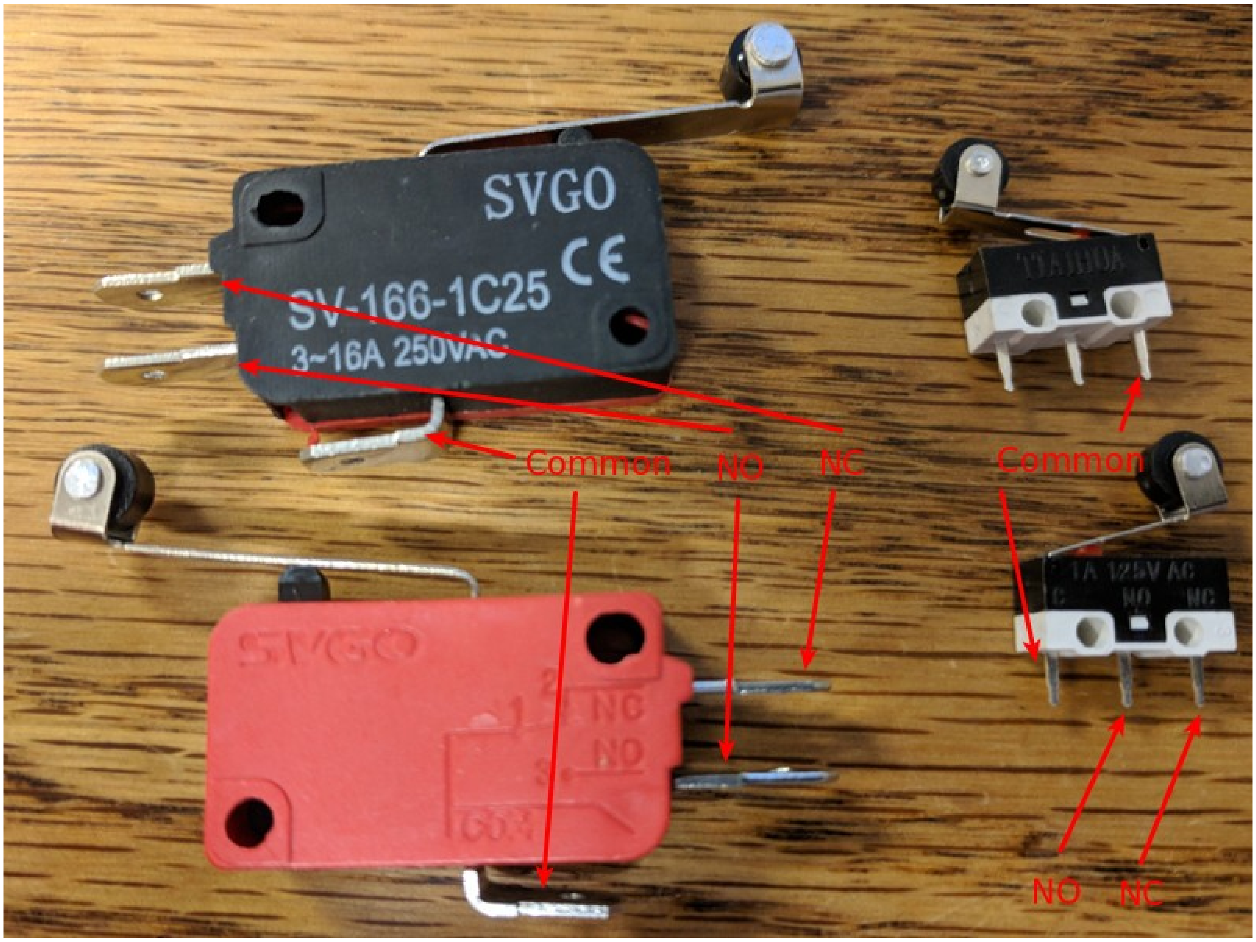
xx defines the color of the target in most cases.

Micro switches

Second, micro switches which are very small and commonly used for roll over switches. Those usually have three connectors:

- `C` - common pin for `NO` and `NC`
- `NO` - normally open - connected to `C` only when the switch is pressed
- `NC` - normally closed - connected to `C` only when the switch is not pressed

Usually, you connect `C` to ground and `NO` to your direct input (see below for switch matrices).



Electronically and logically both switches work similarly.

Part numbers (Data East/Sega/Stern):

- 180-5010-xx
- 180-5053-xx
- 180-5119-xx
- 180-5118-xx
- 180-5052-xx
- 180-5186-xx
- 180-5057-xx
- 500-5442-xx
- 180-5175-xx

xx defines the shape of the blade for most parts.

Direct inputs

Switches can be connected to a direct input and ground on almost all platforms. Most direct inputs have an internal pull up which will pull the level to VCC (usually around 10 kOhm). When pushed the switch will pull the input to ground which will be detected as a closed switch by the platform.

TODO: Add electronical drawing for switch on direct input.

Switch matrix

Additionally, you can use switches in a switch matrix. In a switch matrix columns are connected to drivers and rows to switches. Columns are then pulsed sequentially and the rows are read. Each switch has to use a diode to prevent closing other columns.

TODO: Add electronical drawing for switch in matrix.

Switch matrices are driven using your hardware platform and MPF will read the values from the platform. Usually the numbers for switches reflect their row and column in the matrix. Consult your hardware platform documentation for details.

MPF Config

This is an example of switches in MPF:

```
switches:
  my_direct_switch:
    number: 23        # number depends on your platform
  my_matrix_switch_row_1_column_3:
    number: 1/3       # number depends on your platform
```

Service and Door Switches

Related Config File Sections
switches:

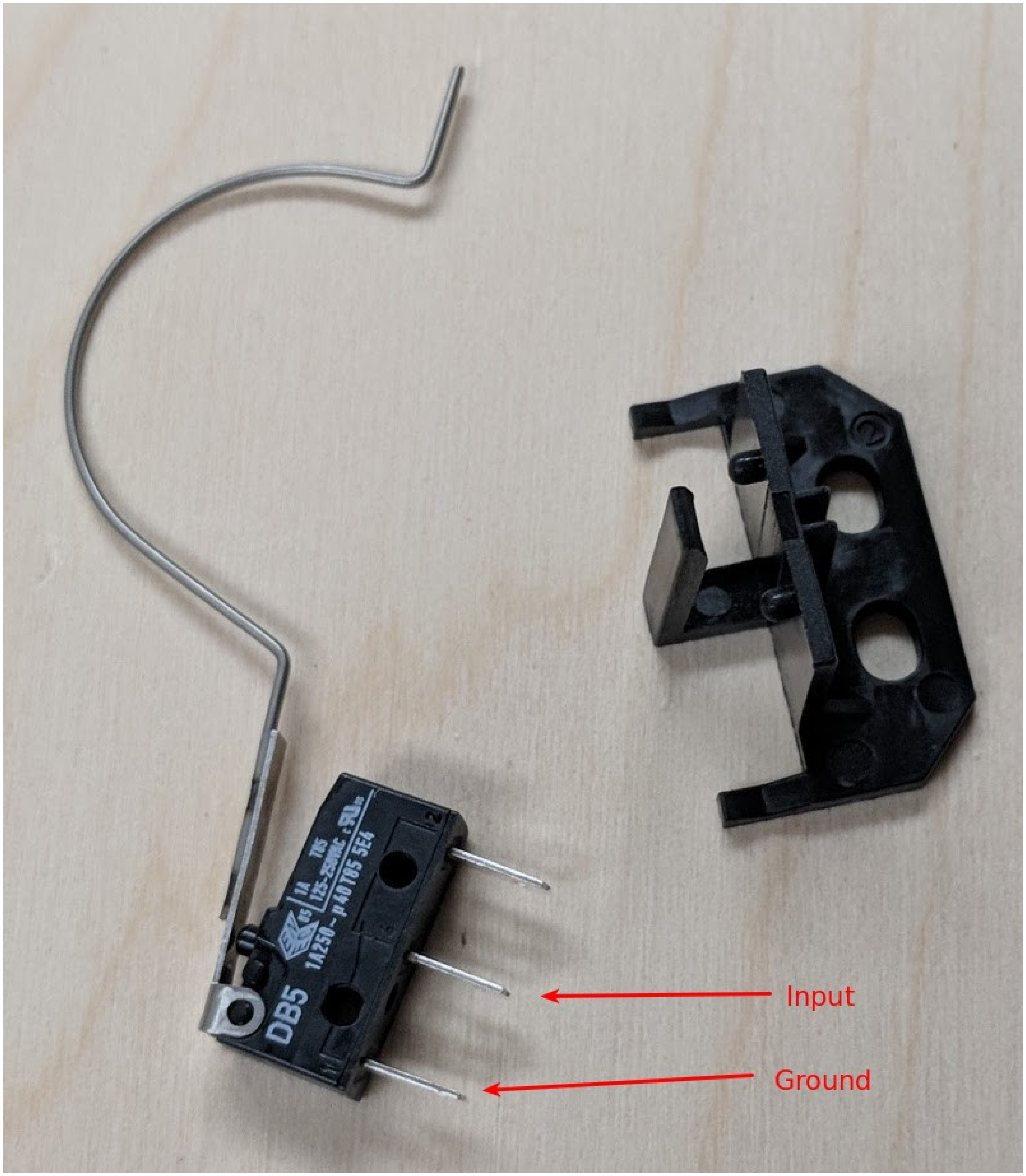
Most pinball machines have service switches inside their service door. Additionally, there is usually a switch to detect if the door is open.

You can configure those to control your *service mode*.

Rollover Switches

Related Config File Sections
switches:

Rollover switches in MPF are configured as normal switches. Furthermore, they are often paired with a light (below an insert) which qualifies them as candidate for a shots in MPF. They are usually *mechanical micro switches*.



Typical part numbers:

- Stern/Sega/Data East: 500-6227-01/500-6227-03 or 500-6227-02/500-6227-04 or 500-5707-00
- Spooky Pinball: SP-SW-001 or SP-SW-002

This is an example config:

```
# this is in your machine-wide config
switches:
  s_outlane_left:
    number: 0
  s_inlane_left:
    number: 1
  s_inlane_right:
    number: 6
  s_outlane_right:
    number: 7
lights:
  l_outlane_left:
    number: 0
  l_inlane_left:
    number: 1
  l_inlane_right:
    number: 6
  l_outlane_right:
    number: 7
##! mode: my_mode
# put this into a mode
shots:
  shot_outlane_left:
    switches: s_outlane_left
    show_tokens:
      leds: l_outlane_left
  shot_inlane_left:
    switches: s_inlane_left
    show_tokens:
      leds: l_inlane_left
  shot_inlane_right:
    switches: s_inlane_right
    show_tokens:
      leds: l_inlane_right
  shot_outlane_right:
    switches: s_outlane_right
    show_tokens:
      leds: l_outlane_right
shot_groups:
  sg_lanes:
    shots: shot_outlane_left, shot_inlane_left, shot_inlane_right, shot_outlane_right
    rotate_left_events: s_flipper_left_active
    rotate_right_events: s_flipper_right_active
    reset_events:
      sg_lanes_lit_complete: 1s
```

We configure four lane rollover switches (and their corresponding lights). Then inside a mode we define one shot for each group and a shot_group which enables rotation of the shots using the flipper buttons.

Start, Tournament and Launcher Buttons

Related Config File Sections
switches:
lights:

Probably all pinball machines have a start button which will start the *game* once you press it and there are enough *credits*. Furthermore, machines have either a mechanical plunger or a launcher button which will *shoot the ball from the launcher*. Additionally, some machines have tournament buttons to start a tournament.

Hardware

Those buttons usually come with a *micro switch* and a *#555 bulb*. You can connect the switches to any direct input on your controller or put them into your switch matrix (with an additional diode). The LED is rated at 6.3V which works fine at either 5V or in a lamp matrix at 12V (the latter commonly used).

Config

To configure your start button you can use this config:

```
lights:
  l_start_button:
    number: 3           # number depends on your platform
    subtype: matrix     # depends on your platform
switches:
  s_start:
    number: 23          # number depends on your platform
    tags: start
```

The tag `start` will hook the button into your game. See *Tutorial step 9. Add the start button* for details. You might want to integrate the button into your attract light show.

Related How To Guides
Tutorial step 9. Add the start button
Mechanical Switches
Matrix Lights (Bulbs)

Switch/Opto Breakout Boards

Normally, eight switches are connected to one bank of a pinball controller (in almost all platforms). Ground is then chained from one switch to the next to simplify wiring (as only one connector with one ground pin is required on the board). This works well for all sub-playfield switches. However, it would be tricky for optos or switches in ramps above the playfield.

To solve this breakout boards are used which connect to the bank and provide a separate connector per switch. If you are building a homebrew game there are a few designs around which can be build in China for a few bucks (just ask in our forum). Some of those boards also provide power for *optos*.

Stern occasionally also uses breakouts for optos (usually for two optos). In Stern Spike they added a few “breakout” connectors to some of their node boards to add optos and above playfield features without additional breakouts.

Common parts:

- PBL-600-0385-00 - Optos Breakout Board for 8 optos (emitter + receiver) intended to be used with Multimorphic SW-16. (Should also work with FAST hardware)
- FAST - FAST 4-Channel 12v Constant Current Opto Emitter (4 emitter)
- Stern Spike Opto Amplifier - 520-5239-01 (2 emitter + receiver)
- Multimorphic PCBA-0018-0002 - One JST connector per switch and also distributes power to optos.

Have a look at the PCB section of hardware.missionpinball.org for DIY designs.

See also *How to configure opto switches* for how this works technically.

Proximity Switches

Related Config File Sections
switches:

Proximity switches operate via the interaction of the ball within a magnetic field created by the switch. Unlike a reed switch, which also uses a magnetic field to sense the ball, proximity switches differ in that they do not have any moving parts. However, a voltage must be applied and they require additional circuitry compared to a reed switch. Alien pinball (heighway Pinball, 2017) makes signifigant use of proximity switches in liu of the traditional thru-playfield mechanical leaf-blade style switches. An advantage of proximity switches (other than not needing to make thru-playfield cuts or have mechanical parts wear out), is that they can be designed with different levels of sensitivity, or even made tunable.

Todo: Add a picture (*Help us to write it*).

For homebrew pinball applications, while they are not typically available from major pinball suppliers due to their scarcity in current and past pinballs, there are a few online sources of these switches – including the exact ones used in Alien Pinball.

Wiring will depend on the exact switch used. With SW-16 switch boards, the use of pull-up resistors will likely be required when supplying a direct input to the switch ports. FAST boards should work similarily.

Todo: Describe wiring (*Help us to write it*).

Reed Switches

Related Config File Sections
switches:

Todo: *Help us to write it*

Targets

Drop Targets

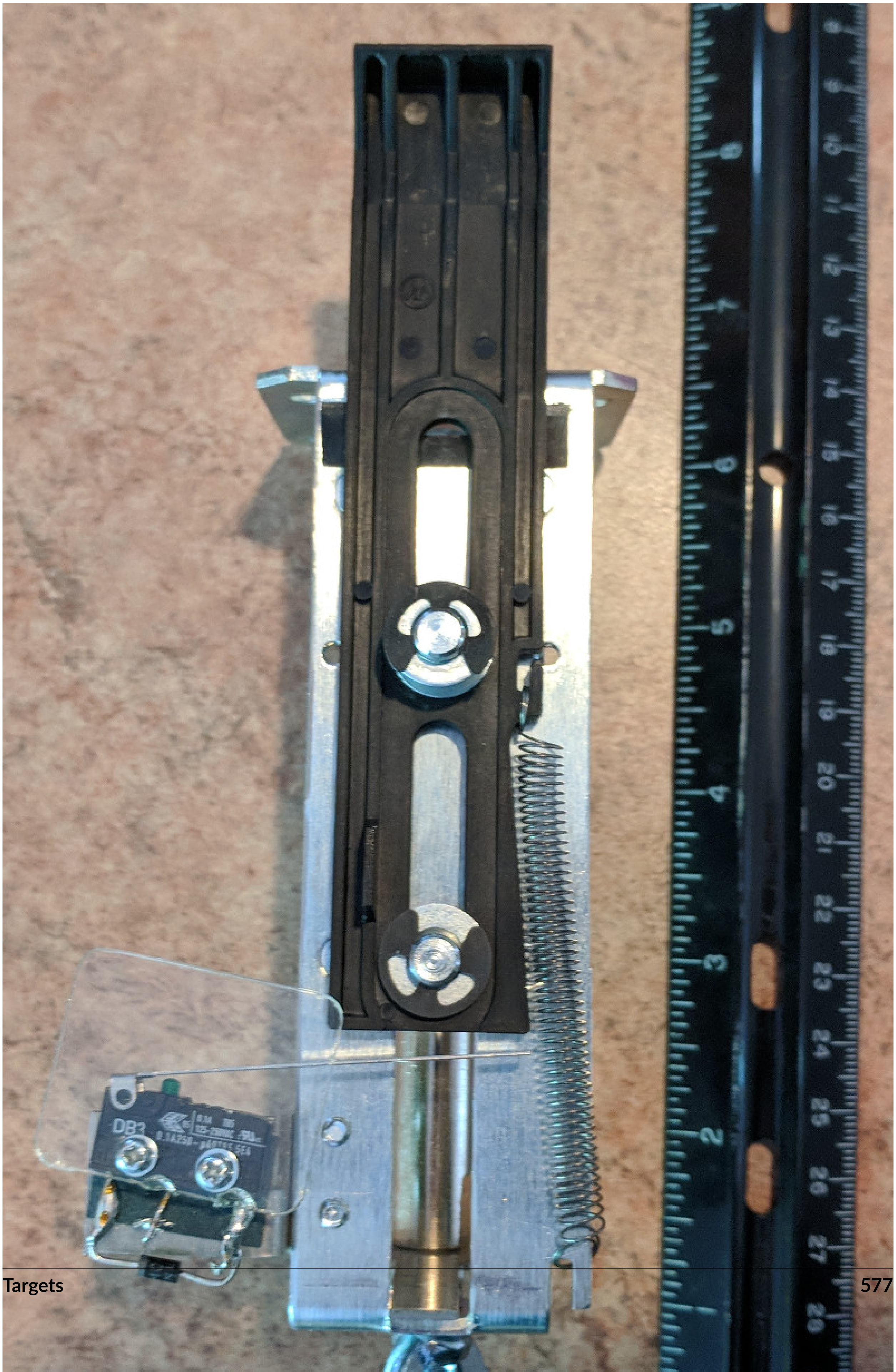
Related Config File Sections
drop_targets:
drop_target_banks:

- *Monitorable Properties*
- *Related How To guides*
- *Related Events*

Mission Pinball Framework's (MPF) *drop target* device represents a switch in a pinball machine. This device is used for drop target banks with a coil for resetting. If the reset coil resets more than just this one drop target configure all targets as a *drop target bank* and put the coil there. Additionally, there may be a knockdown coil which allows the software to knock the target down.


AE-26-1200


AE-26-1200



This is an example:

```
switches:
  s_drop_target:
    number:
coils:
  c_reset_drop_target:
    number:
  c_knock_down_coil:
    number:
drop_targets:
  d_drop_target:
    switch: s_drop_target
    reset_coil: c_reset_drop_target
    knockdown_coil: c_knock_down_coil
```

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for drop targets is `device.drop_targets.<name>`.

complete Boolean (true/false) which shows whether this drop target is complete (down).

Related How To guides

- *Drop Target Bank*
- *Fixing Drop Target Reset Issues*

Related Events

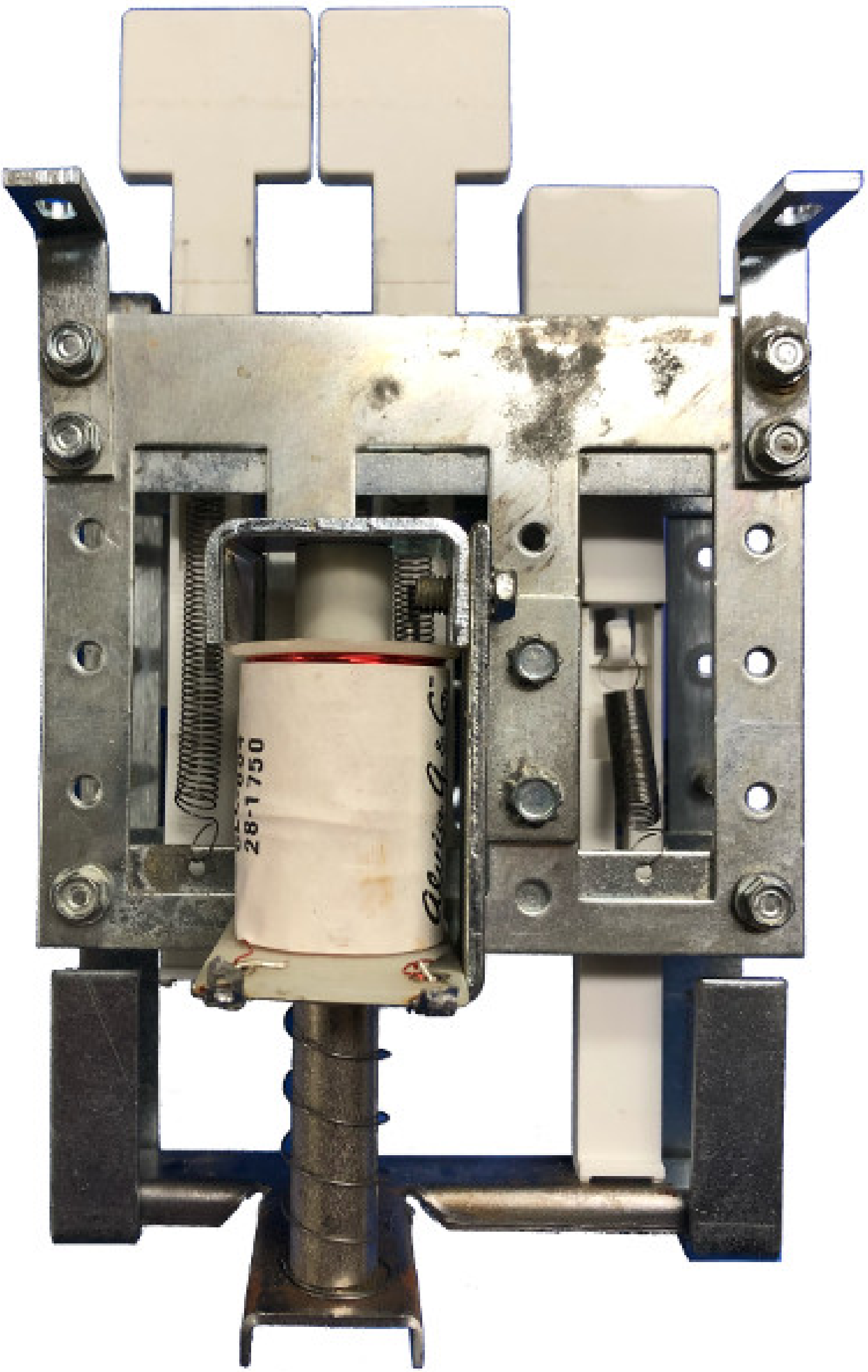
- *drop_target_(name)_down*
- *drop_target_(name)_up*

Drop Target Bank

Related Config File Sections
drop_targets:
drop_target_banks:

- *Monitorable Properties*
- *Related How To guides*
- *Related Events*

In MPF, you can combine multiple drop targets into drop target banks. The main reasons for doing this are to combine reset coils (since one coil typically resets an entire bank) and to get additional events posted when the entire bank is up, down or in a mixed state.



This is an example:

```
drop_targets:
  front:
    switch: s_drop_front
  middle:
    switch: s_drop_middle
  back:
    switch: s_drop_back
drop_target_banks:
  vuk_bank:
    drop_targets: front, middle, back
    reset_coils: c_drop_reset
    reset_on_complete: 1s
```

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for drop target banks is `device.drop_target_banks.<name>`.

complete Boolean (true/false) which shows whether every target in this bank is complete (down).

down Number of drop targets in the bank that are in the down state.

up Number of drop targets in the bank that are in the up state.

Related How To guides

- *Drop Targets*
- *Fixing Drop Target Reset Issues*

Related Events

- *drop_target_bank_(name)_down*
- *drop_target_bank_(name)_up*
- *drop_target_bank_(name)_mixed*

Fixing Drop Target Reset Issues

Related Config File Sections
drop_targets:
drop_target_banks:
psus:

- *Configuring PSU Magic*
- *Configuring Pulse Times*

Sometimes your drop targets or drop target banks will not reset reliably. This often has mechanical reasons but it could be also caused by a stressed power supply or bad electrical wiring. Try to rule those out or fix them first. However, you still might have issues afterwards and we can ease them a bit in software.

Configuring PSU Magic

Behind the scenes MPF performs *some magic for you to prevent stress on your power supply unit*. The default should be fine for most machine but if your PSU is very weak try this config:

```
psus:
  default:
    release_wait_ms: 50    # defaults to 10ms
```

Configuring Pulse Times

Increasing the pulse time on your reset coil should help with reliably resetting your drop target or drop target bank. However, it will also cause mechanical stress, heats up your reset coil and might draw a lot of power out of your power supply.

One solution to this is to lower `default_pulse_power` to something between `.5` and `.8`. Your hardware will PWM the pulse and you can use much longer pulse times without too much stress on your hardware. Also reduces the sound caused by the reset.

Resetting a Drop Target Multiple Times

If all the above does not help and you got an old mech which somehow does not like to snap in place all the time you can also try to trigger the reset multiple times. MPF will not reset drop targets or drop target banks if they are already completely up so this should not cause too much stress.

The following example will try to reset your drop target bank up to three times on ball start:

```
drop_targets:
  front:
    switch: s_drop_front
  middle:
    switch: s_drop_middle
  back:
    switch: s_drop_back
drop_target_banks:
  vuk_bank:
    drop_targets: front, middle, back
    reset_coils: c_drop_reset
    reset_on_complete: 1s
    reset_events:
      ball_started.1: 0
      ball_started.2: 1s
      ball_started.3: 2s
      machine_reset_phase_3: 0
```

Kicking Targets

Related Config File Sections
switches:
kickbacks:

TODO: Add a picture of a kicking target

Mission Pinball Framework's (MPF) *kicking target* device represents a switch in a pinball machine. This device is used for kicking targets with a coil for kicking. Used rarely, these targets look like stationary targets, but when hit they kick the back in the opposite direction much like a *slingshot* or *bumper*.

```
switches:
  s_kicking_target:
    number: 1
coils:
  c_kicking_target:
    number: 1
    default_pulse_ms: 10ms
kickbacks:
  kicking_target:
    coil: c_kicking_target
    switch: s_kicking_target
```

Stationary or Standup Targets

Related Config File Sections
switches:


Ground
Input

Mission Pinball Framework's (MPF) *stationary target* device represents a switch in a pinball machine. This might also be know as a stand-up target. It is essentially a switch above the playfield with a scoring value associated with it. When the ball hits it the value is scored.

```
switches:
  s_target:
    number: 5
    debounce: quick
    ignore_window_ms: 1000ms
```

Most platforms support debouncing of switches for a few ms. Usually, you have to reduce debouncing to 1-2ms because a strong hit to a target might be very short (see `debounce` in *switches:*). However, targets sometimes start to swing after a hit and would cause multiple hits. To prevent that you can set `ignore_window_ms` to prevent multiple hits within that window.

Vari Targets

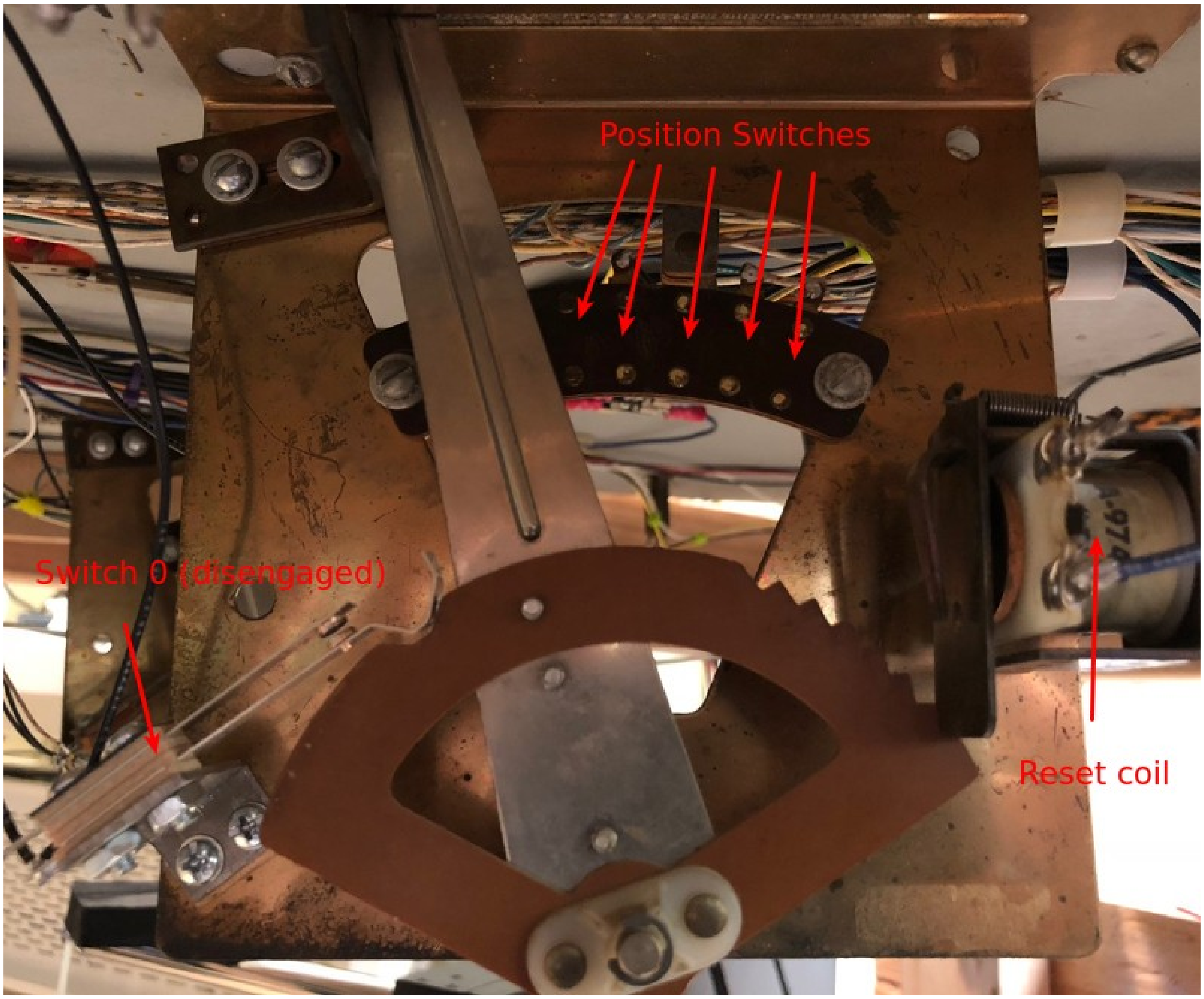
Related Config File Sections
switches:

Mission Pinball Framework's (MPF) *vari target* device represents a switch in a pinball machine. It is a metal arm that pivots under the playfield and awards a scoring value associated with it that changes depending on how hard the ball hits it. Typically the harder the ball hit the more points awarded.

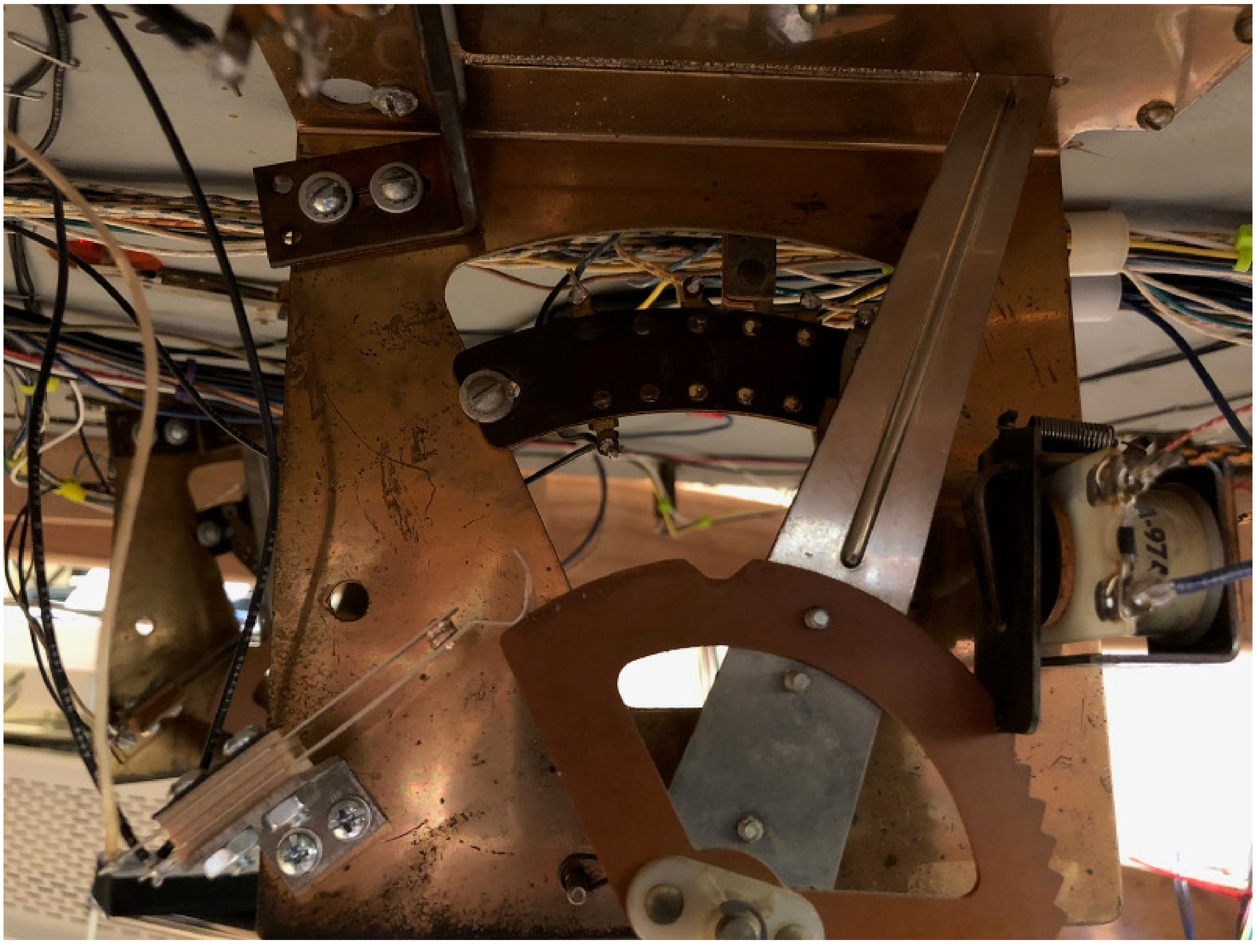
This is a vari-target in a Gottlieb Playball (1971):


HOME RUN

Technically, a vari-target has one switch per position and a reset coil to reset the target:



It can reset the target at any position. Either directly after a hit or once it has moved till the end (or never). This is how a vari-target looks fully engaged:



If you got an example config for a vari target *please contribute it*.

Related Config File Sections
drop_targets:
drop_target_banks:
switches:

There are many types of targets on a pinball playfield some of which are described here. In the Mission Pinball Framework(MPF) they are handled in a number of ways depending. In some instances they are just a switch hit while in others they may require a coil to be fired to reset or fire the ball back at the player.

Related How To Guides
Drop Targets
Kicking Targets
Stationary or Standup Targets
Vari Targets

Related Events
drop_target_(name)_down
drop_target_(name)_up
drop_target_bank_(name)_down
drop_target_bank_(name)_mixed
drop_target_bank_(name)_up

Tilt Bob

Related Config File Sections
switches:

TODO: Add a picture of a tilt bob

The tilt bob is a plumb pop centered in a metal ring which acts as a switch. On movement the switch closes which usually triggers a tilt warning.

You can configure it just like a *mechanical switch*. In addition you want to add the `tilt_warning` tag and add the built-in *tilt mode* in the list of your modes.

This is an example:

```
modes:
  - tilt
switches:
  s_tilt:
    number: 23    # number depends on your platform
    tags: tilt_warning
```

Part numbers:

- A-15361 or 04-10346 (Williams/Bally)
- 500-5023-00 (Stern)
- A-205-1 (Chicago Coin/early Stern)
- 95-0328-00 or PLABS (Bally/Capcom)

Related How To guides
Tilt

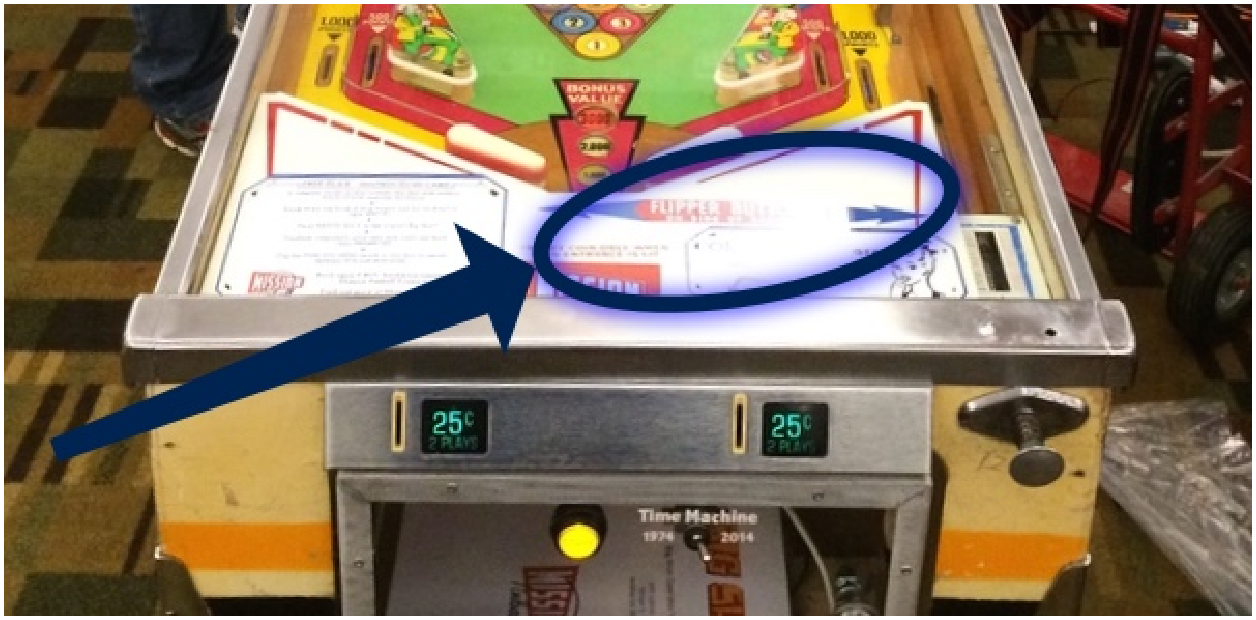
Troughs / Ball Drains

Related Config File Sections
ball_devices:

Every pinball machine will have some kind of ball trough / drain device. This is the place where the balls go when they drain from the playfield before they're ejected into the plunger lane.

In many cases, this device (or series of devices) holds multiple balls and is the location where unused balls are stored.

There are several different designs for troughs and drains that have been used over the past 70 years, and (as far as we know), MPF supports all of them. So regardless of what's in your machine, we're talking about whatever is under here:



Here are the options:

- *Modern trough with opto sensors*
- *Modern trough with mechanical switches*
- *Older style with two coils and switches for each ball*
- *Older style with two coils and only one ball switch*
- *Classic single ball, single coil*
- *Classic single ball, single coil, no shooter lane*

Since there are so many different options, you need to first identify which type of trough or ball drain system your machine has. So look at the following pictures to match up what you have, and then follow the specific links to see how to configure MPF to use it in your machine.

Video on ball tracking in MPF:

https://youtu.be/Yh8ittsuWIc

Option 1: Modern trough with opto sensors

Modern-style troughs (which have been used since about 1993 or so) are mostly located underneath the playfield and hold the balls at an incline so they roll down to the end. There is a single coil which fires to eject a ball up and out where it's directed to the plunger lane.

Todo: We need to add a photo of this type of trough (*Help us to write it*).

The advantage of modern troughs are (1) the balls entering are gravity-fed, meaning they only need one coil, and (2) they can hold a lot of balls. (Most hold 4-6 balls but you can buy ones that hold up to 8.)

If you have a modern-style trough with a circuit board on each side, that means your trough uses opto sensors to detect the presence of a ball. One of those circuit boards contains infrared LEDs which are always on which shoot invisible beams across the ball paths, and the board has sensors that detect if a light beam is broken, meaning a ball is sitting there blocking the path.

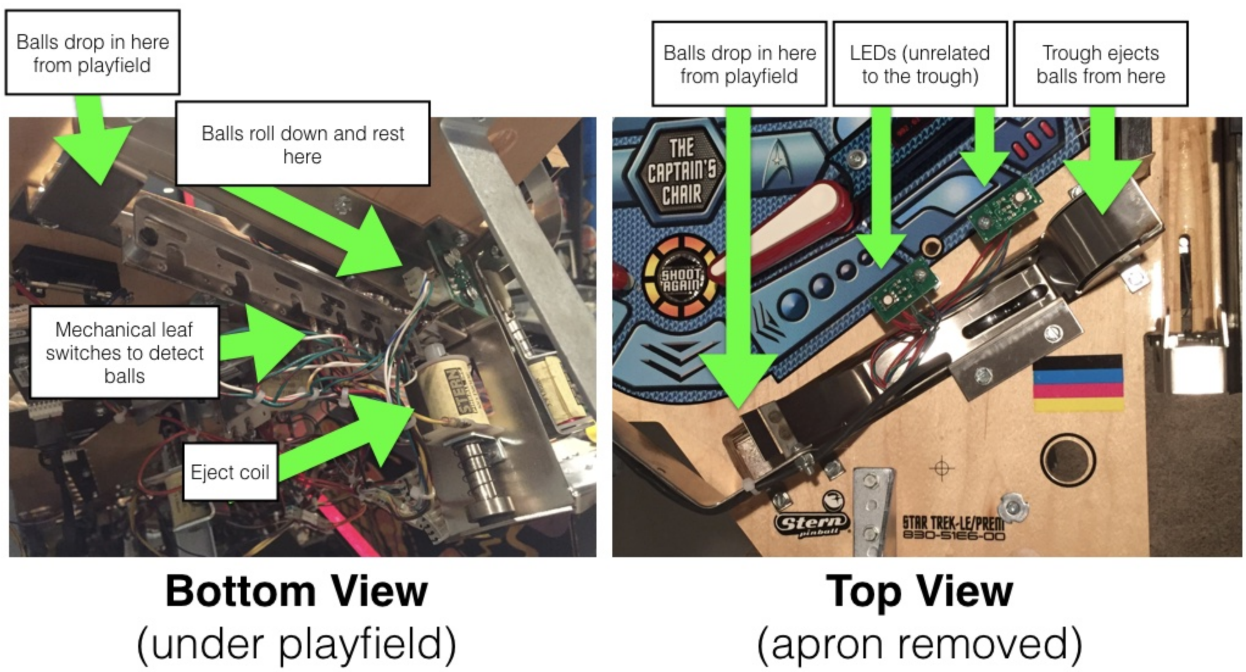
Common parts include:

- Williams: #A-16809
- Mantis Trough
- Stern #500-9820-00

If you have a modern trough with opto sensors, read the *How to configure a modern trough with opto switches* guide to continue.

Option 2: Modern trough with mechanical switches

Some modern-style troughs use mechanical switches to detect the balls rather than infrared opto boards. (Other than that, they're the same as the opto-based troughs.) Here's a photo of a modern trough with mechanical switches from a Stern Star Trek Premium machine:



If you have a modern-style trough with mechanical switches instead of opto boards, then read the *How to configure a modern trough with mechanical switches* guide to continue.

Common parts include:

- Stern: #500-6318-24 (trough assembly), #535-8393-00 (center drain ball guide), #535-7329-01 (entry/exit scoop)

- Spooky: #100-0015-00 (4 balls) or #100-0016-00 (8 balls), #100-0002-00 (drain guide + enter exit scoop)

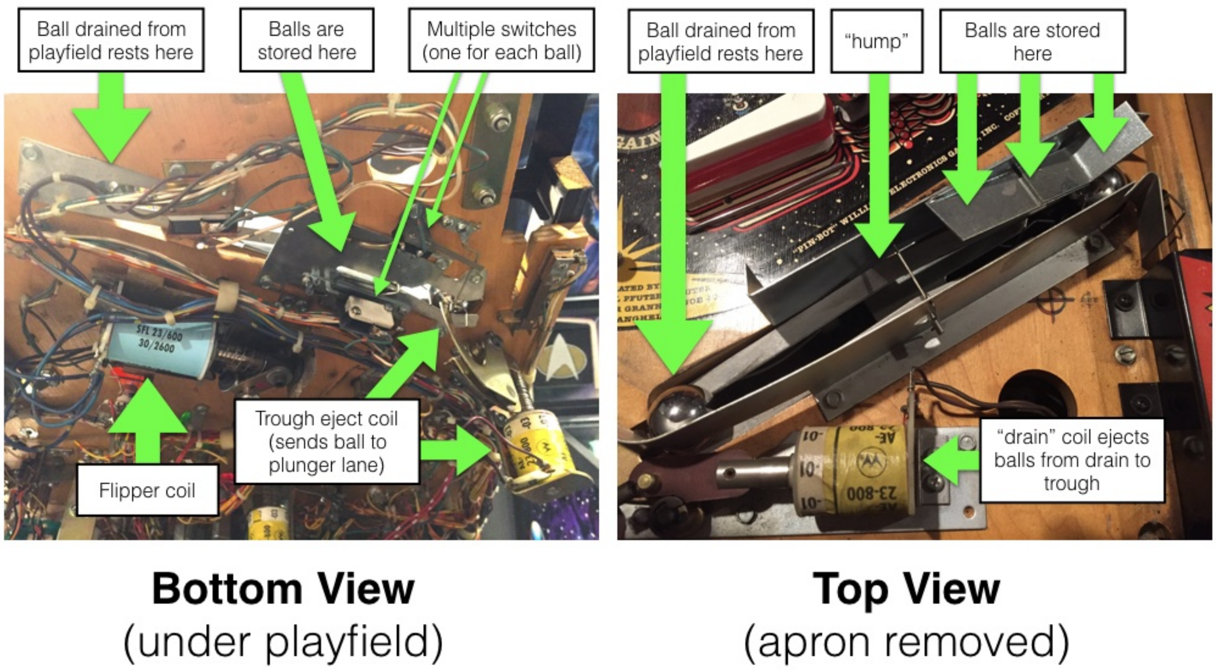
Option 3: Older style with two coils and switches for each ball

Many machines from the 1980s and early 1990s have a ball trough system that consists of two separate coils and where the balls stay "on top" of the playfield (under the apron).

In this case, when a ball drains, a coil in the drain area pulses to eject the ball up over a hump where the balls are stored. Then a second coil near the plunger lane is used to eject a single ball at a time into the plunger lane.

Some of these types "two coil" systems have multiple switches on the side that stores the balls, with there being one switch for each ball. That lets the machine know exactly how many balls are sitting there because each ball is sitting on a switch.

Here's a photo of this type of trough system from a Pin*Bot machine:



If you have this kind of trough system, read the *How to configure an older style trough with two coils and switches for each ball* guide to continue.

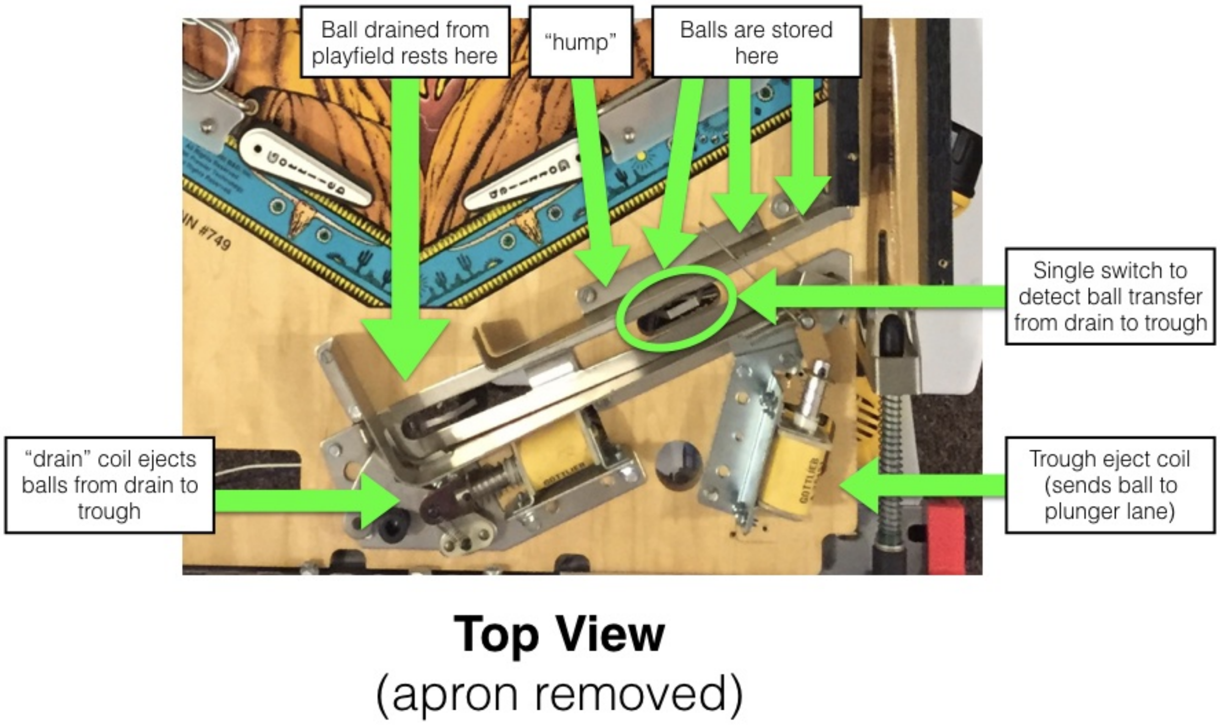
Option 4: Older style with two coils and only one ball switch

Another option is similar to Option 3 above, except there's only one switch on the trough side instead of separate switches for each ball. In these types of trough systems, the behavior of that switch changes depending on how many balls are in the trough.

If there are fewer than the max number of balls in the trough, when the drain coil pulses to eject the ball from the drain into the trough, the ball will roll over that trough switch, meaning it's activated momentarily and then deactivated again.

However, if the ball ejecting into the trough will be the final ball that will fill the trough, then that ball will rest on that trough switch, meaning that switch is solid active as long as the trough is full.

Here's a photo from a Gottlieb System 3 machine (Brooks 'n Dunn) which shows what this type of system looks like:

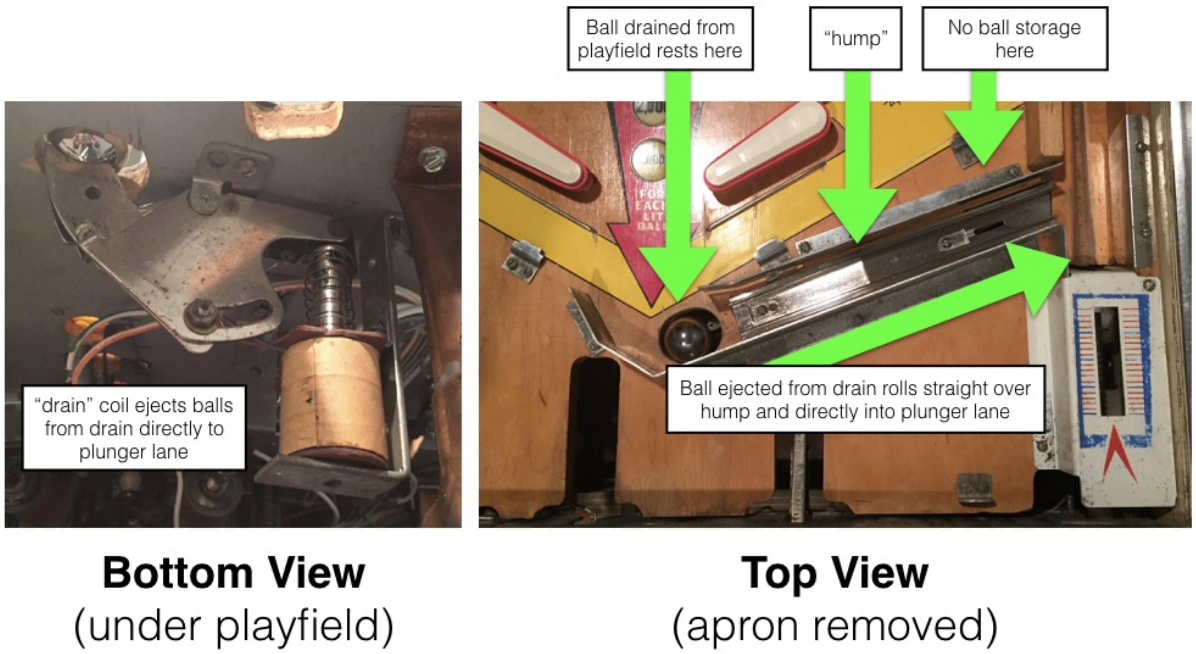


If your machine has a system similar to this, then read the *How to configure an older style trough with two coils and only one ball switch* guide to continue.

Option 5: Classic single ball, single coil

Older single-ball machines have a trough system that is on top of the playfield under the apron, but they only have a single coil near the ball drain position. The ball is stored in the drain area, and when it needs to be ejected, a coil pulses to eject it from the drain all the way into the plunger lane in a single action.

Here's an example from Gottlieb Big Shot:



If you have a system like this, read the *How to configure a classic single-ball trough* guide to continue.

Option 6: Classic single ball, single coil, no shooter lane

Very similar to Option 5 but the drain directly ejects back into the playfield. There is no shooter lane. This was used in early EM machines.

Here's an example from Gottlieb Playball:

If you have a system like this, read the *How to configure a classic single-ball trough without shooter lane* guide to continue.

Option 7: Something we haven't seen yet

If you're using MPF with a machine that has some kind of trough or drain system that we haven't covered here, we would like to know about it so we can write a how to guide and/or add support for it in MPF.

As far as we know, however, these 6 options should cover everything. For example, you might have a machine that you think is different, but when you really look at it, it's just a weird form of one of these 6 options. (Bally Fathom is a great example of this. It's like a classic single-ball trough where there is a drain that ejects a ball all the way into the plunger lane, but there are two additional switches in the apron wall where balls rest before they land in the drain device. That style of drain and trough is actually configured using Option 2, the modern trough with mechanical switches.)

If you have something weird that you can't figure out, we're happy to help! Just post a photo of it to MPF Users Google Group and we'll go from there.

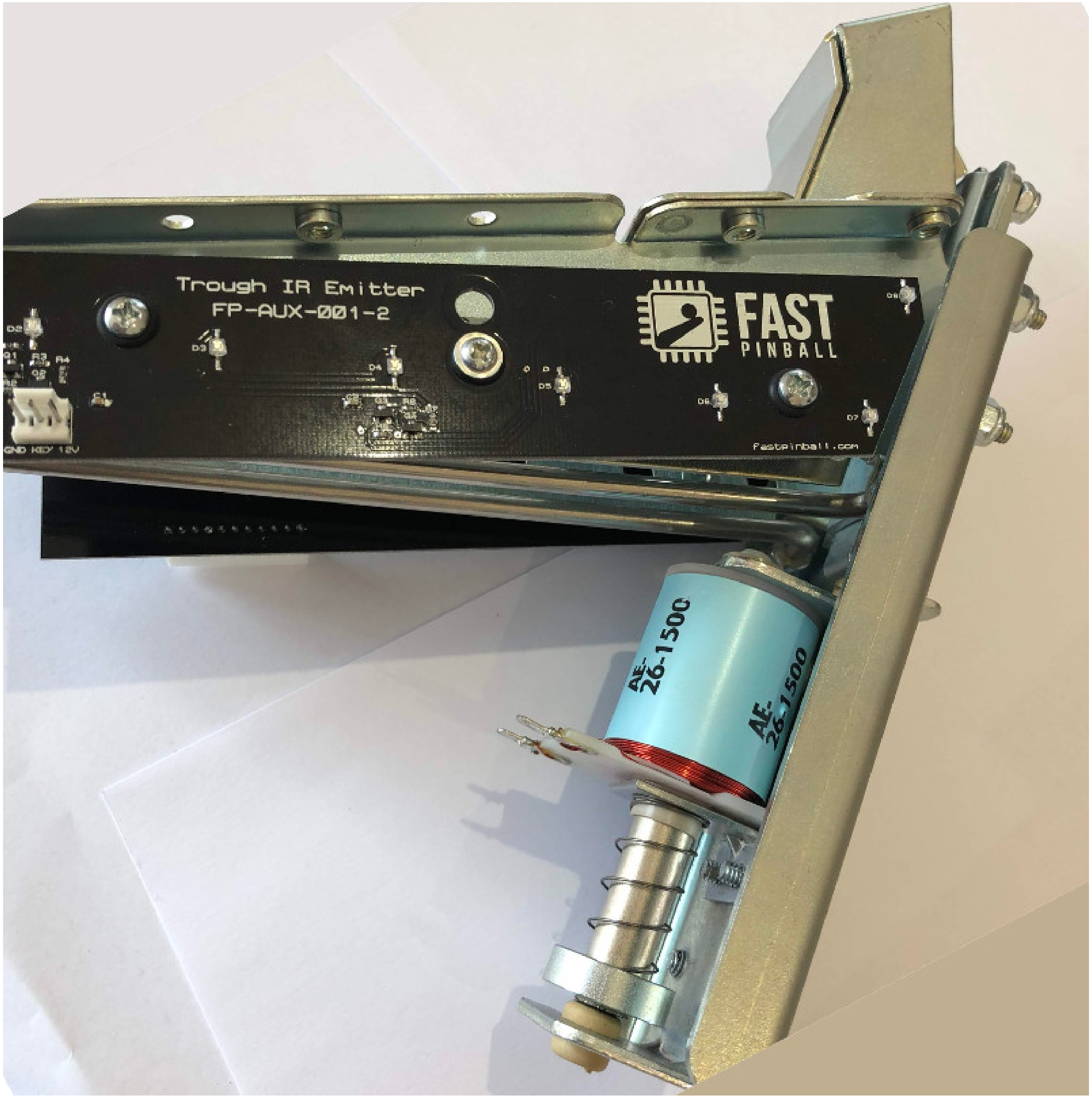
Related How To Guides
Tutorial step 7: Add your trough
Troubleshooting P-Roc/P3-Roc

Related Events
ball_drain
balldevice_ball_missing
balldevice_balls_available
balldevice_(name)_ball_missing
balldevice_captured_from_(captures_from)
balldevice_(name)_ball_eject_attempt
balldevice_(name)_ball_eject_failed
balldevice_(name)_ball_eject_success
balldevice_(name)_ejecting_ball

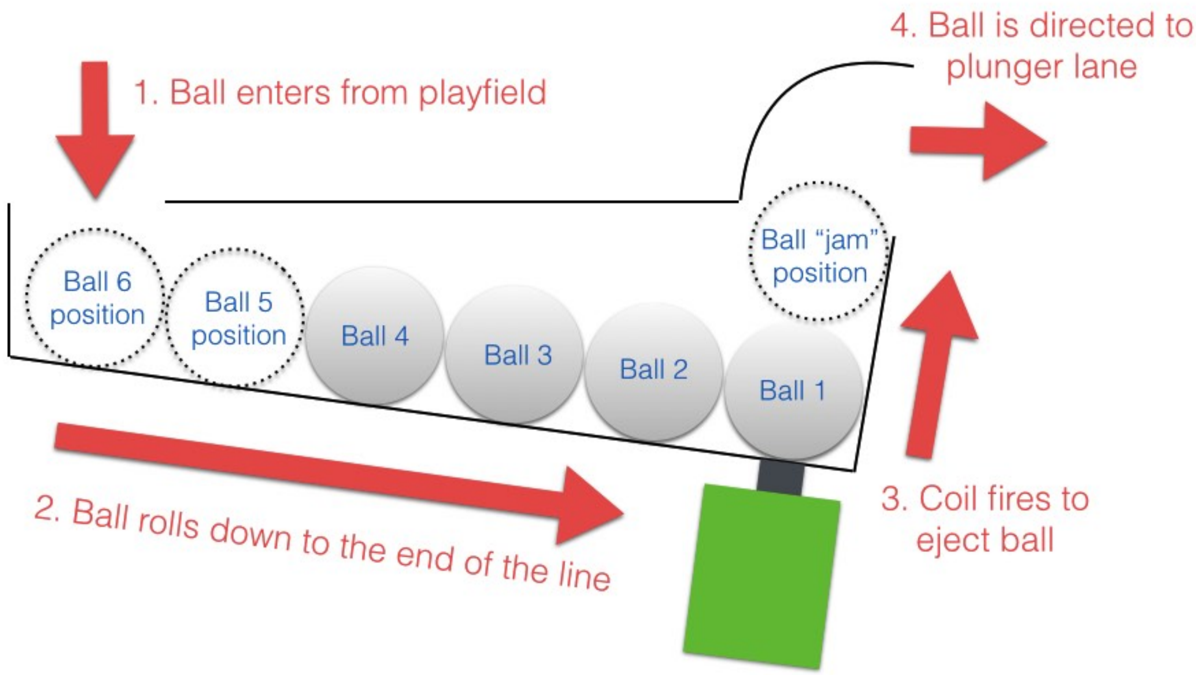
How to configure a modern trough with opto switches

Related Config File Sections
ball_devices:
playfields:

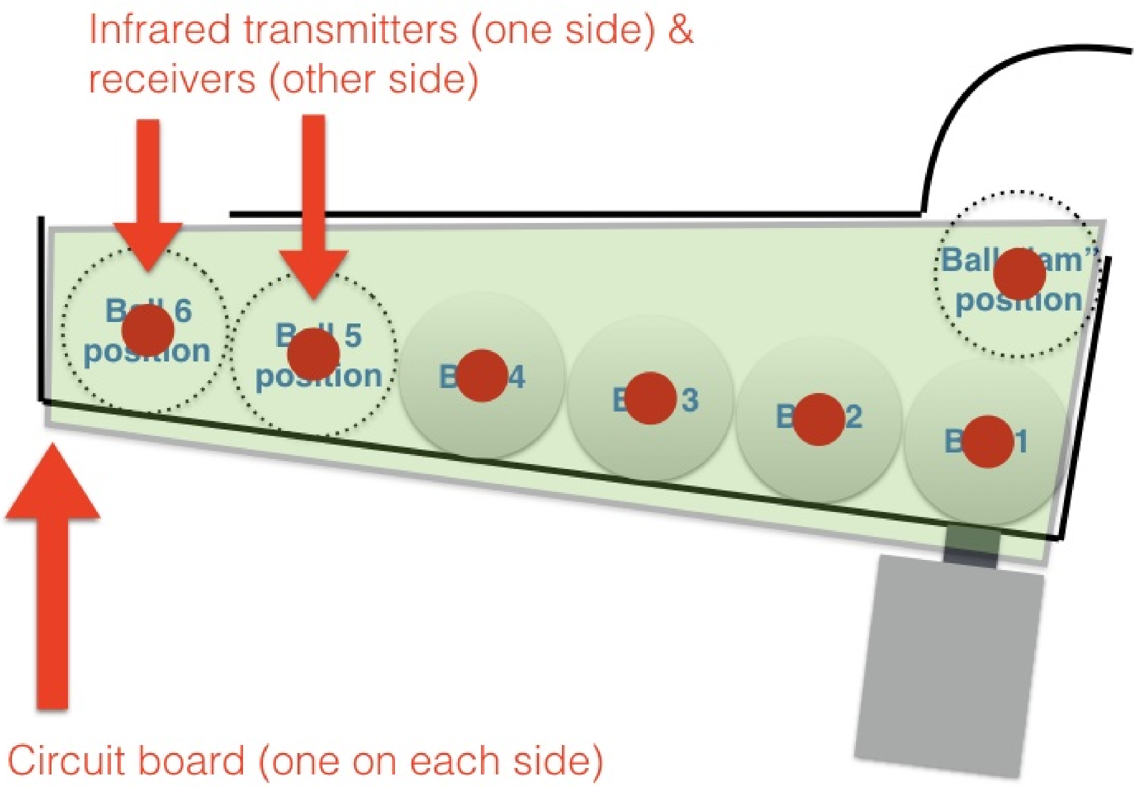
This guide will show you how to configure MPF to use a modern-style trough with opto boards. (If you have a modern-style trough which uses mechanical leaf switches, use *this guide* instead.)



The following diagram shows how the ball flow and eject coil work in a modern trough. (This is a side view)



And this diagram shows how the "opto boards" are typically located. Note that one of the opto boards is a "transmit" board that contains infrared LEDs which are always on, and the other side is the "receive" board which contains photo transistors which are activated when the IR beam is hitting them (i.e. when there is no ball blocking the path) and inactive when a ball is present and in the way.



If you got a Stern Spike Trough but are not using Stern Spike (not recommended) read the *Stern Spike Trough guide*.

0. Connect your trough

Skip this step if your trough is already connected. Otherwise, you need to power your opto transmitters and connect the opto receivers to your inputs. Make sure that you got proper current limiting in place. This might be already present on the trough PCB (i.e. on older Stern troughs) or you might need to add current limiting resistors. Read the *Opto section* for details if in doubt.

Bally/Williams Trough Opto Boards:

Part numbers:

- Transmitter: #A-18617-1 or 5768-14121-02 or #600-0035-00 or #600-0005-00
- Receiver: #A-18618-1 or 5768-14122-02 or #600-0036-00 or #600-0006-00
- Transmitter/Receiver: #600-0054-00 or #600-0055-00

Those boards need an additional current limiting resistor on the transmitter. Read the *Opto section* for details if in doubt. You can connect the receivers one by one to your inputs. Don't forget to connect your the receiver board to ground.

FAST Trough Opto Boards:

Part numbers:

- Transmitter: FP-AUX-001-?
- Receiver: FP-AUX-001-2

The FAST transmitter already has parts for current limiting and you can connect it directly to 12V and ground. You can connect the receivers one by one to your inputs. Don't forget to connect your the receiver board to ground.

Stern Trough Opto Boards:

Part numbers:

- Transmitter: 515-0173-00/520-5173-00
- Receiver: 515-0174-00/520-5174-00

This board only covers the first ball position and the jam position. All other positions are typically covered by normal switches. Transmitter contains current limiting circuit and you can connect it directly to 5V. The receiver needs to be powered and also inverts the optos. There is typically no need to set `NC` on using those boards. You can follow the *How to configure a modern trough with mechanical switches* guide to configure your trough.

Spike Trough Opto Boards:

Part numbers:

- Transmitter: 520-5344-00
- Receiver: 520-5345-00/520-5345-01

If you got a Stern Spike Trough but are not using Stern Spike (not recommended) read the *Stern Spike Trough guide*.

1. Add your trough switches

The first step is to add your trough's switches to the `switches:` section of your machine config file. Create an entry in the `switches:` section for each switch in your trough, like this: (This example has six switches plus the jam switch. Yours may have more or less.)

```
switches:
  s_trough1:
    number: 2
    type: NC
  s_trough2:
    number: 3
    type: NC
  s_trough3:
    number: 4
    type: NC
  s_trough4:
```

(continues on next page)

(continued from previous page)

```
    number: 5
    type: NC
  s_trough5:
    number: 6
    type: NC
  s_trough6:
    number: 7
    type: NC
  s_trough_jam:
    number: 8
    type: NC
```

Note that we configured this switches with numbers `02` through `08`, but you should use the actual switch numbers for your control system that the trough optos are connected to. (See *How to configure "number:" settings* for instructions for each type of control system.)

It makes no difference which switch is which (in terms of whether Switch 1 is on the left side or the right side). Also the actual switch names don't really matter. We use *s_trough1* through *s_trough6* plus *s_trough_jam*, though you can call them *s_ball_trough_1* or *trough_ball_1* or *s_mr_potatohead*.

Note: The "jam" switch position is the switch which detects if a ball is sitting on top of the lowest ball. We think all modern opto troughs have optos to detect the jams, but if yours doesn't, that's fine—just don't enter it. (If you have it though you definitely want to use it because it makes MPF smarter about how it handles balls that get stacked.)

2. Add your trough eject coil

Next, create an entry in your `coils:` section for your trough's eject coil. Again, the name doesn't matter. We'll call this *c_trough_eject* and enter it like this:

```
coils:
  c_trough_eject:
    number: 04
    default_pulse_ms: 20
```

Again, the `number:` entries in your config will vary depending on your actual hardware, and again, you can pick whatever name you want for your coil.

You'll also note that we went ahead and entered a `default_pulse_ms:` value of 20 which will override the default pulse time of 10ms. It's hard to say at this point what value you'll actually need. You can always adjust this at any time. You can play with the exact values in a bit once we finish getting everything set up.

3. Add your "trough" ball device

In MPF, the trough is a *ball device*, so you'll add a configuration for it to the `ball_devices:` section of your machine config. (If you don't have that section add it now.)

Then in your `ball_devices:` section, create an entry called `bd_trough:`, like this:

```
ball_devices:
  bd_trough:
```

This means that you're creating a ball device called *bd_trough*. We use the preface *bd_* to indicate that this is a ball device which makes it easier when we're referencing them later. Then under your `bd_trough:` entry, start entering the configuration settings for your trough ball device:

3a. Add your trough switches to your trough ball device

Indented under `bd_trough:`, create an entry called `ball_switches:` and then add a comma-separated list of all the switches in your trough, like this:

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6, s_trough_jam
```

So this is eight spaces, followed by the word "ball_switches", then a colon, then a space, then the name of your first switch, comma, then your second switch, comma, etc. . .

Again these switches can be in any order. The key is that you're entering one switch for each position that's used to detect whether a ball is in the trough at that position.

If you have the switch in the jam position, enter it in this list too, since a ball sitting on top of another one still "counts" as a ball in the trough.

The number of switches you enter here will tell MPF how many balls your trough can hold. When MPF wants to know how many balls are in the trough, it will check all these switches to see which ones are active, and the total number active represents how many balls it's holding at that moment.

3b. Add your eject coil to your trough ball device

Next create a setting called `eject_coil:` which will be the name of the coil that MPF should fire when it wants to eject a ball from the trough. This should be the name of the coil you added in Step 2, *c_trough_eject* in our case:

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6, s_trough_jam
    eject_coil: c_trough_eject
```

Note that MPF will simply pulse the eject coil at its default pulse time when it wants to eject a ball from the trough.

3c. Add some tags to tell MPF about this device

The final configuration setting you need to enter for your trough is a list of tags which tell MPF certain things about this device.

Tags are just a comma-separated list of words you add to the `tags:` setting for a device. Ball devices can use some special tag names that tell MPF how it should use it.

First, add a tag called `trough` which tells MPF that a ball device wants to hold as many balls as it can. This probably doesn't make sense right now, which is fine, but without this tag then MPF won't know what to do with all the balls that are sitting in the trough waiting to be launched. This tag tells MPF that it's fine for this device to hold lots of balls.

Next, add a tag called `home` which tells MPF that any balls in this device are considered to be in their "home" positions. When MPF first starts up, and after a game ends, it will automatically eject any balls from any devices that are not tagged with "home." When a player tries to start a game, MPF will also make sure all the balls in the machine are contained in devices tagged with "home."

Finally, you need to add a tag called `drain` which is used to tell MPF that a ball entering this device means that a live ball has drained from the playfield. At this point you might be wondering why you have to enter all three of these tags. Why can't the simple `trough` tag be enough to tell MPF that a ball entering it should trigger a drain and that balls are home? This is due to the flexibility of MPF and the nearly unlimited variations of pinball machine hardware in the world. Some machines have multiple troughs. Some machines have drain devices which aren't troughs. Some machines consider balls outside the trough to be home. So even though these all might seem similar, just know that for now you have to add `trough`, `home`, and `drain` tags to your trough. You can specify the tags in any order, and your `tags:` entry should look something like this:

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6, s_trough_jam
    eject_coil: c_trough_eject
    tags: trough, home, drain
```

3d. Add & configure your jam switch

If you have a jam switch, add a setting called `jam_switch:` and add it there, like this:

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6, s_trough_jam
    eject_coil: c_trough_eject
    tags: trough, home, drain
    jam_switch: s_trough_jam
```

You can also configure an eject pulse time (in ms) that will be used when the trough wants to eject a ball but the jam switch is active. You'll have to play with your actual trough to see what this time should be. In most cases it's actually *less* time than the regular eject pulse time, because in most cases, the regular pulse time will kick out two balls (the jammed ball and the one below it).

So for our example, we'll set the jam pulse time to 15ms.

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6, s_trough_jam
    eject_coil: c_trough_eject
    tags: trough, home, drain
    jam_switch: s_trough_jam
    eject_coil_jam_pulse: 15ms
```

(Note that this setting is a time string, so you can include the "ms" in the setting value.)

4. Configure your virtual hardware to start with balls in the trough

While we're talking about the trough, it's probably a good idea to configure MPF so that when you start it in virtual mode (with no physical hardware) that it starts with the trough full of balls. To do this, add a new section to your config file called `virtual_platform_start_active_switches:`. (Sorry this entry name is hilariously long.) As its name implies, *virtual_platform_start_active_switches:* lets you list the names of switches that you want to start in the "active" state when you're running MPF with the virtual platform interfaces.

The reason these only work with the virtual platforms is because if you're running MPF while connected to a physical pinball machine, it doesn't really make sense to tell MPF which switches are active since MPF can read the actual switches from the physical machine. So you can add this section to your config file, but MPF only reads this section when you're running with one of the virtual hardware interfaces. To use it, simply add the section along with a list of the switches you want to start active. For example:

```
virtual_platform_start_active_switches:
  - s_trough1
  - s_trough2
  - s_trough3
  - s_trough4
  - s_trough5
  - s_trough6
```

5. Add your plunger lane

Remember that ball devices in MPF know what their "target" devices are, meaning that they understand the chain of devices the ball path takes. (For example, the trough ejects to the plunger lane which ejects to the playfield which drains to the trough. . .)

So in order to completely configure your trough, you need to tell it the name of thes devices that it ejects to. For the purposes of this How To guide, we'll just create a placeholder plunger lane called *bd_plunger*, though you should see the *Plungers & Ball Launch Devices* documentation for full details since there are lots of different types of plungers.

You add an eject target via the `eject_targets:` section, like this:

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6, s_trough_jam
    eject_coil: c_trough_eject
    tags: trough, home, drain
    jam_switch: s_trough_jam
    eject_coil_jam_pulse: 15ms
    eject_targets: bd_plunger
  bd_plunger:
    ball_switches: s_plunger
    mechanical_eject: true
```

Of course you should enter the name of your actual plunger lane / ball launcher device.

Note that the `eject_targets:` entry is "targets" (plural), but in this case we're only adding a single target. That's fine and how you would configure a trough since it only ejects to one place (the plunger lane). Some devices eject to pathways with diverters which can direct the ball to multiple different places, so that's the scenario where you'd enter more than one target. But for the trough, it's just the one.

6. Configure eject timeouts

Your trough will try to eject as fast as possible (i.e. during a multiball) but it has to wait that ball cannot return and stack up. By default MPF will wait `10s` after a ball to make sure that it settled in the shooter lane or returned (in the latter case the trough will retry the eject). For the trough this works fine if the ball actually settles in the shooter lane but sometimes a player might as well launch the ball without hitting the plunger switch. For that reason it is important to set `eject_timeouts` to your shooter lane and your trough. You should measure how long the maximum time is until a ball cannot possibly return to your trough and plunger (with some safety margin). Usually this is about `2s - 4s` for a trough and `3s - 5s` for a plunger.

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6, s_trough_jam
    eject_coil: c_trough_eject
    tags: trough, home, drain
    jam_switch: s_trough_jam
    eject_coil_jam_pulse: 15ms
    eject_targets: bd_plunger
    eject_timeouts: 3s
  bd_plunger:
    ball_switches: s_plunger
    mechanical_eject: true
    eject_timeouts: 5s
```

Here's the complete config

```
switches:
  s_trough1:
    number: 2
    type: NC
  s_trough2:
    number: 3
    type: NC
  s_trough3:
    number: 4
    type: NC
  s_trough4:
    number: 5
    type: NC
  s_trough5:
    number: 6
    type: NC
  s_trough6:
    number: 7
    type: NC
  s_trough_jam:
    number: 8
```

(continues on next page)

(continued from previous page)

```
    type: NC
  s_plunger:
    number: 10
coils:
  c_trough_eject:
    number: 4
    default_pulse_ms: 20
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6, s_trough_jam
    eject_coil: c_trough_eject
    tags: trough, home, drain
    jam_switch: s_trough_jam
    eject_coil_jam_pulse: 15ms
    eject_targets: bd_plunger
    eject_timeouts: 3s
  bd_plunger:
    ball_switches: s_plunger
    mechanical_eject: true
    eject_timeouts: 5s
playfields:
  playfield:
    default_source_device: bd_plunger
    tags: default
virtual_platform_start_active_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6
```

What if it doesn't work?

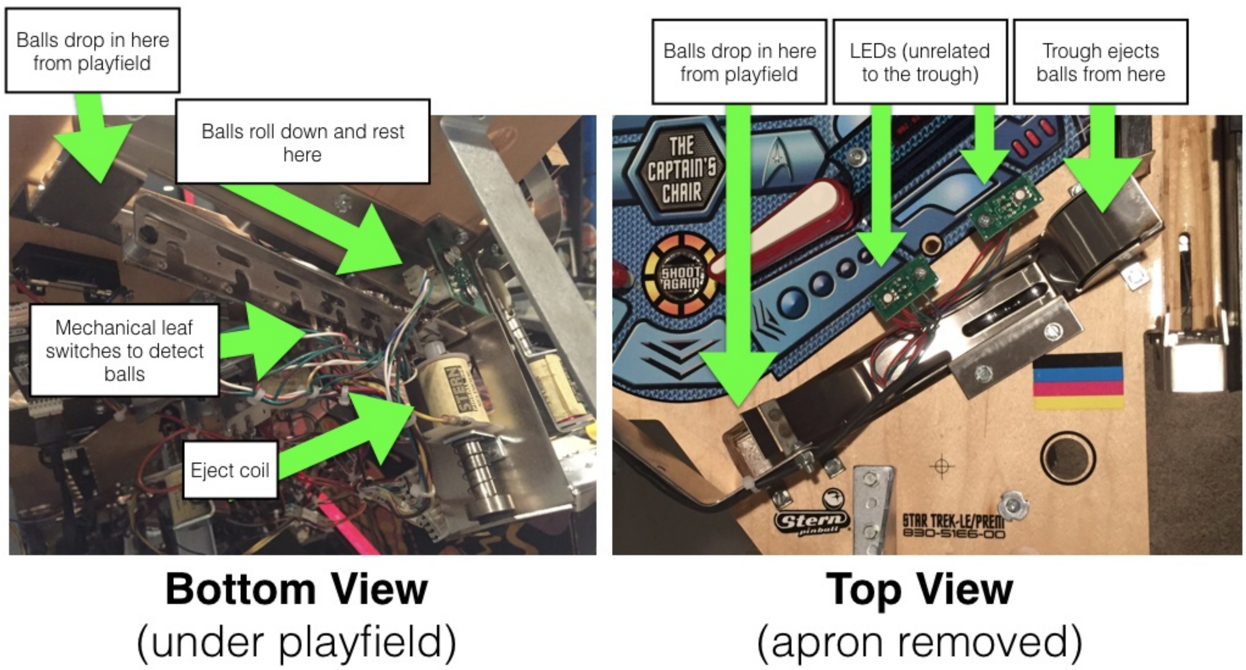
Have a look at our *troubleshooting guide for ball_devices*.

How to configure a modern trough with mechanical switches

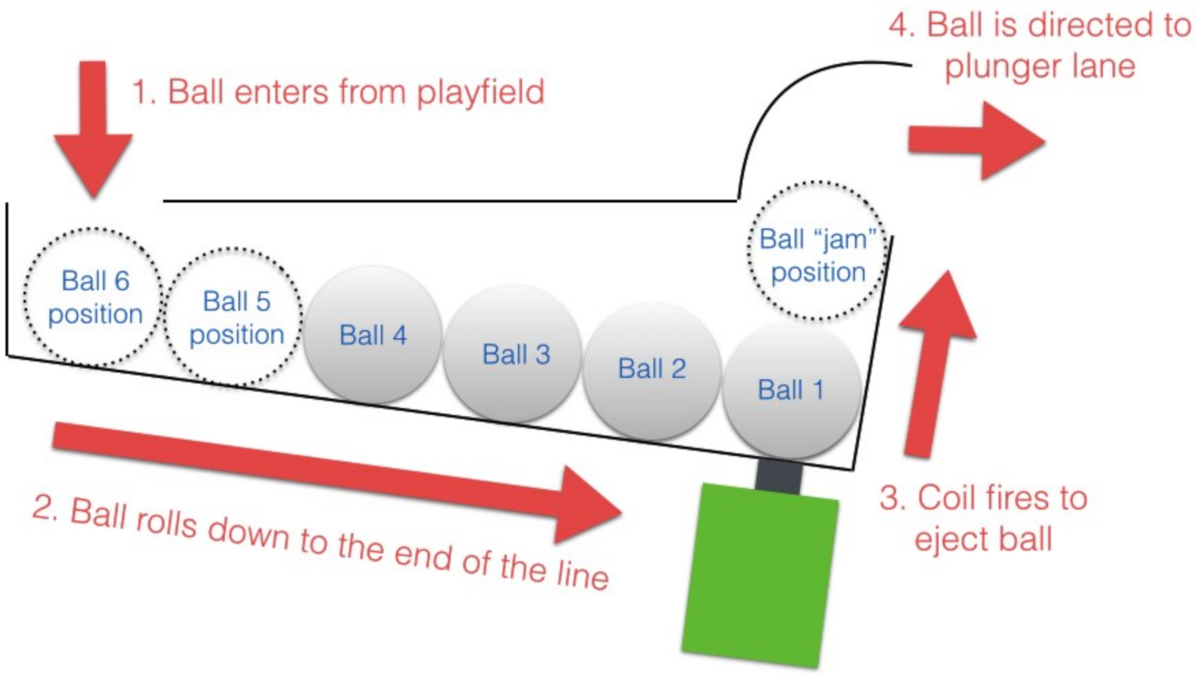
Related Config File Sections
ball_devices:
playfields:

This guide will show you how to configure MPF to use a modern-style trough which uses mechanical leaf switches. If you have a modern trough that uses opto boards, use *this guide* instead.

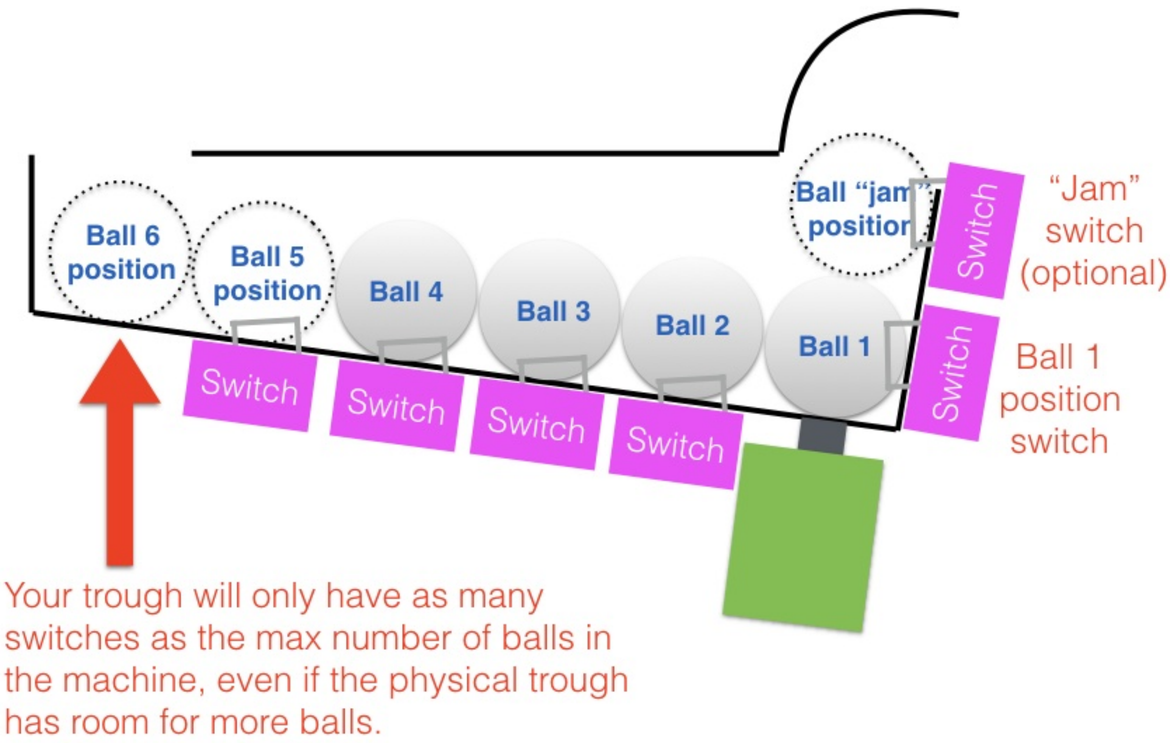
Here's an example from a Stern Star Trek Premium machine:



The following diagram shows how the ball flow and eject coil work in a modern trough. (This is a side view)



And this diagram shows how the switches are typically arranged in a modern trough with mechanical switches:



Note: Not all modern troughs have the "jam" switch, and depending on how many balls were designed to go in your machine, it's possible that not all the ball switches are populated. (Though you can add more to increase the number of balls in your machine!)

1. Add your trough switches

The first step is to add your trough's switches to the `switches:` section of your config file. Create an entry in your `switches:` section for each switch in your trough, like this: (This example has six switches plus the jam switch. Yours may have more or less.)

```
switches:
  s_trough1:
    number: 2
  s_trough2:
    number: 3
  s_trough3:
    number: 4
  s_trough4:
    number: 5
  s_trough5:
    number: 6
  s_trough6:
    number: 7
  s_trough_jam:
    number: 8
```

Note that we configured this switches with numbers `02` through `08`, but you should use the actual switch numbers for your control system that the trough switches are connected to. (See *How to configure "number:" settings* for instructions for each type of control system.)

It makes no difference which switch is which (in terms of whether Switch 1 is on the left side or the right side). Also the actual switch names don't really matter. We use *s_trough1* through *s_trough6* plus *s_trough_jam*, though you can call them *s_ball_trough_1* or *trough_ball_1* or *s_mr_potatohead*.

Note: The "jam" switch position is the switch which detects if a ball is sitting on top of the lowest ball. Not all troughs have this, so if yours doesn't, that's fine—just don't enter it. (If you have it though you definitely want to use it because it makes MPF smarter about how it handles balls that get stacked.)

---

2. Add your trough eject coil

Next, create an entry in your `coils:` section for your trough's eject coil. Again, the name doesn't matter. We'll call this *c_trough_eject* and enter it like this:

```
coils:
  c_trough_eject:
    number: 4
    default_pulse_ms: 20
```

Again, the `number:` entries in your config will vary depending on your actual hardware, and again, you can pick whatever name you want for your coil.

You'll also note that we went ahead and entered a `default_pulse_ms:` value of 20 which will override the default pulse time of 10ms. It's hard to say at this point what value you'll actually need. You can always adjust this at any time. You can play with the exact values in a bit once we finish getting everything set up.

3. Add your "trough" ball device

In MPF, the trough is a *ball device*, so you'll add a configuration for it to the `ball_devices:` section of your machine config. (If you don't have that section add it now.)

Then in your `ball_devices:` section, create an entry called `bd_trough:`, like this:

```
ball_devices:
  bd_trough:
```

This means that you're creating a ball device called *bd_trough*. We use the preface *bd_* to indicate that this is a ball device which makes it easier when we're referencing them later. Then under your `bd_trough:` entry, start entering the configuration settings for your trough ball device:

3a. Add your trough switches to your trough ball device

Indented under `bd_trough:`, create an entry called `ball_switches:` and then add a comma-separated list of all the switches in your trough, like this:

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6, s_trough_jam
```

So this is eight spaces, followed by the word "ball_switches", then a colon, then a space, then the name of your first switch, comma, then your second switch, comma, etc. . .

Again these switches can be in any order. The key is that you're entering one switch for each position that's used to detect whether a ball is in the trough at that position.

If you have the switch in the jam position, enter it in this list too, since a ball sitting on top of another one still "counts" as a ball in the trough.

The number of switches you enter here will tell MPF how many balls your trough can hold. When MPF wants to know how many balls are in the trough, it will check all these switches to see which ones are active, and the total number active represents how many balls it's holding at that moment.

3b. Add your eject coil to your trough ball device

Next create a setting called `eject_coil:` which will be the name of the coil that MPF should fire when it wants to eject a ball from the trough. This should be the name of the coil you added in Step 2, *c_trough_eject* in our case:

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6, s_trough_jam
    eject_coil: c_trough_eject
```

Note that MPF will simply pulse the eject coil at its default pulse time when it wants to eject a ball from the trough.

3c. Add some tags to tell MPF about this device

The final configuration setting you need to enter for your trough is a list of tags which tell MPF certain things about this device.

Tags are just a comma-separated list of words you add to the `tags:` setting for a device. Ball devices can use some special tag names that tell MPF how it should use it.

First, add a tag called `trough` which tells MPF that a ball device wants to hold as many balls as it can. This probably doesn't make sense right now, which is fine, but without this tag then MPF won't know what to do with all the balls that are sitting in the trough waiting to be launched. This tag tells MPF that it's fine for this device to hold lots of balls.

Next, add a tag called `home` which tells MPF that any balls in this device are considered to be in their "home" positions. When MPF first starts up, and after a game ends, it will automatically eject any balls from any devices that are not tagged with "home." When a player tries to start a game, MPF will also make sure all the balls in the machine are contained in devices tagged with "home."

Finally, you need to add a tag called `drain` which is used to tell MPF that a ball entering this device means that a live ball has drained from the playfield. At this point you might be wondering why you have to enter all three of these tags. Why can't the simple `trough` tag be enough to tell MPF that a ball entering it should trigger a drain and that balls are home? This is due to the flexibility of MPF and the nearly unlimited variations of pinball machine hardware in the world. Some machines have multiple troughs. Some machines have drain devices which aren't troughs. Some machines consider balls outside the trough to be home. So even though these all might seem similar, just know that for now you have to add `trough`, `home`, and `drain` tags to your trough. You can specify the tags in any order, and your `tags:` entry should look something like this:

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6, s_trough_jam
    eject_coil: c_trough_eject
    tags: trough, home, drain
```

3d. Add & configure your jam switch

If you have a jam switch, add a setting called `jam_switch:` and add it there, like this:

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6, s_trough_jam
    eject_coil: c_trough_eject
    tags: trough, home, drain
    jam_switch: s_trough_jam
```

You can also configure an eject pulse time (in ms) that will be used when the trough wants to eject a ball but the jam switch is active. You'll have to play with your actual trough to see what this time should be. In most cases it's actually *less* time than the regular eject pulse time, because in most cases, the regular pulse time will kick out two balls (the jammed ball and the one below it).

So for our example, we'll set the jam pulse time to 15ms.

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6, s_trough_jam
    eject_coil: c_trough_eject
    tags: trough, home, drain
    jam_switch: s_trough_jam
    eject_coil_jam_pulse: 15ms
```

(Note that this setting is a time string, so you can include the "ms" in the setting value.)

4. Configure your virtual hardware to start with balls in the trough

While we're talking about the trough, it's probably a good idea to configure MPF so that when you start it in virtual mode (with no physical hardware) that it starts with the trough full of balls. To do this, add a new section to your config file called `virtual_platform_start_active_switches:`. (Sorry this entry name is hilariously long.) As its name implies, *virtual_platform_start_active_switches:* lets you list the names of switches that you want to start in the "active" state when you're running MPF with the virtual platform interfaces.

The reason these only work with the virtual platforms is because if you're running MPF while connected to a physical pinball machine, it doesn't really make sense to tell MPF which switches are active since MPF can read the actual switches from the physical machine. So you can add this section to your config file, but MPF only reads this section when you're running with one of the virtual hardware interfaces. To use it, simply add the section along with a list of the switches you want to start active. For example:

```
virtual_platform_start_active_switches:
  - s_trough1
  - s_trough2
  - s_trough3
  - s_trough4
  - s_trough5
  - s_trough6
```

5. Add your plunger lane

Remember that ball devices in MPF know what their "target" devices are, meaning that they understand the chain of devices the ball path takes. (For example, the trough ejects to the plunger lane which ejects to the playfield which drains to the trough. . .)

So in order to completely configure your trough, you need to tell it the name of thes devices that it ejects to. For the purposes of this How To guide, we'll just create a placeholder plunger lane called *bd_plunger*, though you should see the *Plungers & Ball Launch Devices* documentation for full details since there are lots of different types of plungers.

You add an eject target via the `eject_targets:` section, like this:

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6, s_trough_jam
    eject_coil: c_trough_eject
    tags: trough, home, drain
    jam_switch: s_trough_jam
    eject_coil_jam_pulse: 15ms
    eject_targets: bd_plunger
  bd_plunger:
    ball_switches: s_plunger
    mechanical_eject: true
```

Of course you should enter the name of your actual plunger lane / ball launcher device.

Note that the `eject_targets:` entry is "targets" (plural), but in this case we're only adding a single target. That's fine and how you would configure a trough since it only ejects to one place (the plunger lane). Some devices eject to pathways with diverters which can direct the ball to multiple different places, so that's the scenario where you'd enter more than one target. But for the trough, it's just the one.

6. Configure eject timeouts

Your trough will try to eject as fast as possible (i.e. during a multiball) but it has to wait that ball cannot return and stack up. By default MPF will wait `10s` after a ball to make sure that it settled in the shooter lane or returned (in the latter case the trough will retry the eject). For the trough this works fine if the ball actually settles in the shooter lane but sometimes a player might as well launch the ball without hitting the plunger switch. For that reason it is important to set `eject_timeouts` to your shooter lane and your trough. You should measure how long the maximum time is until a ball cannot possibly return to your trough and plunger (with some safty margin). Usually this is about `2s - 4s` for a trough and `3s - 5s` for a plunger.

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6, s_trough_jam
    eject_coil: c_trough_eject
    tags: trough, home, drain
    jam_switch: s_trough_jam
    eject_coil_jam_pulse: 15ms
    eject_targets: bd_plunger
    eject_timeouts: 3s
  bd_plunger:
    ball_switches: s_plunger
    mechanical_eject: true
    eject_timeouts: 5s
```

Here's the complete config

```
switches:
  s_trough1:
    number: 2
  s_trough2:
    number: 3
  s_trough3:
    number: 4
  s_trough4:
    number: 5
  s_trough5:
    number: 6
  s_trough6:
    number: 7
  s_trough_jam:
    number: 8
  s_plunger:
    number: 10
coils:
  c_trough_eject:
    number: 4
    default_pulse_ms: 20
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6, s_trough_jam
    eject_coil: c_trough_eject
    tags: trough, home, drain
    jam_switch: s_trough_jam
    eject_coil_jam_pulse: 15ms
    eject_targets: bd_plunger
    eject_timeouts: 3s
  bd_plunger:
    ball_switches: s_plunger
    mechanical_eject: true
    eject_timeouts: 5s
```

(continues on next page)

(continued from previous page)

```
playfields:
  playfield:
    default_source_device: bd_plunger
    tags: default
virtual_platform_start_active_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6
```

What if it doesn't work?

Have a look at our *troubleshooting guide for ball_devices*.

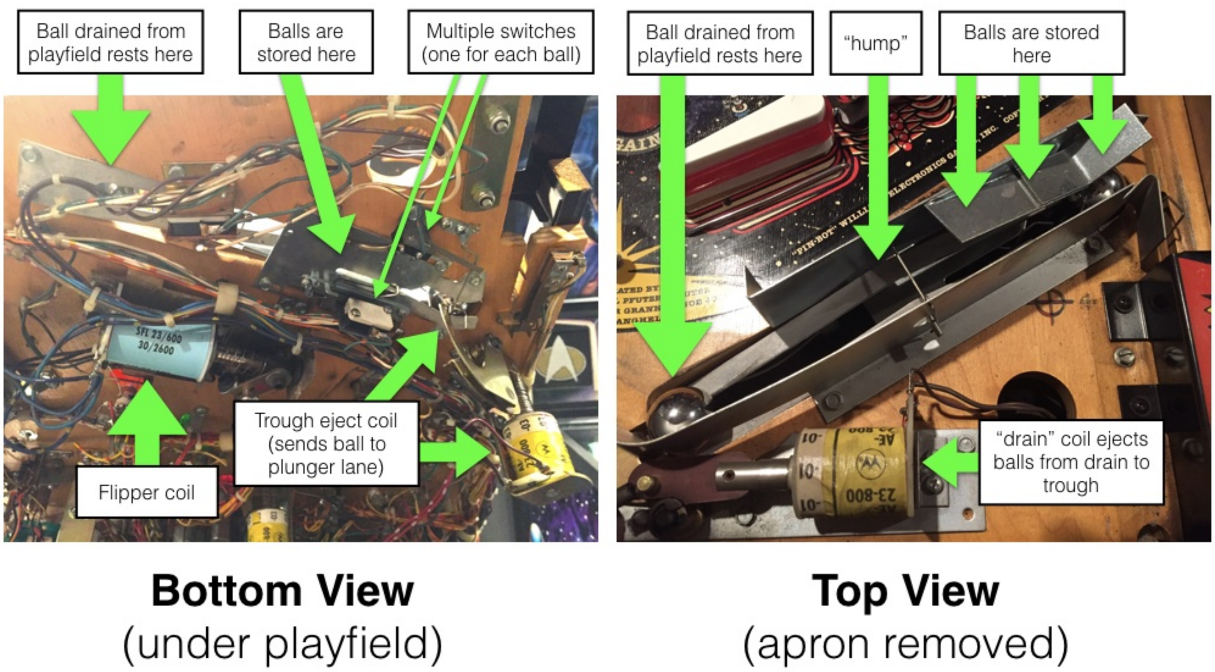
How to configure an older style trough with two coils and switches for each ball

Related Config File Sections
ball_devices:
playfields:

This guide will show you how to configure MPF to use an older-style drain and trough combination that uses two coils (one to eject the ball from the drain hole and a second to release a ball into the plunger lane).

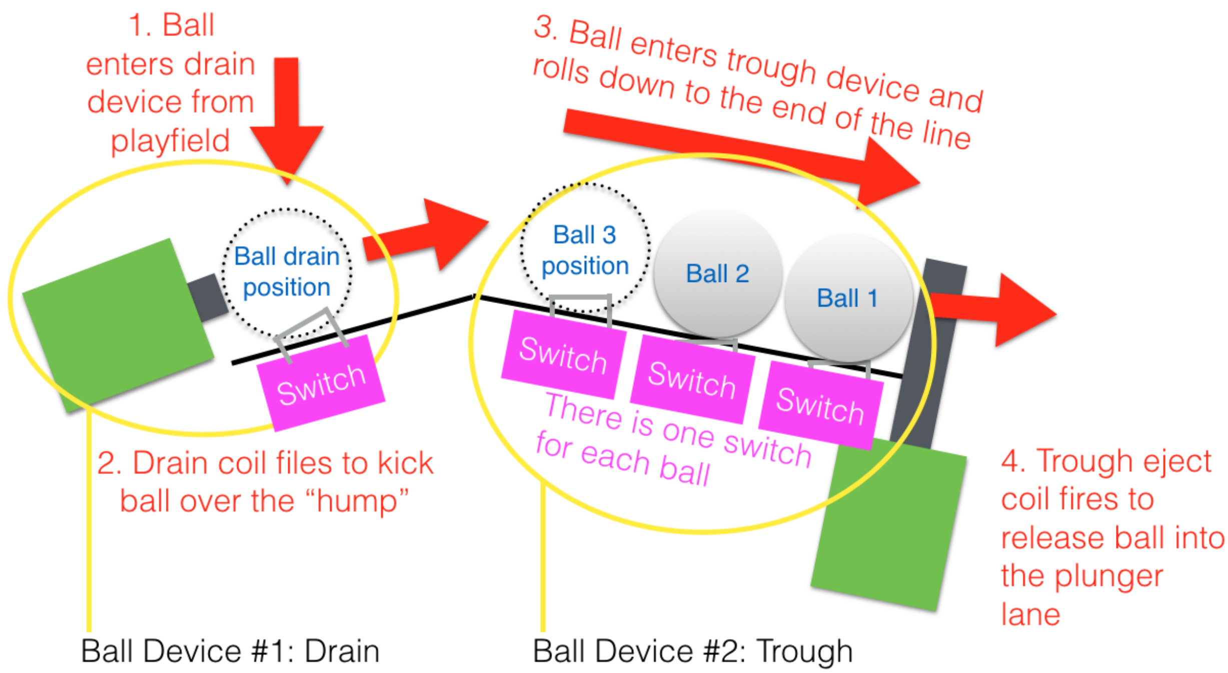
This guide is written for the types of systems where the trough side (after the "hump") has multiple switches—one for each ball that's sitting there.

Here's an example of a Williams System 11 trough that uses this system, from a Pin*Bot machine:



If your machine's trough system is like this but you only have one switch on the trough side (like Gottlieb System 3 machines), then use *this guide* instead.

The following diagram shows how the layout that this guide is written for works: (This is a side view)



This style of trough and drain was used in Williams System 11 machines and early WPC machines (Addams Family, T2, Hurricane, and a few others).

1. Add the switches

The first step is to add all the switches to the `switches:` section of your config file. Create an entry in your `switches:` section for the drain switch as well as each switch in your trough, like this: (This example has three switches in the trough. Yours may have more or less.)

```
switches:
  s_drain:
    number: 1
  s_trough1:
    number: 2
  s_trough2:
    number: 3
  s_trough3:
    number: 4
```

Note that we configured this switches with numbers `01` through `04`, but you should use the actual switch numbers for your control system that the trough switches are connected to. (See *How to configure "number:" settings* for instructions for each type of control system.)

It makes no difference which switch is which (in terms of whether Switch 1 is on the left side or the right side). Also the actual switch names don't really matter. We use *s_trough1* through *s_trough3* though you can call them *s_ball_trough_1* or *trough_ball_1* or *s_mr_potatohead*.

2. Add the coils

Next, create the entries in your `coils:` section for the drain eject coil and the trough release coil. Again, the names don't matter. We'll call them *c_drain_eject* and *c_trough_release* and enter them like this:

```
coils:
  c_drain_eject:
    number: 3
    default_pulse_ms: 20
  c_trough_release:
    number: 4
    default_pulse_ms: 20
```

Again, the `number:` entries in your config will vary depending on your actual hardware, and again, you can pick whatever name you want for your coil.

You'll also note that we went ahead and entered `default_pulse_ms:` values of 20 which will override the default pulse times of 10ms. It's hard to say at this point what values you'll actually need. You can always adjust this at any time. You can play with the exact values in a bit once we finish getting everything set up.

Note that some trough coils use a shorter pulse to pop the ball into the plunger lane. However, some machines have gates or rotational devices that need to be active for much longer. So having a long pulse time, like `default_pulse_ms: 1000` (for one second) is totally fine. However, if the pulse time is over 255ms, then technically that coil is enabled and disabled versus pulsed, so in that case, you also need to add `allow_enable: true` which tells MPF it's ok to enable this coil for more than 255ms (since 255ms is the maximum pulse time for most platforms).

In other words, a trough release time of 1s would look like this:

```
coils:
  c_trough_release:
    number: 4
    default_pulse_ms: 1000
    allow_enable: true
```

3. Add your "drain" ball device

In MPF, anything that holds and releases a ball is a *ball device*. With this drain/trough setup, there are actually two ball devices—one for the drain and a second for the trough.

Let's add the drain device first, which we'll add to the `ball_devices:` section of your machine config. (If you don't have that section add it now.)

Then in your `ball_devices:` section, create an entry called `bd_drain:`, like this:

```
ball_devices:
    bd_drain:
```

This means that you're creating a ball device called *bd_drain*. We use the preface *bd_* to indicate that this is a ball device which makes it easier when we're referencing them later. Then under your `bd_drain:` entry, you'll start entering the configuration settings for your drain ball device.

- Add `ball_switches: s_drain` which means this device will use the *s_drain* switch to know whether or not this device has a ball.
- Add `eject_coil: c_drain_eject` which is the name of the coil that will eject the ball from the drain.
- Add `eject_targets: bd_trough` which tells MPF that this ball device ejects its balls into the device called *bd_trough*. (We'll create that device in the next step.)
- Add `tags: drain` which tells MPF that balls entering this device mean that a ball has drained from the playfield.

Your drain device configuration should look now look like this:

```
ball_devices:
  bd_drain:
    ball_switches: s_drain
    eject_coil: c_drain_eject
    eject_targets: bd_trough
    tags: drain
```

4. Add your "trough" ball device

Next create a second entry in the `ball_devices:` section called `bd_trough` that will be for the trough device that holds the balls that are ejected from the drain before they're released into the plunger lane.

The configuration is pretty straightforward:

- Add `ball_switches: s_trough1, s_trough2, s_trough3` tells this device that those switches are used to count balls in the trough. (You may have more or less than 3. Also the order of these doesn't matter.
- Add `eject_coil: c_trough_release` which is the name of the coil that will be pulsed to eject the ball from the drain.
- Add `eject_targets: bd_plunger_lane` which tells MPF that this ball device ejects its balls into the device called *bd_plunger_lane*. (We won't actually create the plunger device in this How To guide, but you need to have it, so see the *Plungers & Ball Launch Devices* documentation for full details since there are lots of different types of plungers.
- Add `tags: home, trough` which tells MPF that it's ok to store unused balls here and that it's ok for balls to be here when games start.
- Set `eject_timeouts` to the maximum time the ball can take to return if the eject fails.

Your trough device configuration should look now look like this:

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3
    eject_coil: c_trough_release
    eject_targets: bd_plunger_lane
    tags: home, trough
    eject_timeouts: 3s
```

5. Configure your virtual hardware to start with balls in the trough

While we're talking about the trough, it's probably a good idea to configure MPF so that when you start it in virtual mode (with no physical hardware) that it starts with the trough full of balls. To do this, add a new section to your config file called `virtual_platform_start_active_switches:`. (Sorry this entry name is hilariously long.) As its name implies, *virtual_platform_start_active_switches:* lets you list the names of switches that you want to start in the "active" state when you're running MPF with the virtual platform interfaces.

The reason these only work with the virtual platforms is because if you're running MPF while connected to a physical pinball machine, it doesn't really make sense to tell MPF which switches are active since MPF can read the actual switches from the physical machine. So you can add this section to your config file, but MPF only reads this section when you're running with one of the virtual hardware interfaces. To use it, simply add the section along with a list of the switches you want to start active. For example:

```
virtual_platform_start_active_switches: s_trough1, s_trough2, s_trough3
```

Here's the complete config

```
#config_version=5
switches:
  s_drain:
    number: 1
  s_trough1:
    number: 2
  s_trough2:
    number: 3
  s_trough3:
    number: 4
  s_plunger:
    number: 10
coils:
  c_drain_eject:
    number: 3
    default_pulse_ms: 20
  c_trough_release:
    number: 4
    default_pulse_ms: 20
ball_devices:
  bd_drain:
    ball_switches: s_drain
    eject_coil: c_drain_eject
    eject_targets: bd_trough
    tags: drain
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3
    eject_coil: c_trough_release
    eject_targets: bd_plunger_lane
    tags: home, trough
    eject_timeouts: 3s
  bd_plunger_lane:
```

(continues on next page)

(continued from previous page)

```
    ball_switches: s_plunger
    mechanical_eject: true
    eject_timeouts: 5s
playfields:
  playfield:
    default_source_device: bd_plunger_lane
    tags: default
virtual_platform_start_active_switches: s_trough1, s_trough2, s_trough3
```

What if it doesn't work?

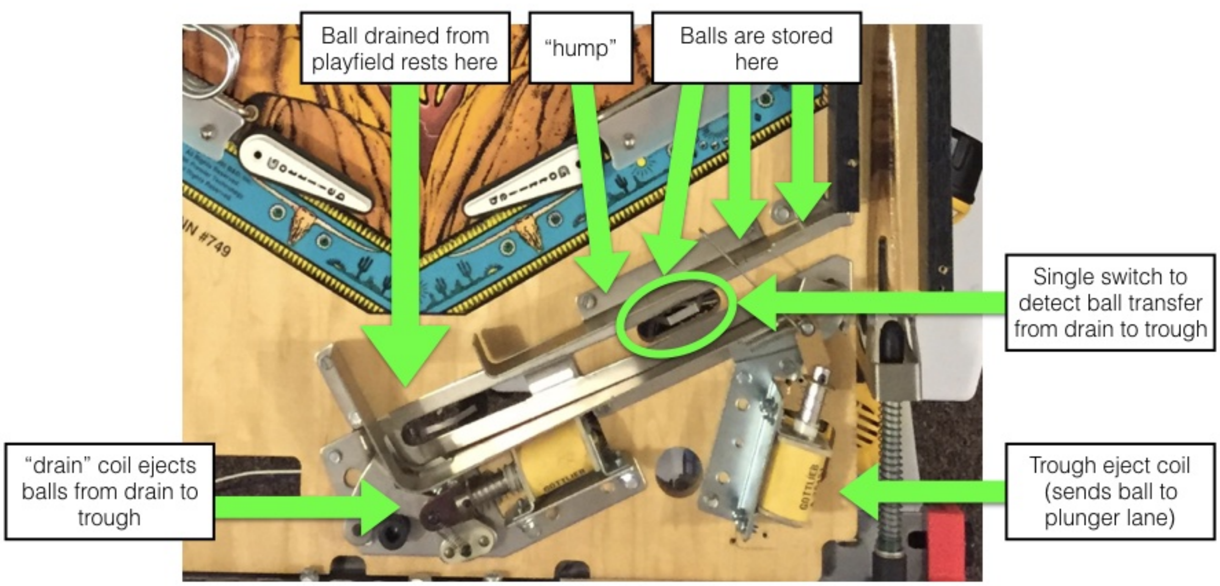
Have a look at our *troubleshooting guide for ball_devices*.

How to configure an older style trough with two coils and only one ball switch

Related Config File Sections
ball_devices:
playfields:

This guide will show you how to configure MPF to use an older-style drain and trough combination that uses two coils (one to eject the ball from the drain hole and a second to release a ball into the plunger lane).

This guide is written for the types of devices that have only have one switch on the trough side, like this example of a Gottlieb System 3 machine (Brooks 'n Dunn):

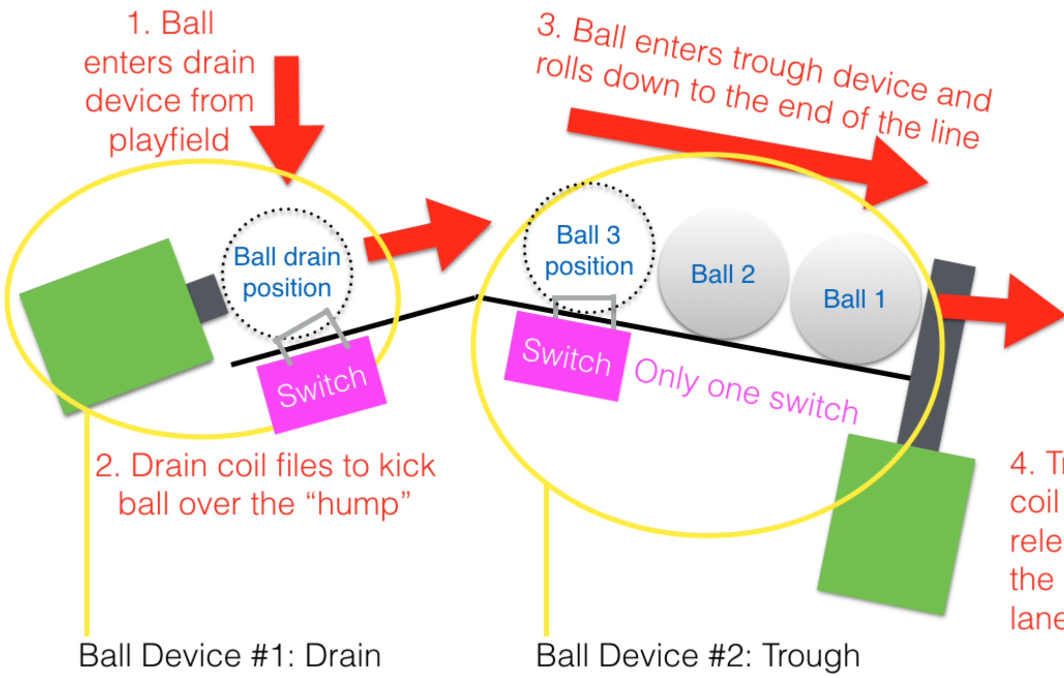


Top View
(apron removed)

If your trough system has multiple switches in the trough (one for each ball), then use *this guide* instead.

In the types of troughs this guide is for, a ball ejected from the drain over the hump into the trough will only momentarily activate the trough switch as the ball rolls by, unless the trough is full, in which case the last ball that goes into it sits on the switch.

The following diagram shows a more clear view of the type of trough system this guide is for: (This is a side view)



1. Add the switches

The first step is to add all the switches to the `switches:` section of your config file. Create an entry in your `switches:` section for the drain switch as well as each switch in your trough, like this: (This example has three switches in the trough. Yours may have more or less.)

```
switches:
  s_drain:
    number: 1
  s_trough_enter:
    number: 2
```

Note that we configured this switches with numbers `01` and `02`, but you should use the actual switch numbers for your control system that the switches are connected to. (See *How to configure "number:" settings* for instructions for each type of control system.)

It makes no difference what the actual switch names are. We use *s_drain* and *s_trough_entry*, though you can call them whatever you want.

2. Add the coils

Next, create the entries in your `coils:` section for the drain eject coil and the trough release coil. Again, the names don't matter. We'll call them *c_drain_eject* and *c_trough_release* and enter them like this:

```
coils:
  c_drain_eject:
    number: 3
    default_pulse_ms: 20
  c_trough_release:
    number: 4
    default_pulse_ms: 20
```

Again, the `number:` entries in your config will vary depending on your actual hardware, and again, you can pick whatever name you want for your coil.

You'll also note that we went ahead and entered `default_pulse_ms:` values of 20 which will override the default pulse times of 10ms. It's hard to say at this point what values you'll actually need. You can always adjust this at any time. You can play with the exact values in a bit once we finish getting everything set up.

Note that some trough coils use a shorter pulse to pop the ball into the plunger lane. However, some machines have gates or rotational devices that need to be active for much longer. However, if the pulse time is over about 50ms, then that coil should be enabled with PWM and disabled versus pulsed, so in that case, you also need to add `default_pulse_power:` which tells MPF it's ok to enable this coil (with 25% hold power in this case).

In other words, a trough with long release time would look like this:

```
coils:
  c_trough_release:
    number: 4
    default_pulse_ms: 20ms
    default_hold_power: 0.25
```

3. Add your "drain" ball device

In MPF, anything that holds and releases a ball is a *ball device*. With this drain/trough setup, there are actually two ball devices—one for the drain and a second for the trough.

Let's add the drain device first, which we'll add to the `ball_devices:` section of your machine config. (If you don't have that section add it now.)

Then in your `ball_devices:` section, create an entry called `bd_drain:`, like this:

```
ball_devices:
    bd_drain:
```

This means that you're creating a ball device called *bd_drain*. We use the preface *bd_* to indicate that this is a ball device which makes it easier when we're referencing them later. Then under your `bd_drain:` entry, you'll start entering the configuration settings for your drain ball device.

- Add `ball_switches: s_drain` which means this device will use the *s_drain* switch to know whether or not this device has a ball.

- Add `eject_coil: c_drain_eject` which is the name of the coil that will eject the ball from the drain.
- Add `eject_targets: bd_trough` which tells MPF that this ball device ejects its balls into the device called *bd_trough*. (We'll create that device in the next step.)
- Add `tags: drain` which tells MPF that balls entering this device mean that a ball has drained from the playfield.
- Set `eject_timeouts` to the maximum time the ball can take to return if the eject fails.

Your drain device configuration should look now look like this:

```
ball_devices:
  bd_drain:
    ball_switches: s_drain
    eject_coil: c_drain_eject
    eject_targets: bd_trough
    tags: drain
    eject_timeouts: 4s
```

4. Add your "trough" ball device

Next create a second entry in the `ball_devices:` section called `bd_trough` that will be for the trough device that holds the balls that are ejected from the drain before they're released into the plunger lane.

The configuration is pretty straightforward:

- Add `entrance_switch: s_trough_enter` which tells MPF which switch is used as the "entrance" switch to this device. (An entrance switch is the switch that's momentarily activated as balls enter this device.)
- Add `entrance_switch_full_timeout: 500ms` which tells MPF that if the entrance switch stays active for more than this amount of time, that means that this device is full.
- Add `ball_capacity: 3` (or whatever the number of balls is that can be stored on the trough side). This tells MPF how many balls are in this device when a ball is sitting on the entrance switch.
- Add `eject_coil: c_trough_release` which is the name of the coil that will be pulsed to eject the ball from the drain.
- Add `eject_targets: bd_plunger_lane` which tells MPF that this ball device ejects its balls into the device called *bd_plunger_lane*. (We won't actually create the plunger device in this How To guide, but you need to have it, so see the *Plungers & Ball Launch Devices* documentation for full details since there are lots of different types of plungers.
- Add `tags: home, trough` which tells MPF that it's ok to store unused balls here and that it's ok for balls to be here when games start.

Your trough device configuration should look now look like this:

```
ball_devices:
  bd_trough:
    entrance_switch: s_trough_enter
    entrance_switch_full_timeout: 500ms
    ball_capacity: 3
```

(continues on next page)

(continued from previous page)

```
    eject_coil: c_trough_release
    eject_targets: bd_plunger_lane
    tags: trough, home
    eject_timeouts: 3s
```

If you need to enable `c_trough_release` for 1s (more than a few ms) it would look like this:

```
ball_devices:
  bd_trough:
    entrance_switch: s_trough_enter
    entrance_switch_full_timeout: 500ms
    ball_capacity: 3
    eject_coil: c_trough_release
    eject_coil_enable_time: 100ms
    eject_targets: bd_plunger_lane
    tags: trough, home
    eject_timeouts: 3s
```

5. Configure the balls installed

One of the downsides of only having one switch in the trough is that if that switch is not active, then MPF doesn't actually know how many balls are in it. (In the example diagram at the beginning of this guide where the trough can hold three balls, if that trough entry switch is not active, then there could be zero, 1, or 2 balls in the trough.)

MPF is able to keep track of how many balls are in the trough by tracking balls entered versus balls released. However when MPF starts up, if that entrance switch isn't active, then it won't know how many balls are there.

There's a setting in the machine config called `machine:balls_installed:` that tells MPF how many actual balls are installed in the machine. So when MPF starts, it can count up all the balls in all the devices and see if they're all there or if any are missing. Since that's a bit tricky with the single switch in the trough, you telling MPF how many total balls are installed in the machine help it know what to do if that entrance switch isn't active when MPF starts up.

Here's an example from the machine config:

```
machine:
  balls_installed: 4
```

6. Configure your virtual hardware to start with balls in the trough

While we're talking about the trough, it's probably a good idea to configure MPF so that when you start it in virtual mode (with no physical hardware) that it starts with the trough full of balls. To do this, add a new section to your config file called `virtual_platform_start_active_switches:`. (Sorry this entry name is hilariously long.) As its name implies, *virtual_platform_start_active_switches:* lets you list the names of switches that you want to start in the "active" state when you're running MPF with the virtual platform interfaces.

The reason these only work with the virtual platforms is because if you're running MPF while connected to a physical pinball machine, it doesn't really make sense to tell MPF which switches are active since MPF can read the actual switches from the physical machine. So you can add this section to your config file, but MPF only reads this section when you're running with one of the virtual hardware interfaces. To use it, simply add the section along with a list of the switches you want to start active. For example:

```
virtual_platform_start_active_switches: s_trough_enter
```

Here's the complete config

```
switches:
  s_drain:
    number: 01
  s_trough_enter:
    number: 02
  s_plunger:
    number: 10
coils:
  c_drain_eject:
    number: 03
    default_pulse_ms: 20
  c_trough_release:
    number: 04
    default_pulse_ms: 20
ball_devices:
  bd_drain:
    ball_switches: s_drain
    eject_coil: c_drain_eject
    eject_targets: bd_trough
    tags: drain
    eject_timeouts: 4s
  bd_trough:
    entrance_switch: s_trough_enter
    entrance_switch_full_timeout: 500ms
    ball_capacity: 3
    eject_coil: c_trough_release
    eject_targets: bd_plunger
    tags: trough, home
    eject_timeouts: 3s
  bd_plunger:
    ball_switches: s_plunger
    mechanical_eject: true
    eject_timeouts: 5s
playfields:
  playfield:
    default_source_device: bd_plunger
    tags: default
machine:
  balls_installed: 4
virtual_platform_start_active_switches: s_trough_enter
```

What if it doesn't work?

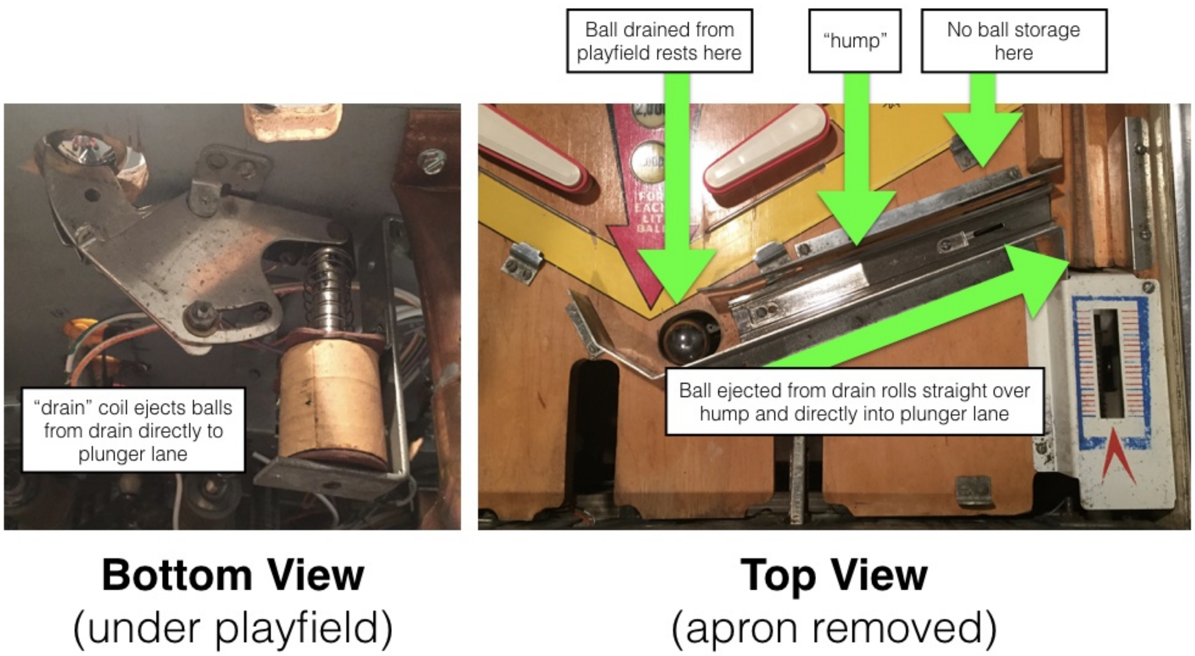
Have a look at our *troubleshooting guide for ball_devices*.

How to configure a classic single-ball trough

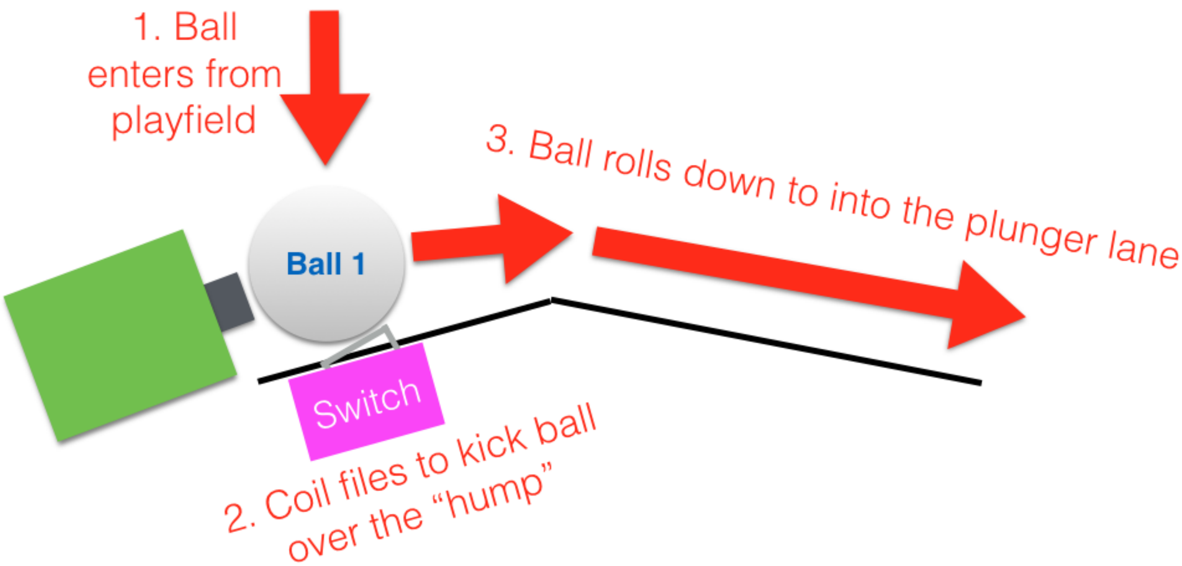
Related Config File Sections
ball_devices:
playfields:

This guide will show you how to configure MPF to use an older-style single ball drain. This is the type of configuration that most (all?) single-ball machines use, from EM machines of the 1950s through electronic single ball machines of the early 1980s.

Here's an example from a Gottlieb Big Shot (1974 EM):



And here's a diagram which shows this a bit more clearly: (This is a side view)



We assume that your machine has a shooter lane switch. If that is not the case see *How to configure a classic single-ball trough without shooter lane*.

1. Add the drain and plunger switch

The first step is to add the drain and plunger switches to the `switches:` section of your machine config file.

```
switches:
  s_drain:
    number: 01
  s_plunger:
    number: 02
```

Note that we configured those switches with number `01` and `02`, but you should use the actual switch number for your control system that the switch is connected to. (See *How to configure "number:" settings* for instructions for each type of control system.)

2. Add the eject coil

Next, create the entry in your `coils:` section for the drain eject coil. Again, the name doesn't matter. We'll call it *c_drain_eject* and enter it like this:

```
coils:
  c_drain_eject:
    number: 03
    default_pulse_ms: 20
```

Again, the `number:` entry in your config will vary depending on your actual hardware, and again, you can pick whatever name you want for your coil.

You'll also note that we went ahead and entered a `default_pulse_ms:` value of 20 which will override the default pulse times of 10ms. It's hard to say at this point what values you'll actually need. You can always adjust this at any time. You can play with the exact values in a bit once we finish getting everything set up.

3. Add your "drain" ball device

In MPF, anything that holds and releases a ball is a *ball device*. So in your `ball_devices:` section, create an entry called `bd_drain:` like this: (If you don't have that section add it now.)

```
ball_devices:
    bd_drain:
```

This means that you're creating a ball device called *bd_drain*. We use the preface *bd_* to indicate that this is a ball device which makes it easier when we're referencing them later. Then under your `bd_drain:` entry, you'll start entering the configuration settings for your drain ball device.

- Add `ball_switches: s_drain` which means this device will use the *s_drain* switch to know whether or not this device has a ball.
- Add `eject_coil: c_drain_eject` which is the name of the coil that will eject the ball from the drain.
- Add `eject_targets: bd_plunger_lane` which tells MPF that this ball device ejects its balls into the device called *bd_plunger_lane*. (We won't actually create the plunger device in this How To guide, but you need to have it, so see the *Plungers & Ball Launch Devices* documentation for full details since there are lots of different types of plungers.
- Add `tags: drain, home, trough` which tells MPF that balls entering this device mean that a ball has drained from the playfield, that it's ok to start a game with a ball here, and that this device is used to store unused balls.
- Set *eject_timeouts* to the maximum time the ball can take to return if the eject fails.

Your drain device configuration should look now look like this:

```
ball_devices:
  bd_drain:
    ball_switches: s_drain
    eject_coil: c_drain_eject
    eject_targets: bd_plunger_lane
    tags: drain, home, trough
    eject_timeouts: 3s
```

4. Add your "plunger" ball device

We also add the plunger as ball_device `bd_plunger_lane`:

```
ball_devices:
  bd_plunger_lane:
    ball_switches: s_plunger
    mechanical_eject: true
    eject_timeouts: 5s
```

5. Configure your virtual hardware to start with balls in the trough

While we're talking about the trough, it's probably a good idea to configure MPF so that when you start it in virtual mode (with no physical hardware) that it starts with the trough full of balls. To do this, add a new section to your config file called `virtual_platform_start_active_switches:`. (Sorry this entry name is hilariously long.) As its name implies, *virtual_platform_start_active_switches:* lets you list the names of switches that you want to start in the "active" state when you're running MPF with the virtual platform interfaces.

The reason these only work with the virtual platforms is because if you're running MPF while connected to a physical pinball machine, it doesn't really make sense to tell MPF which switches are active since MPF can read the actual switches from the physical machine. So you can add this section to your config file, but MPF only reads this section when you're running with one of the virtual hardware interfaces. To use it, simply add the section along with a list of the switches you want to start active. For example:

```
virtual_platform_start_active_switches: s_drain
```

Here's the complete config

```
#config_version=5
switches:
  s_drain:
    number: 01
  s_plunger:
    number: 02
coils:
  c_drain_eject:
    number: 03
    default_pulse_ms: 20
ball_devices:
  bd_drain:
    ball_switches: s_drain
    eject_coil: c_drain_eject
    eject_targets: bd_plunger_lane
    tags: drain, home, trough
    eject_timeouts: 3s
  bd_plunger_lane:
    ball_switches: s_plunger
    mechanical_eject: true
    eject_timeouts: 5s
playfields:
  playfield:
    default_source_device: bd_plunger_lane
    tags: default
virtual_platform_start_active_switches: s_drain
```

What if it did not work?

Have a look at our *troubleshooting guide for ball_devices*.

How to configure a classic single-ball trough without shooter lane

Related Config File Sections
ball_devices:
playfields:

This guide will show you how to configure MPF to use an older-style single ball drain without shooter lane. This is the type of configuration that some single-ball machines use, from EM machines of the 1950s through electronic single ball machines of the early 1980s.

Here's an example from a Gottlieb Playball (1971 EM):

1. Add the drain switch

The first step is to add the drain switch to the `switches:` section of your machine config file.

```
switches:
  s_drain:
    number: 01
```

Note that we configured this switches with number `01`, but you should use the actual switch number for your control system that the switch is connected to. (See *How to configure "number:" settings* for instructions for each type of control system.)

2. Add the eject coil

Next, create the entry in your `coils:` section for the drain eject coil. Again, the name doesn't matter. We'll call it *c_drain_eject* and enter it like this:

```
coils:
  c_drain_eject:
    number: 03
    default_pulse_ms: 20
```

Again, the `number:` entry in your config will vary depending on your actual hardware, and again, you can pick whatever name you want for your coil.

You'll also note that we went ahead and entered a `default_pulse_ms:` value of 20 which will override the default pulse times of 10ms. It's hard to say at this point what values you'll actually need. You can always adjust this at any time. You can play with the exact values in a bit once we finish getting everything set up.

3. Add your "drain" ball device

In MPF, anything that holds and releases a ball is a *ball device*. So in your `ball_devices:` section, create an entry called `bd_drain:` like this: (If you don't have that section add it now.)

```
ball_devices:
    bd_drain:
```

This means that you're creating a ball device called `bd_drain`. We use the preface `bd_` to indicate that this is a ball device which makes it easier when we're referencing them later. Then under your `bd_drain:` entry, you'll start entering the configuration settings for your drain ball device.

- Add `ball_switches: s_drain` which means this device will use the *s_drain* switch to know whether or not this device has a ball.
- Add `eject_coil: c_drain_eject` which is the name of the coil that will eject the ball from the drain.
- Add `tags: drain, home, trough` which tells MPF that balls entering this device mean that a ball has drained from the playfield, that it's ok to start a game with a ball here, and that this device is used to store unused balls.
- Set `eject_timeouts` to the maximum time the ball can take to return if the eject fails.

Your drain device configuration should look now look like this:

```
ball_devices:
  bd_drain:
    ball_switches: s_drain
    eject_coil: c_drain_eject
    tags: drain, home, trough
    eject_timeouts: 3s
```

4. Add the trough as default_source_device

Normally you would use your plunger device as source device for your playfield. But since there is no plunger lane, that means we have to go back to the trough ball device and use it as source device. Therefore, you need to add your trough ball device as `default_source_device` to your playfield to tell MPF that this ball device is used to add a new ball into play.

To do that, add your trough device as `default_source_device` in the default `playfield`, like this:

```
playfields:
  playfield:
    default_source_device: bd_drain
    tags: default
```

Then when MPF needs to add a live ball into play, it will eject a ball from the trough and you're all set!

5. Configure your virtual hardware to start with balls in the trough

While we're talking about the trough, it's probably a good idea to configure MPF so that when you start it in virtual mode (with no physical hardware) that it starts with the trough full of balls. To do this, add a new section to your config file called `virtual_platform_start_active_switches:`. (Sorry this entry name is hilariously long.) As its name implies, *virtual_platform_start_active_switches:* lets you list the names of switches that you want to start in the "active" state when you're running MPF with the virtual platform interfaces.

The reason these only work with the virtual platforms is because if you're running MPF while connected to a physical pinball machine, it doesn't really make sense to tell MPF which switches are active since MPF can read the actual switches from the physical machine. So you can add this section to your config file, but MPF only reads this section when you're running with one of the virtual hardware interfaces. To use it, simply add the section along with a list of the switches you want to start active. For example:

```
virtual_platform_start_active_switches: s_drain
```

Here's the complete config

```
#config_version=5
switches:
  s_drain:
    number: 01
coils:
  c_drain_eject:
    number: 03
    default_pulse_ms: 20
ball_devices:
  bd_drain:
    ball_switches: s_drain
    eject_coil: c_drain_eject
    tags: drain, home, trough
    eject_timeouts: 3s
playfields:
```

(continues on next page)

(continued from previous page)

```
  playfield:
    default_source_device: bd_drain
    tags: default
virtual_platform_start_active_switches: s_drain
```

What if it did not work?

Have a look at our *troubleshooting guide for ball_devices*.

Using the Stern Spike Trough

Related Config File Sections
ball_devices:
playfields:
spi_bit_bang:
digital_outputs:

Unlike other troughs the Stern Spike trough contains an 74HCT165 chip and is interfaced via SPI. This is a problem if your platform is not using SPI to read switches (which most platforms are not). If you are on *Stern Spike* then just configure your trough as described in *How to configure a modern trough with opto switches*.

Note: While the Stern Spike trough works with other platforms we do not recommend to buy it if you are not using the Stern Spike platform. Instead, if you did not yet buy a trough buy one with normal switches or optos (unless you are using Stern Spike). This will make your life easier.

Part numbers:

- Transmitter: 520-5344-00
- Receiver: 520-5345-00/520-5345-01

Config (if you are not on Stern Spike):

If you got a Stern Spike trough but are not using *Stern Spike* you can use our *SPI Bit Bang platform* to read the switches of your trough:

```
hardware:
  platform: your_platform, spi_bit_bang      # add your platform first here
spi_bit_bang:
  miso_pin: s_miso
  cs_pin: o_cs
  clock_pin: o_clock
digital_outputs:
  o_cs:
```

(continues on next page)

(continued from previous page)

```
    number: 1       # adjust this for your platform
    type: driver
  o_clock:
    number: 2       # adjust this for your platform
    type: driver
switches:
  s_trough0:
    number: 0
    platform: spi_bit_bang
  s_trough1:
    number: 1
    platform: spi_bit_bang
  s_trough2:
    number: 2
    platform: spi_bit_bang
  s_trough3:
    number: 3
    platform: spi_bit_bang
  s_trough4:
    number: 4
    platform: spi_bit_bang
  s_trough5:
    number: 5
    platform: spi_bit_bang
  s_trough6:
    number: 6
    platform: spi_bit_bang
  s_trough_jam:     # this might be also number 0
    number: 7
    platform: spi_bit_bang
  s_miso:
    number: 10      # adjust this for your platform
  s_plunger:
    number: 11      # adjust this for your platform
# the following is the same as in the "modern trough with opto switches" tutorial
coils:
  c_trough_eject:
    number: 4
    default_pulse_ms: 20
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough5, s_trough6, s_trough_jam
    eject_coil: c_trough_eject
    tags: trough, home, drain
    jam_switch: s_trough_jam
    eject_coil_jam_pulse: 15ms
    eject_targets: bd_plunger
    eject_timeouts: 3s
  bd_plunger:
    ball_switches: s_plunger
    mechanical_eject: true
    eject_timeouts: 5s
playfields:
  playfield:
```

(continues on next page)

(continued from previous page)

```
    default_source_device: bd_plunger
    tags: default
```

What if it doesn't work?

Have a look at our *troubleshooting guide for ball_devices*.

Let us know in the forum if you are missing a mech in MPF.

CHAPTER 8

Game Logic

Most (potentially all) of your game logic can be configured in the MPF config files. For classical language programmers new to MPF, an introduction to how the framework handles logical decisions may be helpful. All game logic is tied to event posts. Mostly, this is achieved through config file programming of timers, shots, counters, multiballs, accruals, etc. . . . These prebuilt modules (listed below) listen for events to be posted then read the state of the hardware and/or perform manipulations on player or device variables. In turn these modules issue their own event posts which drive the behavior or other modules and devices to start and stop modes, control diverts, set bonus multipliers and everything else game related.

A question beginners may have is “How do I tell MPF to perform an action when two or more conditions are met simultaneously?” In an event driven framework this is not the correct way to conseptualize the logic. Again, nothing game related happens without being driven by a posted event. Because events only exist as a descrete moments in time, it does not work to attempt (pseudocode) logic such as IF event1 and event2 then post event3. Nevertheless, MPF provides a flexible and robust mechanism for performing logic on events. This is where Conditional Events come in.

In brief, the the way conditional events work is by telling MPF to process a particular event if and only if additional conditions are met. These conditions (listed inside curly brackets) can relate to player variables (such as score) machine variables (such as credit) or device variables (such as timer ticks or number of balls locked). See <conditional/index> for specific examples.

With this flexibility in mind, Here is a list of pre-built game logic modules containing the description, how to guides, links to tutorials, event listings, and configuration

Note: Most of the “How To” guides for these sections still need to be written.

Achievements

Related Config File Sections
achievements:

MPF uses "achievements" to track major goals that a player must achieve throughout the progression of a game. Achievements typically have an associated light or LED on the playfield (though not always), and they're tracked separately per player.

The biggest use for achievements is for modes, where you have a bunch of modes in a machine which each have a light, and as you complete the modes, the light turns on. (In many cases the lights/LEDs associated with achievements have multiple states, for example, they're "off" when not complete, "flashing" when active, "on" when complete, etc.)

Here are some examples from real machines that would map to "achievements" in MPF:

- **Attack from Mars:**
  - The countries (France, Germany, Italy, England, USA)
  - The Capture inserts (Capture 1, Capture 2, Capture 3)
  - The Big -O- Beam inserts (1, 2, and 3)
  - The Atomic Blaster inserts (1, 2, and 3)
  - The Blue circles to Rule The Universe (Super Jackpot, Super Jets, Martian Attack Multiball, Total Annihilation, Conquer Mars, and 5-way Combo)
- **Indiana Jones: The Pinball Adventure:**
  - The Modes inserts (Streets of Cairo, Well of Souls, Monkey Brains, etc.)
- **The Addams Family:**
  - Mansion Modes (Raise the Dead, Hit Cousin It, Mamushka, etc.)
- **Star Trek: The Next Generation:**
  - Missions (Time Rift, Asteroid Threat, Rescue, Q's Challenge, etc.)
- **Red & Ted's Road Show:**
  - The cities on the Map (each city is an achievement)
  - The wheel (Lunch Time, Flying Rocks, Big Blast, Special, etc.)

You can have as many achievements as you want in your machine, and you can re-use the same lights/LEDs for different achievements in different modes. (For example, you might have red arrow inserts that turn on and off to highlight shots in your base mode, but then you might have a timed mode where those inserts are mapped to achievements and they're all lit, and they go out as they're hit.)

You can also group individual achievements into "achievement groups". This is useful for tracking when all the achievements in the group have been complete (e.g. to light a wizard mode). You can also use achievement groups to "rotate" lit achievements (e.g. every slingshot hit changes the achievement that's flashing, but it only rotates through incomplete achievements.)

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for achievements is `device.achievements.<name>`.

state The string name of the state this achievement is in. Options will be one of the following: *disabled*, *enabled*, *started*, *stopped*, or *completed*. If this achievement is in a mode that has not been started yet, then its state will be an empty string.

selected boolean (*true* or *false*)

Related How To guides

- *Recipe: The Addams Family Mansion Awards*

Related Events

- *achievement_(name)_changed_state*
- *achievement_(name)_state_(state)*
- Plus any custom events as defined in the achievement's configuration in your config files.

Achievement Groups

Related Config File Sections
achievement_groups:

Achievement groups are used to group together individual achievements.

If you look at the real-world examples we used in the achievements documentation, each of the entries in that list is an achievement "group" that's made up of individual achievements.

For example, in The Addams Family, the mansion awards would be individual achievements, for example:

- 9 Mil
- 6 Mil
- 3 Mil

- Thing
- Quick Multiball
- Grave Yard at Max
- Raise the Dead
- Etc.

Each of those individual achievements has a state (enabled, started, completed, etc.)

If you were building a config for *The Addams Family (TAF)* with MPF, you would create an achievement group called "Mansion Awards", and then you would add the individual achievements to that group.

The achievement group will let you perform group-level actions on the achievements in the group. For example:

- Randomly select one of the incomplete achievements (so you can flash that achievement's light to indicate it's selected).
- Change which achievement is selected. (In *TAF*, each hit to a pop bumper changes the lit achievement, so you'd configure your achievement group to pick a new achievement when the pop bumper hit event was posted.)
- Post an event when all achievements are complete (to start a wizard mode, etc.)
- Post a "start" event for whichever achievement is lit (In *TAF*, you shoot the lit electric chair or the swamp to start the flashing achievement.)

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for achievement groups is `device.achievement_groups.<name>`.

enabled Boolean (true/false) as to whether this achievement group is enabled.

selected_member The achievement in the group that is currently in the selected state, or *None* if no achievement is selected.

Related How To guides

- *Recipe: The Addams Family Mansion Awards*

Related Events

- Custom events as defined in the achievement's configuration in your config files.

Ball Holds

Related Config File Sections
ball_holds:

MPF's *ball holds* are used to temporarily hold a ball that has entered a *Ball Devices* while something else happens.

The most common use cases are to hold a ball while you play a show, or while a video mode is going on. Ball holds do not affect the balls in play count, and if all other balls drain while a ball hold is in progress, the players ball does not end.

Ball holds are *not* used to lock balls for multiball. (See the *multiball_locks* device for that).

You can have lots of different ball holds in your game, typically configured per mode.

Video about ball locks and multiballs:

https://youtu.be/2mFkgIlksC4

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for ball holds is `device.ball_holds.<name>`.

balls_held The number of balls this ball hold is currently holding

enabled Boolean (true/false) which shows whether this ball hold is enabled.

Related How To guides

- *Using ball_holds for a mystery award*

Related Events

- *ball_hold_(name)_held_ball*
- *ball_hold_(name)_full*
- *ball_hold_(name)_balls_released*

Ball Locks

Related Config File Sections
multiball_locks:
ball_holds:

MPF supports ball locks which are used to hold a ball that has entered a *Ball Devices*. To separate use-cases MPF supports two cases of ball locks:

- *Multiball_locks* which lock balls for a multiball. Locked balls are no longer in play (i.e. deducted from ball count).
- *Ball_holds* which only hold balls temporarily. This is used to play animations or stop the ball during a video mode. Those balls are technically still in play.

Ball Saves

Related Config File Sections
ball_saves:

MPF uses *ball saves* to automatically re-serve a ball that has drained. (Essentially this means the ball drain doesn't count.)

Ball saves are typically used in several scenarios:

- Give the player their ball back if they drain right after their ball starts.
- Give the player their ball back if there's a particularly wicked shot that tends to drain which the game designers feel bad about. (You should avoid this if possible, and instead, as Lyman Sheets would say, "Fix your f-ing game layout!")
- Use to make a timed mode where the player has unlimited drains.
- Etc.

You can configure ball saves to have various start and stop events and timers, and you can configure multiple ones in different modes that do different things.

This is an example:

```
ball_saves:
  random_ball_save:
    active_time: 5s
    hurry_up_time: 2s
    grace_period: 2s
    enable_events: event_on_dangerous_action
    auto_launch: true
    balls_to_save: 1
```

When `event_on_dangerous_action` is posted the ball save will be active for 5s `active_time` + 2s `grace_period` = 7s. Hurry up will start after 5s `active_time` - 2s `hurry_up_time` = 3s.

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for ball saves is `device.ball_saves.<name>`.

enabled Boolean (true/false) which shows whether this ball hold is enabled.

saves_remaining How many balls saves are remaining.

state String value of the state of this ball save. Values will be one of the following: *enabled*, *disabled*, *hurry_up*, or *grace_period*.

timer_started Boolean (true/false) which shows whether the timer is started.

Related How To guides

- *Ball save at ball start*

Center Post Ball Save

Some machines have a mechanical ball save called center post. It pops up between the flippers and prevents the ball from draining.

Video about center posts:

https://youtu.be/eR0C5ft546c

To use it in MPF we reuse a diverter. A simple *coil_player:* would work as well but then we would have to reimplement ball search and service mode logic. The diverter will already implement all that for us.

This is an example:

```
# in your machine config
coils:
  c_ball_save_post_up:
    number: 1-10  # yours might be different
    default_pulse_ms: 15
  c_ball_save_post_down:
    number: 1-15  # yours might be different
    default_pulse_ms: 15
lights:
  ball_saver:
    number:
diverters:
  ball_save_post:
    activation_coil: c_ball_save_post_up
    deactivation_coil: c_ball_save_post_down
    activate_events: ball_save_post_up
    deactivate_events: ball_save_post_down
    enable_events: ball_started
    type: pulse
##! mode: base
# in base mode
event_player:
  ball_save_default_timer_start:
    - ball_save_post_up
  ball_save_default_disabled:
    - ball_save_post_down
ball_saves:
  default:
    active_time: 10s
    grace_period: 2s
    hurry_up_time: 5s
    enable_events: mode_base_started
    timer_start_events: balldevice_bd_plunger_ball_eject_success
    disable_events: ball_will_end
    auto_launch: true
    balls_to_save: 1
    early_ball_save_events: s_right_outlane_active, s_left_outlane_active
show_player:
  ball_save_default_timer_start:
    ball_save_show:
      action: play
      speed: 5
  ball_save_default_hurry_up:
    ball_save_show:
      action: play
      speed: 10
  ball_save_default_disabled:
    ball_save_show:
      action: stop
shows:
  ball_save_show:
    - time: 0
      lights:
        ball_saver:
          color: black
```

(continues on next page)

(continued from previous page)

```
  - time: '+1'
    lights:
      ball_saver:
        color: red
```

Related Events

- *ball_save_(name)_enabled*
- *ball_save_(name)_disabled*
- *ball_save_(name)_timer_start*
- *ball_save_(name)_hurry_up*
- *ball_save_(name)_grace_period*
- *ball_save_(name)_saving_ball*

Ball Search

Related Config File Sections
ball_devices:
switches:
playfields:
example ball_search

- *Related How To guides*
- *Related Events*

Note: Ball search is off by default in MPF because it might hurt users not expecting it. In a prototype game it might trigger quite frequently and coils can seriously injure humans. To turn it on follow *How to configure Ball Search*.

MPF contains ball search functionality which is used to try to dislodge a stuck ball if MPF thinks there's a ball loose on the playfield but it hasn't hit any playfield switches in awhile and the player is not holding the flipper button in.

Ball searching in MPF has multiple "rounds", with the early rounds doing a simple search that doesn't screw anything up (like firing pop bumpers and pulsing eject coils from ball devices that don't contain any balls), but after a few rounds of that, if it still hasn't found the ball, it can start to to things like resetting drop targets.

Eventually MPF will give up and mark the ball as lost and kick a new ball into play.

Everything is fully configurable, including the timeouts, the order devices are searched, the number of rounds, etc.

Ball search in MPF is fairly automatic. It's enabled when MPF thinks that balls are on the playfield, and disabled when no balls are free. (This means that even when a machine tilts, ball search is still active until the balls drain, etc.)

Related How To guides

- *How to configure Ball Search*

Related Events

- *ball_search_failed*
- *ball_search_started*
- *ball_search_stopped*
- *flipper_cradle*
- *flipper_cradle_release*

How to configure Ball Search

To enable ball search set *enable_ball_search* to True for your playfield(s). In most cases, this is as simple as this:

```
playfields:
  playfield:
    enable_ball_search: true
```

Ball search will run in multiple phases with increasing intensity (phase 1 to 3) and give up afterwards. To change the timeout before ball search starts when no ball was seen by MPF, change *ball-search-timeout*. Similarly, *ball-search-interval* determines the delay between coil fires during search. You can further configure ball search per *playfield*.

Coils are included indirectly using their devices. Most devices allow you to configure their order in ball search using the *ball_search_order* attribute (see the *example ball_search*). By default flippers are not included in ball search. However, you might want to enable it for upper playfield flippers:

```
flippers:
  f_upper_flipper_left:
    ball_search_order: 15
    include_in_ball_search: true
    main_coil: c_flipper_left
    activation_switch: s_flipper_left
```

Make sure to include the tag *playfield_active* in all playfield switches which are not bound to devices. For instance do not put that tag into your plunger switch but put it to target, inlane and outlane switches.

If you want to pulse a standalone coil which is not bound to any device, you can use pulse_events on ball_search_phase_x_searches (replace x with phase 1 to 3).

Ball Start and End Behaviour

There are multiple ways to play show/lights/sounds during ball start or ending.

Triggering actions without delay on ball start

During game start (see *Flowcharts* for details) you can trigger shows/lights or any other player on the *ball_started* event (see *Ball Start Sequence*). This will not delay the ball start and the ball will eject instantly.

Shows/Lights

This might be useful to start music, flash some lights or start background shows:

```
##! mode: my_mode
# in your mode
mode:
  start_events: ball_started
show_player:
  mode_my_mode_started:
    short_start_show:
      loops: 0
shows:
  short_start_show:
    - duration: .5s
      # add your show here
```

This can simply be embedded in any mode (e.g. in your base mode).

Ball Save

It is also very common to start a ball save on eject:

```
playfields:
  playfield:
    default_source_device: bd_plunger
lights:
  l_ball_save:
    number:
switches:
  s_plunger:
    number:
coils:
  c_eject:
    number:
ball_devices:
  bd_plunger:
    eject_coil: c_eject
    ball_switches: s_plunger
    tags: home, trough, drain
    eject_timeouts: 1s
```

(continues on next page)

(continued from previous page)

```
##! mode: my_mode
# in your mode
mode:
  start_events: ball_started
ball_saves:
  ball_save_ball_save:
    active_time: 10s
    hurry_up_time: 3s
    timer_start_events: balldevice_bd_plunger_ejecting_ball
    auto_launch: true
    balls_to_save: 1
show_player:
  ball_save_ball_save_ball_save_timer_start:
    flash_color:
      key: ball_save
      speed: 2
      show_tokens:
        lights: l_ball_save
        color: orange
  ball_save_ball_save_ball_save_hurry_up:
    flash_color:
      key: ball_save
      speed: 4
      show_tokens:
        lights: l_ball_save
        color: orange
  ball_save_ball_save_ball_save_disabled:
    ball_save: stop
```

The mode will start on `ball_started`. It will enable a *ball save* on mode start and start a timer once the plunger ejects the ball. This will also work with mechanical eject. Once the timer is active the shoot again led `l_ball_save` will flash. During the hurry up (last 2s) it will flash faster and turn off afterwards.

Triggering simple actions without delay on ball end

Similarly, you can trigger events on ball end using the `ball_ended` event (see *Ball End Sequence* for details). Unfortunately, normal game modes will stop on ball end and you will never see the `ball_ended` event in a game mode. This approach will not delay the ball end and the next ball might eject in the meantime. Use it for very short sounds or light flashes:

```
##! mode: my_mode
# in your mode
mode:
  start_events: ball_ending
  stop_events: end_show_done
  game_mode: false
show_player:
  mode_my_mode_started:
    short_stop_show:
      loops: 0
      events_when_completed: end_show_done
```

(continues on next page)

(continued from previous page)

```
shows:
  short_stop_show:
    - duration: 2s
      # add your show here
```

Delaying ball start and end

To delay start and end of a ball use the following mode. It uses a *queue_relay_player:* to delay `ball_starting` and `ball_ending` for the duration of a show. This can be used to show longer sequences and delaying the game flow in the meantime:

```
##! mode: my_mode
# in your mode
mode:
  start_events: ball_will_start     # in normal mode use ball_started instead
  priority: 200
queue_relay_player:
  ball_starting:
    post: start_ball_starting_show
    wait_for: mode_ball_starting_show_ended
  ball_ending:
    post: start_ball_ending_show
    wait_for: mode_ball_ending_show_ended
show_player:
  flipper_cancel:
    ball_starting_show: stop
    ball_ending_show: stop
  start_ball_starting_show:
    ball_starting_show:
      loops: 0
      events_when_stopped: mode_ball_starting_show_ended
  start_ball_ending_show:
    ball_ending_show:
      loops: 0
      events_when_stopped: mode_ball_ending_show_ended
shows:
  ball_starting_show:
    - duration: 5s
  ball_ending_show:
    - duration: 5s
```

Both shows can be canceled using both flippers which will post the `flipper_cancel` event. Remove that *show_player* entry if you don't want that. See the *flipper mech* documentation for details about the `flipper_cancel` event.

You can combine this with conditional variables to only delay the first ball. E.g. use `ball_starting{ball==1 and not is_extra_ball}` to only delay the first ball (excluding extra balls). Similarly, you can use `ball_starting{is_extra_ball}` to delay any extra ball start and show some animations there.

More examples

See *How to design a game in MPF using Modes* and *Game End Modes* in particular for more examples.

Ball Tracking

Keeping track of where all the balls are at any given time is a big part of a pinball. There are four components that make up MPF's ball tracking and management system:

- The Ball Controller, which is a core MPF module that manages everything.
- Individual *Ball Devices* (troughs, locks, etc.) which track how many balls they're currently holding, request new balls, eject balls, etc.
- The *Playfields* device which is a special type of ball device that tracks how many balls are loose on the playfield at any given time.
- Individual *Diverters* which are integral in routing balls to devices that request them.

These four components are active at all times—regardless of whether or not a game is in progress. In other words, if MPF is running, it's tracking balls.

Note that tracking the number of balls on a playfield is somewhat complex. See the *How MPF tracks the number of balls on a playfield* guide for important details about how this works in MPF.

End of Ball Bonus

MPF contains a built-in end of ball bonus mode which you can use to calculate and display a player's bonus score when they drain a ball.

The built-in bonus mode can manage bonus scoring, multipliers, awarding points based on any player variables, and other "standard" things. You can also extend and enhance it if you have specific requirements that aren't covered by the built-in mode.

Related How To guides
How to configure End of Ball Bonus
How to design a game in MPF using Modes

Overview of Bonus Mode

The built-in bonus mode will automatically handle the following steps when it is enabled:

- Pause the game when the ball ends in order to show the bonus awards
- Calculate the score for each bonus entry in the `bonus_entries:` list
- Post an event for each bonus entry with a delay between each event
- Skip events for bonus entries with a zero score (by default, can be overridden)
- Post an event for the subtotal of all bonuses awarded
- Post an event for a total bonus multiplier (if present)
- Post an event for the total of all bonuses awarded

- Add the total bonus award to the player's score
- Start the next ball after all bonuses have been awarded

See the *How to Configure End of Ball Bonus* guide for instructions on enabling bonus mode.

Calculating Points for Bonus Awards

Each award entry will calculate a bonus score based on the *score* value of the entry. If provided, the *player_score_entry* value will be multiplied by the *score*. This makes it very easy to award, for example, 200 points for every time the player captured a castle (tracked by the player variable "castles_captured").

```
##! mode: bonus
#config_version=5
mode_settings:
  bonus_entries:
    - event: bonus_castles
      score: 200
      player_score_entry: castles_captured
```

For advanced score calculation, the *score* value can utilize all of MPF's *dynamic and placeholder variables*.

```
##! mode: bonus
#config_version=5
mode_settings:
  bonus_entries:
    - event: bonus_minerals
      score: (current_player.platinum + current_player.iridium) / 100
    - event: bonus_dropbanks
      score: device.counters.dropbank_completions.value * 20
```

The calculated score is included in the posted event for displaying on a slide, and the score is automatically added to the current player's `score` value.

Showing Slides for Bonus Awards

Each award in the `bonus_entries:` setting requires an *event* value, which is the name of the event that MPF will post when that award is calculated. You can use these events to show slides, play sounds, and anything else. The events will post sequentially at the interval specified by the `display_delay_ms` setting.

After all awards in the entries list have been posted, a final *bonus_total* event will post with the total amount awarded as bonus. This event can be used to show a final slide.

```
#config_version=5
slide_player:
  mode_bonus_started: bonus_start_slide
  bonus_minerals: bonus_minerals_slide
  bonus_dropbanks: bonus_dropbanks_slide
  bonus_total: bonus_total_slide
```

Bonus Multipliers

If the player has a variable called *bonus_multiplier* with a value other than 1, MPF will add two more events between the entries and the total. First it will post *bonus_subtotal* with an argument *score*, which is the sum of all entry awards. Then it will post *bonus_multiplier* with an argument *multiplier*, which is the value of the player's bonus multiplier. The resulting *bonus_total* event value (and the amount added to the player's score) is the bonus subtotal multiplied by the bonus multiplier.

If the player does not have a *bonus_multiplier* value or if this value is 1, these events will not post and the bonus total will be the subtotal.

Additional Configuration

The bonus mode can be configured with more options, including:

- Reset player variables and/or multipliers after bonuses are awarded
- Show bonus scores for entries that awarded zero points
- "Hurry up" the bonus mode based on a triggering event (e.g. *flipper_cancel*)
- After awarding all bonuses, wait for an event before ending the mode

All these options are detailed in the *bonus (mode_settings:)* documentation.

Related Events

- *bonus_multiplier*
- *bonus_start*
- *bonus_subtotal*
- Plus other events defined in your bonus mode's `bonus_entries` settings

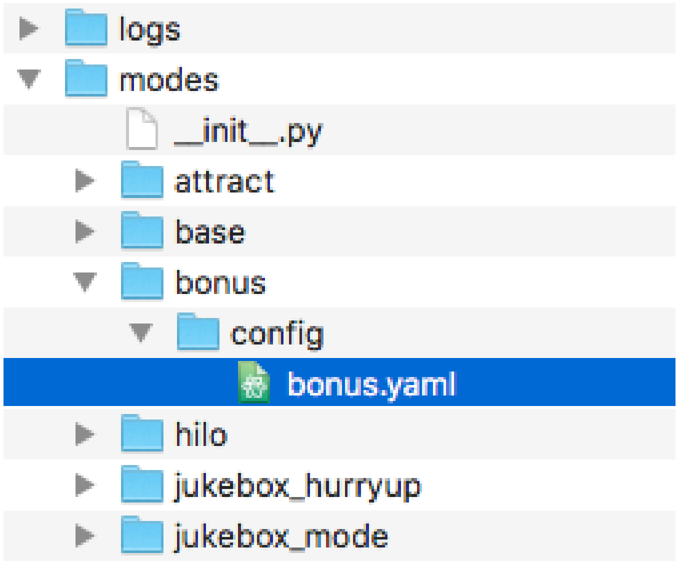
How to configure End of Ball Bonus

This guide walks you through configuring an end-of-ball Bonus mode in MPF.

1. Create your bonus mode folders

Even though the bonus mode is built-in, you'll still need to add a `bonus` folder to your machine's *modes* folder. Then in there, add a `config` folder, and finally, create a file in the config folder called `bonus.yaml`. (So this is just like any other mode so far.)

It should look something like this:



2. Add the bonus mode to your machine-wide modes list

Remember that when you create a new mode, you need to add it to the `modes:` section of your machine-wide config. (Why doesn't MPF just automatically detect modes based on what folders it finds? Because you might want to have different sets of configs that use different modes, or you might want to disable a mode you're testing, etc.)

So just add `- bonus` to the list of modes in the `modes:` section of your machine-wide config, like this:

```
# this is your machine-wide config.yaml

modes:
  - base
  - jukebox_mode
  - skill_shot
  - jukebox_hurryup
  - managers_choice_base
  - managers_choice_multiball
  - managers_choice_timed_mode
  - managers_choice_lit
  - mystery_lit
  - wizard_advance_lit
  - mission_rotator
  - light_mission_select
  - play_poker
  - money_bags
  - world_tour
  - music_awards
  - jukebox_two_ball
  - bonus                  # just add bonus to the list of existing modes
```

The bonus mode is automatically configured to start when the ball ends (as long as the machine is not tilted), running at priority 500.

3. Think about what you want to score bonus on

Most modern pinball machines have bonus scores based on multiple things.

Use a *variable_player:* to count some bonuses:

```
##! mode: mode1
variable_player:
  ramp_shot_hit:
    bonus_ramps: 1
  s_target1_active:
    some_variable: 1
```

4. Add some settings to your bonus mode config

Now go back into your bonus mode folder open up `bonus.yaml` config file (which should be empty at this point), and enter a basic config:

```
##! mode: bonus
#config_version=5
mode_settings:
  display_delay_ms: 1s
  hurry_up_delay_ms: 0
  bonus_entries:
    - event: bonus_ramps
      score: 400
    - event: bonus_math
      score: 1200 * (current_player.some_variable + 2)
slide_player:
  mode_bonus_started: bonus_start_slide
  bonus_ramps: bonus_ramp_slide
  bonus_math: bonus_math_slide
  bonus_total: bonus_total_slide
slides:
  bonus_start_slide:
    widgets:
      - type: text
        text: Bonus
  bonus_ramp_slide:
    - type: text
      text: "Ramps (player|level)"
    - type: text
      text: (score)
  bonus_math_slide:
    - type: text
      text: "Some variable (player|some_variable)"
    - type: text
      text: (score)
  bonus_multiplier_slide:
    - type: text
      text: "Multiplier"
    - type: text
      text: "(multiplier)X"
```

(continues on next page)

(continued from previous page)

```
bonus_total_slide:
  - type: text
    text: "Total Bonus"
  - type: text
    text: (score)
```

You can use *placeholder variables* and math in all your score entries.

Coins & Credits

Related Config File Sections
credits:

This How To guide explains how to setup your machine to take money and track credits. The MPF package contains a the code for a mode called `credits`, so all you have to do to use add some configs to your machine's `modes` folder and sit back and get rich! The credits system has several features and options, including:

- Configuration of different coin/price values per coin switch.
- Tracking money and/or tokens.
- Set price tiers (1 credit for 50 cents, 5 credits for 2 dollars, etc.)
- Specify max credits and credit expiration times
- Retain credits even when the machine is powered off
- Get access to a "credits string" machine variable that will show the number of credits (or configurable free play text) for use on your display.
- Flexible events you can use to show display items based on credits being added, insert coin messages, max credits reached, etc.

Video about credits:

https://youtu.be/jAQpZy5xYGc

https://youtu.be/w7LuHi7gLMs

(A) Create your 'credits' mode folder

The credits mode works like any other mode in MPF. You'll create a folder called `credits` in your machine's `modes` folder, and that folder will contain subfolders config files, images, etc. So to begin, create a folder called `<your_machine>/modes/credits`. Then inside there, create another folder called `config`. Then inside there, create a file called `credits.yaml`. (So that file should be at `<your_machine>/modes/credits/config/credits.yaml`.)

(B) Configure options for the credits mode

Open up your machine config (`<your_machine>/config/config.yaml`). Next, add a section called `credits:`, and then under there, indent a few spaces (it doesn't matter how many, 2 or 4 or whatever you prefer) and add a section called `categories:`. Your file should now look like this:

```
# in your machine wide config
switches:
  s_coin_left:
    number:
  s_service_coin:
    number:
credits:
  max_credits: 12
  free_play: false
  service_credits_switch: s_service_coin
  switches:
    - switch: s_coin_left
      type: money
      value: .25
  pricing_tiers:
    - price: .50
      credits: 1
    - price: 2
      credits: 5
  fractional_credit_expiration_time: 15m
  credit_expiration_time: 2h
  persist_credits_while_off_time: 1h
  free_play_string: FREE PLAY
  credits_string: CREDITS
```

Full details of what each of these settings does is outlined in the *credits:* of the configuration file reference, so check that out for details on anything not covered here. There are a few sections worth pointing out here though:

switches:

The `switches` section is how you map out the monetary values of credit switches in your machine. Notice that the sub-entries under switches are actually a list with the settings for switch, type, and value repeated multiple times. The `switch:` entry is the name of the switch (from your machine-wide `switches:` section) for the credit switch. Pretty simple. The `value:` entry represents the numeric value of how much is added whenever this switch is hit. Notice that there are no currency symbols here or anything. A value of .25 could be 0.25 dollars or 0.25 Euros or 0.25 Francs—it really doesn't matter. The key is that it's 0.25 of whatever monetary system you have. The `type:` entry specifies what type of currency is being deposited when that switch is hit. This doesn't affect the actual behavior of MPF, rather it's just used in as the column name and for totaling the earnings reports (so you can track "money" separate from "tokens"). You can enter whatever you want here: `money`, `dollars`, `dinars`, etc. You can mix & match these in the same machine if you have a machine that accepts tokens and quarters, for example. Note that the sample credits configuration file has three sets of entries for the credit switches. You just need one for each credit switch. It can be one or two or five - it doesn't matter.

pricing_tiers:

The `pricing_tiers:` section is where you actually set your pricing by mapping how many of your monetary units you want to equate to a certain number of credits. The sample config is fairly common, with 0.50 currency resulting in 1 credit, with a price break at 2 that gives the player 5 credits instead of 4. (So basically they get one free credit if they put in enough money for 4 credits.) The most important thing to know here is that MPF always requires that 1 credit is used to start a game, and 1 credit is required to add an additional player to a game. So if you want to change the price of your game, you don't change the number of credits per game, rather, you change the number of credits a certain amount of money is worth. The pricing tier discount processing is reset when Ball 2 starts. So if it costs $0.50 for one credit or $2 for 5 credits, if the player puts $0.50 in the machine and plays a game, if they wait until that game is over and deposit another $1.50, they'll only get 3 more credits. You can have as many `pricing_tiers` as you want. The first one dictates how much a regular game costs and is required. If you don't want any price breaks, then just add the first one.

service_credits_switch:

This is the name of a switch that's used to add so-called "service credits" to the machine. This switch has a 1-to-1 ratio, meaning that one credit is added to the machine each time this switch is pressed. Notice that this line is commented out (with a # sign) by default, so if you want to use it, change the name of the switch to the name of the switch in your actual machine and remove the # character at the beginning of the line. Service credits are tracked separated in your earnings data file. If you don't have a service credits switch, then just don't add that setting.

(C) Add the credits mode to your list of modes

Now that you have some basic credits settings configured, you can add the credits mode to the list of modes that are used in your machine. To do this, add `- credits` to the modes: section in your machine-wide config, like this:

```
modes:
  - base
  - bonus
  - credits
##! mode: base
##! mode: bonus
```

The order doesn't matter here since the priority each mode runs at is configured in its own mode configuration file. All you're doing now is configuring the credits mode as a mode that your machine will use. You might be wondering why your new `credits.yaml` mode configuration file doesn't have a `mode:` section? That's because the `credits` mode is built-in to MPF (in the `mpf/modes/credits`) folder, so when you add a `credits` folder to your own machine's modes folder, MPF merges together the settings from the MPF modes folder and your modes folder. (It loads the MPF mode config first with baseline settings, and then it merges in your machine's mode config which can override them.) If you look at the built-in `credits` mode's config (at `mpf/modes/credits/config/credits.yaml`), you'll see it has the following `mode:` section:

```
##! mode: credits
mode:
  code: mpf.modes.credits.code.credits.Credits
  priority: 1000010
```

(continues on next page)

(continued from previous page)

```
  start_events: reset_complete
  game_mode: false
  stop_on_ball_end: false
```

First is that the priority of this mode is really high, 11000 by default. That's because we want this mode to run "on top" of any other mode so any slides it puts on the display (like the message for new coins being inserts or the `INSERT COINS` message if the start button is pressed without enough credits) are displayed on top of the slides from any other mode that might be running. Also note that the credits mode starts when the `machine_reset_phase_3` event is posted (which is done as part of the MPF startup process), and that there are no stop events. Basically we want the credits mode to start and never stop. Also note that `stop_on_ball_end:` is set to `false`, again because we don't want this mode to ever stop. (Without that setting, MPF would stop the mode when the ball ends.)

(D) Create slides to show the credits when the player deposits money

Open up the credits mode's config file that you just copied into your machine folder. It should be at `<your_machine>/modes/credits/config/credits.yaml`. Since this file is totally blank, add the required `#config_version=5` to the top line. There are several credit-related things you need to show the player on your display. Here are some settings you can use as a starting point:

```
switches:
  s_coin_left:
    number:
  s_service_coin:
    number:
credits:
  max_credits: 12
  free_play: false
  service_credits_switch: s_service_coin
  switches:
    - switch: s_coin_left
      type: money
      value: .25
  pricing_tiers:
    - price: .50
      credits: 1
    - price: 2
      credits: 5
  fractional_credit_expiration_time: 15m
  credit_expiration_time: 2h
  persist_credits_while_off_time: 1h
  free_play_string: FREE PLAY
  credits_string: CREDITS

##! mode: credits
# in modes/credits/config/credits.yaml
# add some credits slides
slide_player:
  credits_added:
    credit_added_slide:
      expire: 2s
  not_enough_credits:
```

(continues on next page)

(continued from previous page)

```
    not_enough_credits_slide:
      expire: 2s
  enabling_free_play:
    enabling_free_play_slide:
      expire: 2s
  enabling_credit_play:
    enabling_credit_play_slide:
      expire: 2s
  max_credits_reached:
    max_credits_reached_slide:
      expire: 2s
  player_added:
    player_added_slide:
      expire: 1s
slides:
  credit_added_slide:
    - type: text
      text: (machine|credits_string)
  not_enough_credits_slide:
    - type: text
      text: (machine|credits_string)
    - type: text
      text: INSERT COINS
  enabling_free_play_slide:
    - type: text
      text: ENABLING FREE PLAY
  enabling_credit_play_slide:
    - type: text
      text: ENABLING CREDIT PLAY
    - type: text
      text: (machine|credits_string)
  max_credits_reached_slide:
    - type: text
      text: MAX CREDITS REACHED
  player_added_slide:
    - type: text
      text: PLAYER ADDED
      font_size: 12
      color: white
sound_player:
  credits_added:
    credit_added_sound:
      action: play
      loops: 0
  not_enough_credits:
    need_more_money:
      action: play
      loops: 0
  player_added:
    player_added_sound:
      action: play
      loops: 0
```

There are several events that the credit module will post which you can use to trigger slides:

- *max_credits_reached* – Posted once when the max number of credits is reached.
- *credits_added* – Posted any time a credit or partial credit is added. Use it with machine variables (below) to show the values.
- *not_enough_credits* – Posted when the player pushes start but there is not at least one credit to add a player. This could happen in attract mode or during the first ball of a game when it's still possible to add players.
- *enabling_free_play* – Posted when the machine is switched to free play mode. (In case you want to have a switch or something which changes it. Details below.)
- *enabling_credit_play* – Posted when the machine is switched to credit (pay) mode.

(E) Adding credits information to game slides

Many of the display slides in a pinball machine display information about the number of credits on the machine. For example, the default score display slide will usually contain a message about how many credits are on the machine. This can be a challenge since the exact text you want to display will change based on whether or not the machine is on free play, and whether there are any fractions of credits on the machine or only whole credits. To handle this, MPF includes a machine variable called `credits_string` that is automatically updated to show the value of credits on the machine. If the machine is set to free play, or if you don't have the credits mode enabled, the `credit_string` value is `FREE PLAY`. Otherwise it's the word CREDIT followed by the number of credits (in fraction, not decimal, as is tradition with pinball machines). Note that you can override the text here with the `free_play_string` and `credits_string` configuration options. Remember that you can include machine variables in a text display element (in either a *slide_player:* or a show YAML file) like this:

```
- type: text
  text: "(machine|credits_string)"
```

And of course you can customize the font, position, and alignment of this display element like any display element. There are several other machine variables created too in case you want to get fancy with how they're displayed in your particular machine. (We'll use an example of `2 1/4` credits here):

- `credits_string` – This is the fully generated string which is ready to use in your slides, including the word `CREDITS` (or `FREE PLAY`) from your settings above, as well as the whole number of credits and any fraction. In the example this would be `CREDITS 2 1/4`.
- `credits_value` – This is just the numeric value of the credits, including the fraction (if there are any partial credits). For example, `2 1/4`.
- `credits_whole_num` – This is just the whole number of credits. Example: *2*.
- `credits_numerator` – This is just the numerator of the fraction of partial credits. Example: *1*.
- `credits_denominator` – This is just the denominator of the fraction of partial credits. Example: *4*.

The denominator of the fraction in the `credit_string` is automatically calculated based on the smallest value coin switch and the price of your game. So 0.25 switches with a game price of 0.50 will use "2" as the denominator (for 1/2 credits). 0.25 switches with 0.75 game will use 3, etc. Remember that text elements with machine variables in slides automatically update themselves when the underlying variable changes. So you can use these in your attract mode DMD show, your score display, etc. See the *slide_player:* from the complete example below for details. You can also change a machine between credit mode and free play mode by posting events. (This is not common, but useful if you want to have a switch or something that changes the mode. The "real" way to set this will come later when we build the service mode.) These control events are:

- `enable_free_play` – Puts the machine into free play mode
- `enable_credit_play` – Puts the machine into credit play mode
- `toggle_credit_play` – Toggles the machine between modes.

(F) Viewing Earnings

A tally of the earnings for your machine is available at `<your_machine_folder>/data/earnings.yaml`. Here's an example:

```
money:
  count: 50
  total_value: 14.0
service_credit:
  count: 4
  total_value: 4
token:
  count: 1
  total_value: 1.0
```

Notice that there are sections in this file for each "type" of switch you configured. The sample configuration from the template file included type values of money and token which is why you see them here. If you changed those to something like dollars then you would see a dollars category here. The `count` is the total number of switch hits that contributed towards that count, and the `total_value` is the total numeric value based on the value of each switch. If you configured a `service_credits_switch` then you'll also see a count of service credits. (The service credits count and `total_value` will always be the same since a service credit switch is always worth one credit.)

(G) Allow operator settings of pricing tiers in service modes

In your final machine you do not want to edit the yaml to change pricing tiers. Luckily, there is the *built-in service mode* which allows you to add *more settings*. Let us add two settings and use them in the credits config:

```
# in your machine wide config
switches:
  s_coin_left:
    number:
  s_service_coin:
    number:
settings:
  credits_price_one_credit:
    label: Price for one credit
    values:
      .25: "25ct"
      .5: "50ct"
      .75: "75ct"
      1: "1 dollar"
      2: "2 dollar"
```

(continues on next page)

(continued from previous page)

```
      3: "3 dollar"
      4: "4 dollar"
      5: "5 dollar"
    default: .5
    key_type: float
    sort: 500
  credits_price_tier2:
    label: Price for price tier 2
    values:
      .25: "25ct"
      .5: "50ct"
      .75: "75ct"
      1: "1 dollar"
      2: "2 dollar"
      3: "3 dollar"
      4: "4 dollar"
      5: "5 dollar"
    default: 2
    key_type: float
    sort: 510
  credits_credits_tier2:
    label: Number of credits for tier 2
    values:
      2: "2"
      3: "3"
      4: "4"
      5: "5"
      6: "6"
      7: "7"
      8: "8"
      9: "9"
      10: "10"
    default: 5
    key_type: int
    sort: 520
credits:
  max_credits: 12
  free_play: false
  service_credits_switch: s_service_coin
  switches:
    - switch: s_coin_left
      type: money
      value: .25
  pricing_tiers:
    - price: settings.credits_price_one_credit
      credits: 1
    - price: settings.credits_price_tier2
      credits: settings.credits_credits_tier2
  fractional_credit_expiration_time: 15m
  credit_expiration_time: 2h
  persist_credits_while_off_time: 1h
  free_play_string: FREE PLAY
  credits_string: CREDITS
```

(H) Check out this complete credits config file

Here's the complete credits config file from the Demo Man sample game. (`demo_man/modes/credits/config/credits.yaml`):

This is an example:

```
# in your machine wide config
switches:
  s_coin_left:
    number:
  s_service_coin:
    number:
settings:
  credits_price_one_credit:
    label: Price for one credit
    values:
      .25: "25ct"
      .5: "50ct"
      .75: "75ct"
      1: "1 dollar"
      2: "2 dollar"
      3: "3 dollar"
      4: "4 dollar"
      5: "5 dollar"
    default: .5
    key_type: float
    sort: 500
  credits_price_tier2:
    label: Price for price tier 2
    values:
      .25: "25ct"
      .5: "50ct"
      .75: "75ct"
      1: "1 dollar"
      2: "2 dollar"
      3: "3 dollar"
      4: "4 dollar"
      5: "5 dollar"
    default: 2
    key_type: float
    sort: 510
  credits_credits_tier2:
    label: Number of credits for tier 2
    values:
      2: "2"
      3: "3"
      4: "4"
      5: "5"
      6: "6"
      7: "7"
      8: "8"
      9: "9"
      10: "10"
    default: 5
```

(continues on next page)

(continued from previous page)

```
    key_type: int
    sort: 520
credits:
  max_credits: 12
  free_play: false
  service_credits_switch: s_service_coin
  switches:
    - switch: s_coin_left
      type: money
      value: .25
  pricing_tiers:
    - price: settings.credits_price_one_credit
      credits: 1
    - price: settings.credits_price_tier2
      credits: settings.credits_credits_tier2
  fractional_credit_expiration_time: 15m
  credit_expiration_time: 2h
  persist_credits_while_off_time: 1h
  free_play_string: FREE PLAY
  credits_string: CREDITS
##! mode: attract
# in modes/attract/config/attract.yaml
# add credits string to your attract show
show_player:
  mode_attract_started: attract_display_loop
shows:
  attract_display_loop:
    - duration: 2s
      slides:
        press_start:
          target: dmd
          widgets:
            - type: Text
              text: PRESS START
          transition:
            type: move_in
            duration: 1s
            direction: top
    - duration: 2s
      slides:
        credits_slide:
          target: dmd
          widgets:
            - type: text
              text: (machine|credits_string)
          transition:
            type: move_in
            duration: 1s
            direction: bottom
##! mode: credits
# in modes/credits/config/credits.yaml
# add some credits slides
slide_player:
  credits_added:
```

(continues on next page)

(continued from previous page)

```
    credit_added_slide:
      expire: 2s
  not_enough_credits:
    not_enough_credits_slide:
      expire: 2s
  enabling_free_play:
    enabling_free_play_slide:
      expire: 2s
  enabling_credit_play:
    enabling_credit_play_slide:
      expire: 2s
  max_credits_reached:
    max_credits_reached_slide:
      expire: 2s
  player_added:
    player_added_slide:
      expire: 1s
slides:
  credit_added_slide:
    - type: text
      text: (machine|credits_string)
  not_enough_credits_slide:
    - type: text
      text: (machine|credits_string)
    - type: text
      text: INSERT COINS
  enabling_free_play_slide:
    - type: text
      text: ENABLING FREE PLAY
  enabling_credit_play_slide:
    - type: text
      text: ENABLING CREDIT PLAY
    - type: text
      text: (machine|credits_string)
  max_credits_reached_slide:
    - type: text
      text: MAX CREDITS REACHED
  player_added_slide:
    - type: text
      text: PLAYER ADDED
      font_size: 12
      color: white
sound_player:
  credits_added:
    credit_added_sound:
      action: play
      loops: 0
  not_enough_credits:
    need_more_money:
      action: play
      loops: 0
  player_added:
    player_added_sound:
      action: play
```

(continues on next page)

(continued from previous page)

```
    loops: 0
```

A game will always cost 1 credit per player. In this example, 50ct will give you 1 credit and $2 will give you 5 credits. When `s_coin_left` is hit 25ct are added (or 1/2 credit).

This mode will also play sounds and show slides when adding credits or players since both can happen before or during a game.

Related How To guides
How to design a game in MPF using Modes

Machine Variables
credit_units
credits_numerator
credits_string
credits_value
credits_whole_num

Related Events
credits_added
enabling_credit_play
enabling_free_play
max_credits_reached
not_enough_credits
player_added

Combo Switches ("flipper cancel", etc.)

Related Config File Sections
combo_switches:

MPF contains support for "combo switches" which are special combinations of switches that post events when they're hit together.

The most basic example of this is the "flipper cancel" combination, where a player can cancel a show or bonus by hitting both flippers at the same time. In fact MPF contains built-in support for the flipper cancel combo. If you add the tag `left_flipper` to your left flipper switch, and `right_flipper` to your right flipper switch, then whenever the player hits both flippers at the same time, an MPF event called *flipper_cancel* will be posted.

Combo switches are also used for things like different kinds of skill shots. For example, in *Attack From Mars*, if the player hits the launch button, the ball is launched into the pop bumper area, but if the player holds down the left flipper button while pressing the launch button, the ball gate (Bally part A-17796) in the upper playfield is raised and the ball is allowed to pass through and is delivered to the flippers for an attempt at a super skill shot. The left flipper + launch button combination is something you can enable with MPF's combo switches.

MPF's combo switches also generate events once both switches are hit together, then one switch is tapped while the other is held in. This can be used to scroll through certain information screens with one button while the combo is active.

You can set various timing options for combo switches, including how close together the two switches have to be hit to count as a combo, how long they have to be held, and how long they have to be released.

Built-in flipper cancel combo

MPF's `mpfconfig.yaml` (the built-in machine config that's merged in with all machine configs) includes the following section:

```
combo_switches:
  both_flippers:
    tag_1: left_flipper
    tag_2: right_flipper
    events_when_both: flipper_cancel
```

This means if you tag add `tags: left_flipper` to your left flipper button and `tags: right_flipper` to your right flipper button, you'll get an event *flipper_cancel* posted anytime the player has both flipper buttons pushed in which you can use to cancel shows or whatever else you want to do. If you want to change or override this (perhaps you want to set a `max_offset_time:` to make sure this event is only posted if the player hits the flipper buttons within 500ms, then you can copy and add this section to your own machine config file and it will overwrite this default config.

Here is an example of using flipper_cancel to cancel a show:

```
switches:
  s_flipper_left:
    tags: left_flipper
    number:
  s_flipper_right:
    tags: right_flipper
    number:

shows:
  mode_intro:
    - duration: 5
      slides:
        mode_intro_slide:
          widgets:
            - type: text
              text: Hit 50 switches to light jackpot
              color: white
              font_size: 100
show_player:
```

(continues on next page)

(continued from previous page)

```
start_mode_intro_show:
  mode_intro:
    loops: 0
    events_when_stopped: mode_intro_show_ended
flipper_cancel:
  mode_intro:
    action: stop
```

The start_mode_intro_show will play for 5 seconds unless both flipper buttons are pressed which will cancel the show.

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for combo switches is `device.combo_switches.<name>`.

state String which reflects what state this combo switch is in. Options wil be one of the following: *inactive*, *both* or *one*.

Related How To guides

- *Canceling ball end shows using flipper_cancel*

Related Events

- *(name)_one*
- *(name)_both*
- *(name)_inactive*
- *(name)_switches_1*
- *(name)_switches_2*

Extra Balls

Related Config File Sections
extra_balls:
extra_ball_groups:

MPF has support for extra balls. Extra balls in MPF are "named", and they're tracked so that (by default) each extra ball can only be awarded once. You can configure as many different extra balls as you want, each with different settings that tie into the events that award them. Every extra ball device can award up to x extra balls (defaults to 1). Additionally, you can define extra ball groups which can further limit the maximum number of extra balls.

Score an Extra Ball Based on Score

Some games (especially EMs) award extra balls based on the score. This is an example:

```
##! mode: base
# in your base mode
extra_balls:
  score_one:
    enabled: true
    award_events: player_score{value>=140000}
  score_two:
    enabled: true
    award_events: player_score{value>=210000}
  score_three:
    enabled: true
    award_events: player_score{value>=300000}
```

Related How To guides

Todo: *Help us to write it*

Related Events

- *extra_ball_award_disabled*
- *extra_ball_(name)_award_disabled*
- *extra_ball_(name)_lit*
- *extra_ball_(name)_awarded*
- *extra_ball_awarded*
- *extra_ball_group_(name)_awarded*
- *extra_ball_group_(name)_lit*
- *extra_ball_group_(name)_unlit*
- *extra_ball_group_(name)_award_disabled*
- *extra_ball_group_(name)_lit_awarded*

High Scores

Related Config File Sections
high_score:

MPF includes support for high scores which is where players can enter their names (or initials) when they've achieved a high score. Features include:

- Set any player variable as a high score option. So in addition to score you could set high score entries for loops, ramps, aliens destroyed, etc.
- Set how many of each high score type are tracked (Top 5 for high scores, Top 3 for loops, Top 1 for aliens, etc.)
- Set what each "award name" is called. (The highest score is "GRAND CHAMPION," the second highest score is "HIGH SCORE 1", the highest loop score is "MAJOR LOOPER", etc.)
- How many characters a player can enter for their name.
- A list of valid characters the player can choose from
- The layout of the display for entering their names and show their rewards.
- Events for high score awards and entry, so you can configure high score entry screens.

Don't have a display to enter initials? See *High Scores in EM Machines* for how to use the high score mode without entering initials.

This is an example (for machines with display):

```
##! mode: my_mode
##! mode: high_score
# modes/high_score/config/high_score.yaml
mode:
  priority: 500
  start_events: game_ending, start_high_score
  use_wait_queue: true
high_score:
  _overwrite: true
  enter_initials_timeout: 60
  award_slide_display_time: 4s
  # define your high score categories and the awards
  categories: !!omap
    - score:
        - GRAND CHAMPION
        - HIGH SCORE 1
        - HIGH SCORE 2
        - HIGH SCORE 3
        - HIGH SCORE 4
        - HIGH SCORE 5
        - HIGH SCORE 6
        - HIGH SCORE 7
        - HIGH SCORE 8
    - loops:
        - LOOP CHAMP
  # set the defaults
  defaults:
    score:
      - MPF: 1000000
      - BRI: 900000
      - JAN: 800000
      - QUI: 700000
      - MAR: 600000
      - JOH: 500000
      - ELI: 400000
      - MIK: 300000
```

(continues on next page)

(continued from previous page)

```
      - ANT: 200000
    loops:
      - JAN: 42
# optional: change the slides (you can omit all the following)
slide_player:
  _overwrite: true
  high_score_enter_initials: high_score_enter_initials
  high_score_award_display: high_score_award_display
slides:
  _overwrite: true
  high_score_enter_initials:
    - type: text
      style: big
      font_size: 18
      text: PLAYER (player_num)
      color: ffff00
      x: 105
      y: 90
    - type: text
      style: big
      font_size: 18
      text: (award)
      color: f0f0f0
      x: 105
      y: 70
    - type: text_input
      initial_char: A
      dynamic_x: false
      key: high_score
      style: big
      font_size: 18
      max_chars: 3
      x: 105
      y: 20
      shift_left_event: sw_lower_left_flipper
      shift_right_event: sw_lower_right_flipper
      select_event: sw_start
      color: ff0000
    - type: text
      style: big
      text: '<       >'
      font_size: 18
      x: 105
      y: 20
      color: ff0000
    - type: text
      text: ''
      key: high_score
      font_size: 18
      style: big
      x: 105
      y: 50
      color: ff00ff
      animations:
```

(continues on next page)

(continued from previous page)

```
      show_slide:
        - property: opacity
          value: 1
          duration: 0.3s
          easing: in_out_quint
        - property: opacity
          value: 0
          duration: 0.3s
          repeat: true
          easing: in_out_quint
high_score_award_display:
  - type: text
    text: (player_name)
    font_size: 18
    style: big
    anchor_y: middle
    anchor_x: middle
    x: middle
    y: middle
    color: 00ff00
    animations:
      show_slide:
        - property: opacity
          value: 1
          duration: 0.05s
        - property: opacity
          value: 0
          duration: 0.05s
          repeat: true
  - type: text
    text: (award)
    font_size: 18
    style: big
    x: 105
    y: 110
    color: 0000ff
  - type: text
    text: (value)
    style: big
    x: 105
    y: 30
    color: 4040FF
    font_size: 20
    number_grouping: true
    min_digits: 2
```

High score mode will also create a few machine variables for you:

- *(high_score_category)(position)_label* - `score1_label` = GRAND CHAMPION
- *(high_score_category)(position)_name* - `score1_name` = MPF
- *(high_score_category)(position)_value* - `score1_value` = 1000000

In this case this will be `score1_value`, `score1_name` and `score1_label` (till `score9_value`, `score9_name` and `score9_label`). Additionally, there will be `loop1_label`, `loop1_value` and `loop1_name`. You can use those in your attract slides to show previous high scores. This is an example of an attract mode which shows high scores:

```
# in your machine wide config file
widget_styles:
  attract_mode_high_score_display_label:
    font_size: 30
    anchor_x: right
    anchor_y: top
    x: center-10
    bold: true
  attract_mode_high_score_display_name:
    font_size: 30
    anchor_x: right
    anchor_y: top
    x: center+70
  attract_mode_high_score_display_score:
    font_size: 30
    anchor_x: left
    anchor_y: top
    x: center+90
    number_grouping: true
    min_digits: 1
##! show: attract
# in your attract mode show file
- duration: 20s
  slides:
    show_high_scores:
      widgets:
      - type: Text
        text: HIGH SCORES
        font_size: 60
        bold: true
        anchor_x: center
        anchor_y: center
        x: center
        y: top-100
      - type: Text
        text: (machine|score1_label)
        style: attract_mode_high_score_display_label
        y: top-200
      - type: Text
        text: (machine|score1_name)
        style: attract_mode_high_score_display_name
        y: top-200
      - type: Text
        text: (machine|score1_value)
        style: attract_mode_high_score_display_score
        y: top-200
      - type: Text
        text: (machine|score2_label)
        style: attract_mode_high_score_display_label
        y: top-240
      - type: Text
        text: (machine|score2_name)
```

(continues on next page)

(continued from previous page)

```
        style: attract_mode_high_score_display_name
        y: top-240
      - type: Text
        text: (machine|score2_value)
        style: attract_mode_high_score_display_score
        y: top-240
      - type: Text
        text: (machine|score3_label)
        style: attract_mode_high_score_display_label
        y: top-280
      - type: Text
        text: (machine|score3_name)
        style: attract_mode_high_score_display_name
        y: top-280
      - type: Text
        text: (machine|score3_value)
        style: attract_mode_high_score_display_score
        y: top-280
      - type: Text
        text: LOOP CHAMPION
        font_size: 60
        bold: true
        anchor_x: center
        anchor_y: center
        x: center
        y: top-500
      - type: Text
        text: (machine|loops1_label)
        style: attract_mode_high_score_display_label
        y: top-600
      - type: Text
        text: (machine|loops1_name)
        style: attract_mode_high_score_display_name
        y: top-600
      - type: Text
        text: (machine|loops1_value)
        style: attract_mode_high_score_display_score
        y: top-600
```

Related How To Guides
Scoring
How to design a game in MPF using Modes
High Scores in EM Machines

Related Events
mode_high_score_started
mode_high_score_stopped

High Scores in EM Machines

Related Config File Sections
high_score:
player_vars:

Electro Mechanical (EM) pinball machines usually do not have a display which allows a player to enter initials. To use the existing *high score mode* we can preset player initials using *player_vars:*.

```
player_vars:
  initials:
    value_type: str
    initial_value: AAA
```

After setting this in your machine config the high score mode will no longer ask for initials. The exact string (here `AAA`) does not matter since you usually will not show it anyway. If you have another way to enter initials you can also use that and set the initials to the `initials` player variable.

Logic Blocks

MPF config files include the concept of "logic blocks" which let you perform logic when certain events occur. Logic blocks can be thought of as the "glue" that ties together all the different shows, shots, achievements, and other parts of your game logic.

There are four types of logic blocks in MPF:

counters Count the number of times an event happens, and when a certain number is hit, a "complete" event is posted.

accruals Watch for several different events to occur, and once they all do (no matter what order they happen in), a "complete" event is posted.

sequences Watch for several different events that need to occur *in a specific order*, and once they do, a "complete" event is posted.

state_machines A generic state machine with arbitrary state transitions and state.

Logic blocks can be configured to store their state in player variables, meaning that each logic block will remember where it was from ball-to-ball.

Logic blocks can be added to modes, and they can have events to enable, disable, and reset them.

To help you understand how logic blocks might be used, here are some real world examples from *Attack from Mars* (if we were building that game in MPF):

- A counter logic block can count the number of times a pop bumper is hit, and then when it hits a certain number, it posts an event to start a "Super Jets" mode.
- A counter can be used to track the three hits to the force field that are needed to lower it.
- A counter can be used (along with a timer) to track combos
- An accrual can be used in the Martian Attack mode to track all 4 of the martians being hit

You should also read about *integration of show and logic blocks*.

Counter Logic Blocks

Related Config File Sections
counters:

Related How To Guides
Integrating Logic_Blocks and Shows
Scoring Based on Logic Blocks
Integrating Logic_Blocks and Lights
Integrating Logic_Blocks and Slides
Persisting the State of a Logic Block in a Player Variable

"Counters" are *logic blocks* that track the number of times a certain event happens towards the progress of a completion goal.

Examples include:

- Hit a target (or shot) X number of times to advance.
- Hit pop bumpers 75 times to start a Super Jets mode.
- Counting the number of combos made
- Keeping track of a bonus multiplier (maybe you use the shot group lane completion event to count progress towards the bonus multiplier, but you configure the max count to be 6, and then if it's hit again, you award an extra ball).

You can use optional parameters to specify whether multiple occurrences in a very short time window should be grouped together and counted as one hit, the counting interval, and whether this counter counts up or down.

Here's an example of a counter you could use to track progress towards super jets:

```
##! mode: my_mode
counters:
  super_jets:
    count_events: sw_pop
    events_when_hit: pop_hit
    starting_count: 75
    count_complete_value: 0
    direction: down
    events_when_complete: super_jets_start
```

And here's the logic block we use for the *Addams Family mansion awards* to make sure the mansions is initialized only once per game:

```
##! mode: my_mode
counters:
  initialize_mansion:
    count_events: mode_chair_lit_started
    events_when_complete: initialize_mansion
    count_complete_value: 1
    persist_state: true
```

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for ball holds is `device.counters.<name>`.

value The count of this counter.

enabled Boolean (true/false) which shows whether this counter is enabled.

completed True if the block is completed. Otherwise False.

This is an example:

```
##! mode: my_mode
counters:
  test_counter:
    count_events: count_up
    reset_on_complete: false
    count_complete_value: 3
event_player:
  test_event{device.counters.test_counter.value > 1}: count_above_one
  test_event{device.counters.test_counter.completed}: count_completed
```

Related Events

- *logicblock_(name)_complete*
- *logicblock_(name)_hit*
- *logicblock_(name)_updated*

Common Issues

We try to answer some common questions regarding logic blocks here. If you question is not answered please ask in the forum.

My block only works once. Why?

This is the default configuration of all logic blocks. To change it you first need to set `reset_on_complete` to `True`. As a result you blocks will reset when they reach the final step. However, that will not be enough in most cases because `disable_on_complete` is `True` by default. Unless you have some enable logic to re-enable the block later, you probably want to set `disable_on_complete` to `False`.

When should I used logic blocks and when should I use shots/show_groups?

There is no definitive answer to this question. Generally, it depends on your usecase. Shots and shot_groups serve a very specific usecase. Basically, they implement a sequences of switch hits which trigger lights along the way. If you want to stay within that specific usecase then go with shots because it will be more convenient. If you plan to extend your mode to use more advanced features then go with logic blocks. For instance if you got conditions in your logic (i.e. on how many balls are locked). Another clear indicator for logic blocks would be if your logic is triggered by other elements such as locks (and not just switches).

Accrual Logic Blocks

Related Config File Sections
accruals:

Related How To Guides
Integrating Logic_Blocks and Shows
Scoring Based on Logic Blocks
Integrating Logic_Blocks and Lights
Integrating Logic_Blocks and Slides
Persisting the State of a Logic Block in a Player Variable

"Accruals" are a type of *Logic Block* where you can trigger a new event based on a series of one or more other events.

Accruals are almost identical to *Sequence Logic Blocks*, the only difference being that the steps in an Accrual Logic Block can be completed in any order, and the steps in a Sequence Logic Block must be completed in the specific order they're listed.

An example might be if you have 3 different things which need to happen in your machine, and when they're all complete, some other event is posted which kicks off some kind of award mode.

You would use an accrual if these 3 events can happen in any order. If they need to happen in a specific 1-2-3 sequence, then you would use a *sequence* logic block instead. (And if you just need the same event to happen three times, then you would use a *counter* logic block instead.

For example, let's say you had a mode where you wanted three shots to be hit, in any order, and when they were all hit, you lit another shot. You'd use an accrual logic block like this:

```
##! mode: my_mode
accruals:
  name_of_my_logic_block:
    events:
      - shot1_hit
      - shot2_hit
      - shot3_hit
    events_when_complete: enable_winning_shot
```

There are much more settings (as you'll see below), but the basic logic block above (which is called "name_of_my_logic_block") will watch for the events *shot1_hit*, *shot2_hit*, and *shot3_hit* to be posted. Once all three of them have been posted once, this logic block will post an event called *enable_winning_shot* which you can use to play a show, light some other shot, play a sound, award points, etc.

Again, since this is an accrual logic block, those three events can be happen in any order. If one of them is posted twice, that's fine. It doesn't count as one of the other events nor does it "undo" the fact that it was hit.

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for accruals is `device.accruals.<name>`.

value The state of this accrual as list. There will be one entry for every element in the accrual. For instance, if your accrual has three elements if will be a list of len three with index 0 for the status of your first element, 1 for the seconds and 2 for the third element. Elements will be 0 at the beginning and turn to 1 when completed.

enabled Boolean (true/false) which shows whether this accrual is enabled.

completed True if the block is completed. Otherwise False.

This is an example:

```
##! mode: my_mode
accruals:
  test_accrual:
    events:
      - shot1_hit
      - shot2_hit
      - shot3_hit
    reset_on_complete: false     # this is needed for the last event player
event_player:
  test_event{device.accruals.test_accrual.value[0]}: shot1_was_hit
  test_event{device.accruals.test_accrual.value[1]}: shot2_was_hit
  test_event{device.accruals.test_accrual.value[2]}: shot3_was_hit
  test_event{device.accruals.test_accrual.completed}: accrual_completed
# Note: For this last conditional logic to be able to evaluate as true, the accrual setting
# reset_on_complete must be set to No/False. Otherwise the accrual will reset instantly and this will␣
↪never be true.
```

Related Events

- *logicblock_(name)_complete*
- *logicblock_(name)_hit*
- *logicblock_(name)_updated*

Common Issues

We try to answer some common questions regarding logic blocks here. If you question is not answered please ask in the forum.

My block only works once. Why?

This is the default configuration of all logic blocks. To change it you first need to set `reset_on_complete` to `True`. As a result you blocks will reset when they reach the final step. However, that will not be enough in most cases because `disable_on_complete` is `True` by default. Unless you have some enable logic to re-enable the block later, you probably want to set `disable_on_complete` to `False`.

When should I used logic blocks and when should I use shots/show_groups?

There is no definitive answer to this question. Generally, it depends on your usecase. Shots and shot_groups serve a very specific usecase. Basically, they implement a sequences of switch hits which trigger lights along the way. If you want to stay within that specific usecase then go with shots because it will be more convenient. If you plan to extend your mode to use more advanced features then go with logic blocks. For instance if you got conditions in your logic (i.e. on how many balls are locked). Another clear indicator for logic blocks would be if your logic is triggered by other elements such as locks (and not just switches).

Sequence Logic Blocks

Related Config File Sections
sequences:

Related How To Guides
Integrating Logic_Blocks and Shows
Scoring Based on Logic Blocks
Integrating Logic_Blocks and Lights
Integrating Logic_Blocks and Slides
Persisting the State of a Logic Block in a Player Variable

"Sequences" are a type of *Logic Block* where you can trigger a new event based on a series of one or more other events that are first posted in a specific order.

Sequences are almost identical to *Accrual Logic Blocks*, the only difference being that the steps in an Accrual Logic Block can be completed in any order, and the steps in a Sequence Logic Block must be completed in the specific order they're listed.

An example might be if you have to hit four shots in a specific order to complete a mode, like this example from the World Tour mode of *Brooks 'n Dunn*:

```
##! mode: my_mode
sequences:
  finish_world_tour:
    events:
      - shot_north_america_hit
      - shot_south_america_hit
      - shot_europe_hit
      - shot_australia_hit
    events_when_complete: wt_done
```

The example above has a single sequence logic block called "finish_world_tour". When it's enabled, it starts watching for the event *shot_north_america_hit* to be posted. Once it's posted, then it starts watching for the event *shot_south_america_hit* to be posted. At this point, if the europe or australia event is posted, it doesn't matter because this is a "sequence" logic block and the events have to happen in order. So this logic block will just sit there waiting for the current event only to be posted, and then once it is, it moves on, and any posted before or after are just ignored.

Once all four events have been posted in order, the event *wt_done* is posted which you can use to stop the mode or add a score or play a show or whatever you want.

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for sequences is `device.sequences.<name>`.

value The state of this sequence as list. There will be one entry for every element in the sequence. For instance, if your sequence has three elements if will be a list of len three with index 0 for the status of your first element, 1 for the seconds and 2 for the third element. Elements will be 0 at the beginning and turn to 1 when completed.

enabled Boolean (true/false) which shows whether this sequence is enabled.

completed True if the block is completed. Otherwise False.

This is an example:

```
##! mode: my_mode
sequences:
  test_sequence:
    events:
      - shot1_hit
      - shot2_hit
      - shot3_hit
    reset_on_complete: false
event_player:
  test_event{device.sequences.test_sequence.value == 1}: shot1_was_hit
  test_event{device.sequences.test_sequence.value == 2}: shot2_was_hit
  test_event{device.sequences.test_sequence.value == 3}: shot3_was_hit
  test_event{device.sequences.test_sequence.completed}: sequence_completed
```

Related Events

- *logicblock_(name)_complete*
- *logicblock_(name)_hit*
- *logicblock_(name)_updated*

Common Issues

We try to answer some common questions regarding logic blocks here. If you question is not answered please ask in the forum.

My block only works once. Why?

This is the default configuration of all logic blocks. To change it you first need to set `reset_on_complete` to `True`. As a result you blocks will reset when they reach the final step. However, that will not be enough in most cases because `disable_on_complete` is `True` by default. Unless you have some enable logic to re-enable the block later, you probably want to set `disable_on_complete` to `False`.

When should I used logic blocks and when should I use shots/show_groups?

There is no definitive answer to this question. Generally, it depends on your usecase. Shots and shot_groups serve a very specific usecase. Basically, they implement a sequences of switch hits which trigger lights along the way. If you want to stay within that specific usecase then go with shots because it will be more convenient. If you plan to extend your mode to use more advanced features then go with logic blocks. For instance if you got conditions in your logic (i.e. on how many balls are locked). Another clear indicator for logic blocks would be if your logic is triggered by other elements such as locks (and not just switches).

State Machine Logic Block

Related Config File Sections
state_machines:
state_machine_states:
state_machine_transitions:

Related How To Guides
Integrating Logic_Blocks and Shows

"State machines" are a type of *Logic Block* where you can trigger state transitions based on the current state and an event.

Technically, this is a finite state machine as known from CS class.

Video about state machines:

https://youtu.be/qakxTF1H57E

This is an example:

```
##! mode: my_mode
state_machines:
  my_state:
    states:
      start:
        label: Start state
      step1:
        label:
        show_when_active:
          show: on
          show_tokens: None
        events_when_started: step1_start
        events_when_stopped: step1_stop
      step2:
        label:
    transitions:
      - source: start
        target: step1
        events: state_machine_proceed
      - source: step1
        target: step2
        events: state_machine_proceed2
        events_when_transitioning: going_to_step2
      - source: step2
        target: start
        events: state_machine_proceed3
      - source: step1, step2
        target: start
        events: state_machine_reset
```

Storing the State in a Player Variable

If you want to store the state of your state machine in a player variable your can use a *variable_player*. You can then use it on slides or in places where conditions do not work (yet).

```
##! mode: my_mode
state_machines:
  my_state:
    states:
      start:
        label: Start state
      step1:
        label:
        show_when_active:
          show: on
          show_tokens: None
        events_when_started: step1_start
        events_when_stopped: step1_stop
      step2:
        label:
    transitions:
      - source: start
        target: step1
        events: state_machine_proceed
      - source: step1
        target: step2
        events: state_machine_proceed2
        events_when_transitioning: going_to_step2
      - source: step2
        target: start
        events: state_machine_proceed3
      - source: step1, step2
        target: start
        events: state_machine_reset

variable_player:
  "{device.state_machines.my_state.state}":
    my_player_var:
      action: set
      string: "{value}"
```

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for state machines is `device.state_machines.<name>`.

state The state of this state machine as string. This will be one of your entries in your states section.

Integrating Logic_Blocks and Shows

Logic_Block-Triggered Events

Logic_blocks can be flexibly integrated with shows using the *(name)_updated* event. It is posted on every state change (i.e. when a counter is incremented) and when logic_blocks are restored (on mode restart). This means that the event may be posted more than once and all handlers should be idempotent (i.e. that you can execute them more than once without changing state after the first time). This event works well to control shows, lights, slides, and to restore them on the next ball. However it should not be used for scoring (to handle an event when the counter changes, consider the *(name)_hit* event instead).

```
##! mode: my_mode
counters:
  my_counter:
    count_events: my_count_event
    starting_count: 0
    count_complete_value: 3
show_player:
  logicblock_my_counter_updated{value == 0}:
    my_show_initial:
      key: my_counter_show # this is to remove the previous show from the same player
  logicblock_my_counter_updated{value == 1}:
    my_show_first_hit:
      key: my_counter_show # this is to remove the previous show from the same player
  logicblock_my_counter_updated{value >= 2}:
    my_show_final:
      key: my_counter_show # this is to remove the previous show from the same player
```

Every time `my_counter` is updated (or restored) it will post `logicblock_my_counter_updated`. Depending on the value of `my_counter` either `my_show_initial` (value is 0), `my_show_first_hit` (value is 1) or `my_show_final` (value is 2 or 3) are shown. All show_players have the same key so they will stop any other show playing with the same key.

Another way to achieve the same thing is this:

You can even achieve this a bit simpler than in the example. Like this:

```
##! mode: my_mode
counters:
  my_counter:
    count_events: my_count_event
    starting_count: 0
    count_complete_value: 3
show_player:
  logicblock_my_counter_updated{enabled}:
    my_show:
      key: my_counter_show
      start_step: value + 1
      show_tokens:
        led1: l_led1
        led2: l_led2
        led3: l_led3
        color: magenta
  logicblock_my_counter_updated{not enabled}:
    my_counter_show: stop
```

This will start the show `my_show` at the value of the counter `my_counter`. For instance when the counter is 0 it will start step 1, counter 1 will run step 2 and so on. Once the counter is disabled the show it stopped (but other behaviours are possible).

my_show could look like this:

```
##! show: my_show
#show_version=5
- duration: -1
  lights:
    (led1): off
    (led2): off
    (led3): off
- duration: -1
  lights:
    (led1): (color)
    (led2): off
    (led3): off
- duration: -1
  lights:
    (led1): (color)
    (led2): (color)
    (led3): off
- duration: -1
  lights:
    (led1): (color)
    (led2): (color)
    (led3): (color)
```

Actions which should only happen once

If you want something to happen only once when the logic_block advances (and not on mode restart) you should use the _hit event. E.g. for a callout use this:

```
##! mode: my_mode
counters:
  my_counter:
    count_events: my_count_event
    starting_count: 0
    count_complete_value: 10
sound_player:
  logicblock_my_counter_hit{remaining == 5}:
    sound_just_5_remaining:
      action: play
  logicblock_my_counter_hit{remaining == 2}:
    sound_just_2_remaining:
      action: play
  logicblock_my_counter_hit{remaining == 1}:
    sound_just_1_remaining:
      action: play
```

Other Triggered Events

You can also have a show depend on the state of a logic block while being triggered by another event, using *Conditional Events*.

You can access the value directly from the device variable using `devices.counters.my_counter.value`:

```
##! mode: my_mode
show_player:
  some_other_event{devices.counters.my_counter.value==0}: my_show_initial
  some_other_event{devices.counters.my_counter.value==1}: my_show_once_hit
  some_other_event{devices.counters.my_counter.value==2}: my_show_twice_hit
```

Related Events
logicblock_(name)_updated
logicblock_(name)_hit

Scoring Based on Logic Blocks

Sometimes you want to score points based on the state of a logic block.

Accruals

This is a simple example with an accrual. Every event can increase the multiplier exactly once. Multiplier starts at 1 and goes up to 4.

```
##! mode: test
mode:
  start_events: ball_started
accruals:
  my_accrual:
    events:
      - event1_to_increase_multiplier
      - event2_to_increase_multiplier
      - event3_to_increase_multiplier
    events_when_complete: go_bumper
    reset_on_complete: false
variable_player:
  some_score_event:
    score: 10000 * (device.accruals.my_accrual.value[0] + device.accruals.my_accrual.value[1] + device.
↪accruals.my_accrual.value[2] + 1)
```

Counters

Similarly, you can use a counter to increase a multiplier. Every event listed can increase the multiplier multiple times.

```
##! mode: test
mode:
  start_events: ball_started
counters:
  my_counter:
    count_events:
      - event1_to_increase_multiplier
      - event2_to_increase_multiplier
```

(continues on next page)

(continued from previous page)

```
      - event3_to_increase_multiplier
    events_when_complete: go_bumper
    reset_on_complete: false
variable_player:
  some_score_event:
    score: 10000 * (device.counters.my_counter.value + 1)
```

Sequences

This also works with sequences.

```
##! mode: test
mode:
  start_events: ball_started
sequences:
  my_sequence:
    events:
      - event1_to_increase_multiplier
      - event2_to_increase_multiplier
      - event3_to_increase_multiplier
    events_when_complete: go_bumper
    reset_on_complete: false
variable_player:
  some_score_event:
    score: 10000 * (device.sequences.my_sequence.value + 1)
```

Related How To Guides
Scoring

Integrating Logic_Blocks and Lights

Related Config File Sections
counters:
lights:
light_player:

You might want to enable lights based on the state of a counter. This is an example for integrating lights via light_player using subscriptions on the value of the counter:

```
lights:
  l_chest_matrix_green_2:
    number:
  l_chest_matrix_green_3:
    number:
  l_chest_matrix_green_4:
    number:
  l_chest_matrix_green_5:
    number:
```

(continues on next page)

(continued from previous page)

```
counters:
  my_counter:
    starting_count: 0
    count_complete_value: 5
    count_events: count_up

light_player:
  "{device.counters.my_counter.value > 0}":
    l_chest_matrix_green_5: green
  "{device.counters.my_counter.value > 1}":
    l_chest_matrix_green_4: green
  "{device.counters.my_counter.value > 2}":
    l_chest_matrix_green_3: green
  "{device.counters.my_counter.value > 3}":
    l_chest_matrix_green_2: green
```

Integrating Logic_Blocks and Slides

Related Config File Sections
counters:
slide_player:
variable_player:

You might want to show the count of your counter on a slide. Unfortunately, MC currently cannot subscribe on the value of your counter. However, you can use variable_player to set the value of your counter to a player variable and then use that variable in your slide.

This is an example:

```
#config_version=5

##! mode: my_mode
counters:
  my_counter:
    starting_count: 0
    count_complete_value: 5
    count_events: count_up

variable_player:
  counter_my_counter_hit:
    my_counter:
      action: set
      int: (count)

slide_player:
  show_slide:
    widgets:
      - type: text
```

(continues on next page)

(continued from previous page)

```
        text: "Count (player|my_counter)"
```

Persisting the State of a Logic Block in a Player Variable

Related Config File Sections
counters:
variable_player:

Prior to MPF 0.50 the state of logic blocks has been persisted to player variables. This only longer holds true for player specific blocks (e.g. if you set `persist_state` to `True`). In that case the variable will be called `(logic_block)_state`. For example, a logic block called "logic_block_1" would store its state in a player variable called *logic_block_1_state*. When you do not want to persist the value you can reference it using `device.counters.logic_block_1.value` (also if you set it).

You can easily use this numerical value in a text widget to show the number of combos complete, or the number of pop bumper hits required for super jets, etc. This player variable "state" is different than the state of the logic block itself, which is an object with *enabled*, *completed*, and *value* attributes. Note the difference in accessing the logic block state as a dynamic value vs. placeholder text:

```
##! mode: my_mode
counters:
  logic_block_1:
    count_events: count_up_event

variable_player:
  counter_logic_block_1_hit:  # this is triggered when the counter changes
    my_widget_placeholder: 100 * device.counters.logic_block_1.value
   # The logic block stores the count as the 'value' attribute
widgets:
  counter_widget:
    - type: text
      text: (my_widget_placeholder) Hits!
  # This placeholder is set by variable_player when the counter changes
```

In this example we persist the value of the counter in the player variable `counter_hit` to use it in a slide.

Note: The player variable is only saved if the logic block is configured with `persist_state: True`. If `persist_state` is `False`, the logic block value will _not_ be saved under any variable name (not even the default).

Match Mode

To use the built-in MPF match mode add this config:

```yaml
##! mode: match
# in modes/match/config/match.yaml
queue_relay_player:
  match_no_match:
    post: no_match
    wait_for: slide_no_match_slide_removed
    pass_args: true
  match_has_match:
    post: has_match
    wait_for: slide_match_slide_removed
    pass_args: true
mode_settings:
  non_match_number_step: 10
slide_player:
  no_match:
    no_match_slide:
      expire: 3s
  has_match:
    match_slide:
      expire: 3s
sound_player:
  match_no_match:
    no_match_sound:
      action: play
  match_has_match:
    match_sound:
      action: play
slides:
  match_slide:
    - type: text
      text: MATCH
    - type: text
      text: "Player 1: (match_number0)"
    - type: text
      text: "Player 2: (match_number1)"
    - type: text
      text: "Player 3: (match_number2)"
    - type: text
      text: "Player 4: (match_number3)"
    - type: text
      text: "Match number: (winner_number)"
  no_match_slide:
    - type: text
      text: NO MATCH
      font_size: 12
      anchor_y: bottom
    - type: text
      text: "Player 1: (match_number0)"
    - type: text
      text: "Player 2: (match_number1)"
    - type: text
      text: "Player 3: (match_number2)"
    - type: text
      text: "Player 4: (match_number3)"
    - type: text
```

(continues on next page)

(continued from previous page)

```
    text: "Match number: (winner_number)"
```

You can extend the slides. See the two events below for available paramters.

Related Events
match_has_match
match_no_match

Related How To guides
How to design a game in MPF using Modes

Modes

Game modes are a big part of pinball programming and a big part of MPF, so it's worth taking an in-depth look at what they are and how they work.

Related Config File Sections
mode:
modes:

As a pinball player, you're probably familiar with the concept of "modes." Most modern machines have lots of different modes, and typically you complete various modes throughout a game on your way to the wizard mode. Many machines have lights on the playfield that show what modes have been completed so far. The player might need to do something to light a "start mode" shot, and then when that shot is made, the mode starts. Then the mode runs for awhile, and while it's running there's typically some kind of sub-goal. (Hit as many standups as you can, shoot both ramps, get as many pop bumper hits as possible, etc.) Some modes run for a predetermined amount of time (e.g. 30 seconds or until the ball drains), some modes are multiball and stop when there's only one ball left, some modes run until the ball ends, some modes run until you complete the mode's objectives, and some modes just sort of run forever.

MPF takes a slightly different approach to modes. In MPF, modes are used for almost everything—a lot more than just "in game" modes. For example, the attract mode is a "mode" in MPF, as is the bonus processing, the high score name entry, and lots of other things that you wouldn't think of as a traditional game mode. In fact even the "game" itself is a mode in MPF! MPF includes many built-in modes (that you can use outright or customize), and you can create your own modes as needed.

We documented the general approach to design a game in the *Game Design* section.

How modes work in MPF

To add a mode to your MPF machine configuration, you create a folder called *modes* in your machine's folder. Then inside there, you create subfolders for each mode in your machine, like this.

In your game, you might have dozens (or even hundreds) of mode folders. Each of your modes folders is almost like a mini-MPF configuration that's only active during that mode. You can have subfolders in each mode folder for game assets, config files, and code that only apply to that mode, like this:

Each of a mode's subfolders follows the same structure as your machine folder in general. The *config* folder holds YAML configuration files, the *shows* folder holds show files, the *sounds* folder contains audio files, the *animations* folder contains animations, etc. (Note that not every type of folder will be in every mode. If a mode doesn't have a specific type of content, then you don't need to include the folder for it.) The idea is that each subfolder holds everything that mode needs, and everything in a mode's folder only applies to that specific mode. For example, in a mode's config file, you can add several types of configuration entries (as detailed in the configuration file reference), that only apply when that mode is active, including:

- shows
- slides
- multiballs
- ball locks
- sounds
- shows
- scoring
- etc.

Again, anything that's specified in a mode's configuration file is only active while that mode is active. So if you have a mode called "multiball" with the following entry in that mode's config file:

```
##! mode: my_mode
variable_player:
  right_ramp_hit:
    score: 50000
```

In that case the *right_ramp_hit* shot event will only award the points when that multiball mode is running. When it stops, that variable_player/scoring configuration is removed. (You can also configure certain events to be "blocked" from propagating down to lower-priority modes. More on that in a bit.)

Machine-wide versus mode-specific folders and configurations

You might have noticed that many of the settings you add to mode- specific configuration files are also valid settings for the machine- wide configuration files which can exist in `your_machine_folder/config/config.yaml` file. So what's the difference between the two? If you configure a setting in a machine- wide configuration file, then that setting will be available at all times in your machine. If you configure a setting in a mode-specific configuration file, then that setting will only apply when that mode is active. The same is true for asset files (in your images, animations, movies, sounds, or shows folder). For example, if you put a sound file in `your_machine_folder/sounds` folder, then that sound will be available to any mode in your machine. If you put it in the *sounds* folder under a specific mode, then that sound file will only be available to that mode. You can even configure assets to automatically load when a mode starts and unload when a mode ends—a feature that is necessary on memory-limited hardware platforms like the BeagleBone Black. The reason MPF's mode system was built this way is so that each mode is self-contained. This is especially useful in situations where more than one person is working on a particular game. You can think of each mode's folder as a mini self-contained MPF environment, as each mode will have its own files and configuration. This also makes it easier to keep track of which modes use which files.

When to use modes

As you read this, it's natural to think of MPF's modes like game modes, and certainly that's a big part of how they're used. But there is no limit to the number of modes that can be active at any one time (and it doesn't negatively affect performance to have dozens of modes running at once), so when you start programming your game you'll probably end up breaking your game logic into lots of little modes.

For example, skill shot should be implemented as a mode. You could create a mode called *skill_shot* that loads when a new player is up, and while it's active it can light certain shots and award points and play light shows and animations associated with the skill shot. You can also setup a timer that automatically starts running when the ball is plunged, and then when the timer ends, you can configure it to unload the skill shot mode. (You would also configure the skill shot mode to stop and unload as soon as the skill shot is made.) You might also have modes which track combos, progress towards ball locks, or really anything else you want.

The key with modes in MPF is to understand that they're more than game modes. You'll create lots and lots of them for all sorts of things. (Basically anything you want which temporarily changes switches, rules, scoring, or any type of device behavior will be a mode in MPF.)

Adding your modes to your machine configuration

If you want to add a mode to your game, you need to add a `modes:` section to your machine configuration file and then create an entry for each mode (by listing the folder), like this: (It's important to have the dash in front of each line.)

```
modes:
  - skillshot
  - base
  - both_ramps_made
  - gun_fight
  - multiball
  - skillshot
  - watch_tower
```

The reason for this is that you might have some modes in your *modes* folder that you're working on that aren't complete yet, or you might want to build different sets of configuration files that use different modes. So you have to list all the modes that you want to use in your machine config file for MPF to read in those modes.

Working with mode-specific config files

We already mentioned that each mode in MPF is really like a full "mini" instance of MPF with settings and assets that only apply to that specific mode. So just like the root MPF config, you create a `config` subfolder in each mode's folder, and then you put a YAML configuration file in that mode's *config* folder that holds all the config settings for that mode. Recall that the default config file name for your machine-wide configuration is a file called `config.yaml`. When you setup a mode's specific config file, you do so by naming the file `<mode_name>.yaml`. (So this file would be `<your_machine_file>/modes/<mode_name>/config/<mode_name>.yaml` file.)

For example, the configuration file for a skill shot mode might be `<your_machine_file>/modes/skillshot/config/skillshot.yaml`. The reason each mode's config file is based on the mode name rather than just being called `config.yaml` is simply for the convenience of the programmer. Our experience is that when we're working on a game, we typically have lots of tabs open in our file editor, and it's really confusing if all the tabs are named *config.yaml*! So we made it so each mode's config file is based on the mode name instead. In each mode's config file, you can add an entry called `mode:` which holds settings for the mode itself. Typically this is just a list of MPF events that will cause the mode to start and stop, as well as the priority the mode runs at, the name of the mode, and whether the mode has any custom Python code that goes with it. (Full details of this are in the `mode:` section of the configuration file reference.)

Organizing modes in subfolders

Modes can also be organized in subfolders. So your modes folder structure could look like:

```
modes
  high_score
  band_gb
     gb_base
     gb_rockfest
  band
     sq
        first_avenue
        release
```

Each mode must include the `config` subfolder with the configuration file. Any folder that includes the `config` subfolder will not be scanned for further modes.

Starting and stopping modes

Modes stop and start based on standard MPF events. For example, if you want a mode to run whenever a ball is in play, you'd add `ball_starting` to the mode's start events list, and you wouldn't specify a stop event. If you want a mode to automatically stop when a timer expires, you'd add the name of the event that's posted when the timer ends to the mode's stop events list.

Mode priorities

When you set up the configuration for a mode (via the `mode:` section of that mode's `config/<mode_name>.yaml` file, you can optionally specify a priority for that mode. Specifying a priority for a mode is useful when you have more than one mode running and you want to control how all the running modes interact with each other.

For example, you can configure scoring events so they "block" lower level modes which have score configured for the same event. So you might have a base game mode which scores 10k points for a ramp shot, but then in one particular mode you might want to make the ramps worth 100k points. To do this you would add the scoring setting for 100k to your special mode, and then you'd run that mode at a higher priority than your base game mode and configure the scoring for that event to block the scoring from the lower mode. (Otherwise you'd get both scoring events and a ramp shot would grant 110k points.) Whether you configure a scoring event to block or not is optional, and you can specify it on an individual basis per scoring event. (And in many case you very well might want to score both events from both modes.)

The mode priorities also affect the priorities of things like all display widgets and slides. For example, your base mode might play an animation and a light show when a ramp shot is made in the base game mode, but when your special higher mode is running you might want to play a different slide and a different light show. So by specifying the special mode to run at a higher priority, it will get priority access to the display and lights. (Again you can configure this on a setting-by-setting basis, because there are plenty of times where you might actually want the lower-priority shows to play even when a higher priority mode is running.)

Note: In MPF prior to v0.20, there was the concept of "machine" modes and "game" modes. Starting with MPF v0.20, those have been combined, and they're just called *modes*. MPF comes with its own built-in modes that will be mixed together with your own machine-specific modes. For example, MPF includes modes for *attract* (priority 10) and *game* (priority 20) which are responsible for the fundamentals of running the attract and game modes.

Using modes as game logic

Using "modes" to implement game logic

One thing I found is that I tend to use modes as a sort of "super" logic block. For example, the Brooks & Dunn rules have a "manager's choice" shot that leads to a ball device. When the shot is lit, one of three things happens depending on what else is going on (one for base game mode, another for when multiball is active, and a third which is a timed mode). The shot may be lit or unlit in any of those three scenarios, and the action I'm talking about should only happen when it's lit, otherwise it just scores some points and kicks out the ball.

I realized pretty quickly that the easiest way to handle this is to create a mode called "managers_choice_lit" which is used to light the shot regardless of what else is happening. When that mode starts, it enables the shot, turns on the light, shows a slide that says the shot is lit, etc. I created a start event "light_managers_choice" which is easy to post from wherever else I need in the game to light the shot.

Then in order to handle the various chains of events that happen when that shot is actually made, I created three more modes:

- managers_choice_base (priority 301)
- managers_choice_timed (priority 302)
- managers_choice_multiball (priority 303)

Each of these modes looks for the "managers_choice_lit" hit (shot) event and then will do their award thing. What's cool is they also each block the shot from the lower down modes. This means that these shots can be stacked and running in any various combination.

So the managers_choice_base mode is running at all times (with a start event of ball_starting). That's safe to run because it doesn't do its award action unless the managers choice lit hit event happens, and that shot is enabled in the managers_choice_hit mode. In other words, managers_choice_base mode can be running at all times, but it will only award the shot if the managers_choice_lit mode is running.

Then if managers_choice_timed or managers_choice_multiball is running, they also do their award thing based on the managers_choice lit hit shot event, so they also can run any time but will not award the shot unless the managers_choice_lit mode is running.

And since those two higher modes block the shot from lower modes, this means that I don't need complicated if/then logic to figure out which of the three award options should be awarded when the shot is lit and hit.

And since the managers_choice_hit mode acts as an on/off switch for whether the shot will be awarded, this means that I can safely start the managers_choice_timed mode any time any other timed mode is running, and I can start the managers_choice_multiball mode anytime multiball play is going on, and they'll each only do their award if the base managers_choice_lit is running and the shot is made.

We documented the general approach to design a game in the *Game Design* section.

Related How To Guides
Creating your first game mode
Game Design

Related Events
mode_(name)_will_start
mode_(name)_starting
mode_(name)_started
mode_(name)_will_stop
mode_(name)_stopping
mode_(name)_stopped
clear

Built-in Modes

MPF includes several "built-in" modes which are ready to use in your game. Some of them are used automatically, and some require that you add some config sections and options to your machine. Click on each for details:

Creating your own modes

Our step-by-step tutorial walks you through creating your own game modes. We just include this page on creating your own modes so you don't read the list of built-in modes and think that's all MPF can do. :)

Also if you haven't read the overview of *how modes work in MPF*, do that now. We documented the general approach to design a game in the *Game Design* section.

Attract (mode)

MPF includes a built-in attract mode which is what runs the machine when a game is not in progress. It starts when either the *game_ended* or *reset_complete* event is posted, and it stops when the *game_start* mode is posted. The attract mode runs at priority 10.

The code and configuration for the built-in attract mode is in the *mpf/modes/attract* folder. It's automatically added to the list of modes in the `modes:` section of your machine-wide config based on settings in the `mpfconfig.yaml` baseline configuration file.

The attract mode is responsible for many things, including:

- Watching for the start button to be pressed & released to kick off the request_to_start_game event
- Recording how long the start button was held in for in order to take different actions based on different times. (For example, maybe pressing the start button normally starts a regular game, and doing a long-press lets the player login with a custom player profile.)
- Recording what other buttons were active when the start button is pressed. (Maybe holding the right flipper button and pushing start enables tournament mode.)

You can completely customize and extend the attract mode. In most cases that's as simple as adding a config file for the attract mode to your game folder and then configuring light and display shows to play. See the tutorial for details on how to do this.

Related How To Guides
high score in attract

Game (mode)

MPF includes a built-in mode called *game* which is responsible for actually running a game in MPF. It starts when a game is started from the attract mode, and it stays running all the way through the entire game, finally stopping again when the game ends and the attract mode starts again.

The code and configuration for the built-in game mode lives in the `mpf/modes/game` folder. It's automatically added to the list of modes in the `modes:` section of your machine-wide config based on settings in the `mpfconfig.yaml` baseline configuration file. The game mode runs at priority 20. It starts when the *game_start* event is posted, and it stops when the *game_ended* event is posted.

The game mode is responsible for many things, including:

- Tracking the number of balls in play. (Remember the number of balls in play is not necessarily the same as the number of live balls on the playfield that the ball controller tracks.)
- Watching for start button pushes to add additional players to the game.
- Restarting the game on a "long press" of the start button.
- Posting the game_started, ball_starting, ball_ending, ball_ended, game_ending, and game_ended events.
- Posting the events relating to multiplayer games.
- Handling ball drains and ending the current player's turn
- Rotating the players and starting the next player's turn
- Processing extra balls and handling shoot again

It's almost never necessary to override or change the behavior of the game mode. Typically anything you want to do to affect the game is done in additional modes you create. (And all the configuration for scoring, game modes, shots, etc. is done in a "base" game mode that runs per player as their turn starts.) See the tutorial for details.

We documented the general approach to design a game in the *Game Design* section.

Credits (mode)

MPF includes a complete credits mode that can be used to enable tracking credits and taking money. See the How To: Add Coins & Credits guide for details of how to set it up.

The credits mode is highly-configurable, including pricing per game, currencies, price tiering ($0.50 for 1 credit, $2.00 for 5, etc.), credit expiration, etc.

See *Coins & Credits* for details.

High score (mode)

MPF includes a built-in high score mode that can be used to track high scores, including letting players enter their names (or initials) and tracking different high score awards. (See the How To: High Scores guide for details).

You can use the config files to completely customize how the high scores work, including the number of scores to track, what you call each award ("GRAND CHAMPION", "HIGH SCORE 1", etc.) and what (and how many) awards you track (score, loops, aliens blasted, etc.).

The high score mode stores its high scores in `<your_machine_folder>/data/high_scores.yaml` file. It automatically reads them in when MPF boots to create machine variables that can be accessed from your game, and it automatically updates the high scores on disk when they change after a game ends.

See *High Scores* for details.

Tilt (mode)

The MPF package includes built-in tilt mode that can be used to track manage tilt warnings, tilts, and slam tilts.

The tilt mode runs at priority 10,000 and automatically starts when MPF boots up. It never stops, even running while the attract mode is running. (This is because you want it to watch for slam tilts that reset the credits even when there's not a game in progress.)

The tilt mode can use traditional mechanical tilts (plumb bobs, weighed switches, and rolling balls), or it can use accelerometers to determine G-forces and angle of the machine which can trigger tilts.

See *Tilt* for details.

Multiballs

Related Config File Sections
multiballs:
multiball_locks:

- *Related Events*

MPF includes a *multiball* feature which can be used to automatically start and stop multiballs.

Each multiball in MPF has a separate name. There are several different types of multiballs (run until a single ball is left, timed multiballs, etc.) Multiballs can also be configured with multiball saves so that (for example) any balls lost in the first 15 seconds of a multiball are automatically re-launched back into play.

MPF also supports stacking of multiple multiballs at the same time.

Balls can be locked for multiball with the related *Multiball Locks* config section.

Video about ball locks and multiballs:

https://youtu.be/2mFkgIlksC4

Common Issues

Why does MPF wait about 10s when adding balls to the playfield from the trough during a multiball?

> When MPF adds a ball to the playfield the launcher waits until the ball is confirmed to be on the playfield. For the first ball this happens when a playfield switch is hit after the eject. However, this will not work with more than one ball on the playfield (e.g. during a multiball). In this case, the launcher will wait until its eject timeout passed (*eject timeouts in ball_devices*) which defaults to 10s. Therefore, you need to tune `eject_timeouts` of your launcher to fix this issue.

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for multiballs is `device.multiballs.<name>`.

balls_added_live Numeric value of how many balls this multiball added into play.

balls_live_target Numeric value of how many balls this multiball is attempting to keep in play.

enabled Boolean (true/false) as to whether this multiball is enabled.

shoot_again Boolean (true/false) as to whether this multiball is in "shoot again" mode which means it's attempting to keep live.

Related How To guides

- *How to create a multiball with a traditional ball lock*
- *How to create a multiball with a virtual ball lock*
- *How to create an "add-a-ball" style multiball*
- *How to create a multiball with a virtual ball lock*
- *How to create a multiball which uses multiple lock devices*

Related Events

- *multiball_(name)_started*
- *ball_save_(name)_timer_start*
- *multiball_(name)_hurry_up*
- *multiball_(name)_grace_period*
- *multiball_(name)_shoot_again*
- *multiball_(name)_lost_ball*
- *multiball_(name)_shoot_again_ended*
- *ball_save_(name)_add_a_ball_timer_start*
- *multiball_(name)_ended*

Multiball Locks

Related Config File Sections
multiballs:
multiball_locks:

Multiball locks work in concert with multiball logic to "lock" balls for multiball. To use a multiball lock, you configure it for the ball device (or devices) that will be used to lock balls, and then when a ball enters one of those devices, the lock count is increased by one.

Video about ball locks and multiballs:

https://youtu.be/2mFkgIlksC4

Multiball locks can be configured in one of four modes of operation:

virtual_only When a new ball is locked, the lock count is increased. Period. It does not matter how many physical balls are locked. Separate counts are maintained per player. This is usually the best option for locks in modern machines.

physical_only As the name implies, the number of balls locked is always the same as the physical number of balls in the lock. A new ball locked will increase the lock count for that player and lock the ball. However if another player "steals" one of the locked balls, then when the previous player starts their turn, the lock count is updated based on the physical balls locked. This is mostly for EM and early solid state machines where balls would be locked in different places on the playfield but the next player could steal them if the player who locked them didn't get multiball started.

min_virtual_physical Similar to physical only except a player locking a ball will always increase the lock count even if that same ball is ejected again.

no_virtual MPF forgets everything when the player changes.

Ball locks are stored on a per-player basic and are NOT based on the number of balls that are physically contained in any ball devices.

When a ball is locked, a new ball will be added into play (from whichever ball device set in `default_source_device` of the playfield) unless the device that just received the locked ball is full, in which case the ball will be released from the device that the ball just entered instead.

Multiball locks can be enabled and disabled with events, so if you want to set up a scenario where a player must "re-light" the lock after each ball is locked, then you can use the event which is posted when a ball is locked as a disable event for this ball lock, and then use the event from some other shot or switch or logic block as an enable event to re-light the lock.

You can configure multiball locks for the total number of balls they should lock which will in turn post a "lock full" event which you can use to start a multiball. That multiball will release all the balls it can from the lock devices this multiball lock uses, and if it still needs more balls (maybe because you're using a virtual lock or because a previous player emptied them out), then it will make up the difference be adding new balls from the ball device set in `default_source_device` of your *playfield*.

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for multiball locks is `device.multiball_locks.<name>`.

enabled Boolean (true/false) as to whether this multiball lock is enabled.

locked_balls The number of balls that are locked. Note that how this number is calculated varies depending on how the ball counting strategy is configured for this multiball lock.

Related How To guides

- *How to create a multiball with a traditional ball lock*
- *How to create a multiball with a virtual ball lock*
- *How to create an "add-a-ball" style multiball*
- *How to create a multiball with a virtual ball lock*
- *How to create a multiball which uses multiple lock devices*

Related Events

- *multiball_lock_(name)_locked_ball*
- *multiball_lock_(name)_full*

How to create a multiball with a traditional ball lock

Related Config File Sections
multiballs:
multiball_locks:

Most pinball machines use a "virtual" ball lock to track multiball progress and MPF is designed to handle these by default. Machines that physically lock multiple balls require a few extra configuration settings to properly count locked balls and release them for a multiball.

Background: How MPF tracks and replaces balls

When a ball enters a ball device that is not the trough, the ball device checks for any locks that want to "claim" the ball. If the ball is claimed by anything, such as an enabled `multiball_lock`, the ball device will hold the ball and request a new ball be added from the trough to the playfield. If nothing claims the ball, the ball device will eject it back onto the playfield.

During this process, the number of "balls in play" never changes. When a ball is claimed by a lock, MPF simply swaps the location of the inactive ball from the trough to the ball device. From the game's perspective the playfield always has one ball in play.

Setting up a simple multiball

An MPF multiball only has one configuration requirement: the number of balls in the multiball (by default the total number of balls, but could also be the number of balls added to those already in play). Consider the following example:

```
##! mode: multiball_mode
multiballs:
  my_multiball:
    ball_count: 3
    ball_count_type: total    # Default
    shoot_again: 10s          # Default
```

With no enable/disable and start/stop events configured, this multiball will begin as soon as its parent mode starts, and it will increase the number of balls on the playfield to a total of 3.

Using a multiball_lock to start a multiball

A typical multiball requires the player to "lock" balls up to the total ball count, which triggers the start of a multiball. You can setup a `multiball_lock` to track progress and use its *multiball_lock_(name)_full* event to start a multiball.

```
##! mode: multiball_mode
multiball_locks:
  madnesslock:
```

(continues on next page)

(continued from previous page)

```
    debug: true
    balls_to_lock: 2
    lock_devices: lockdevice
    reset_count_for_current_player_events: multiball_lock_madnesslock_full
multiballs:
  madnessmball:
    ball_count: 3
    ball_locks: lockdevice
    start_events: multiball_lock_madnesslock_full
```

In the above configuration, the multiball_lock will track the balls entering *lockdevice* and claim up to three. When the third ball is claimed the lock will post its "full" event, which will start the multiball.

Ball-in-play count with physically-locked balls

As noted above, MPF will automatically replace any locked ball with a new ball from the trough, which is necessary for "virtually" locked balls but causes undesirable behavior for physically locked balls. In order to maintain the "balls in play" count, the new ball will be ejected to the playfield immediately—before the multiball can process the *full* event and start itself.

The multiball therefore assumes (correctly) that the last locked ball has already been replaced and thus deducts that "in play" ball from its count of balls to add. In the above example, the multiball would release 2 balls from *lockdevice* which, in addition to the active ball in play, would result in a 3-ball multiball.

Unfortunately, this also leaves one ball locked in *lockdevice* after the multiball starts, which is not the desired outcome.

Overwriting ball replacement for physically-locked balls

You can overwrite the multiball_lock behavior to prevent the automatic replacement of a locked ball with the `balls_to_replace` setting. The default value of -1 instructs the lock to replace every locked ball, but a value of 2 will replace only the first two locked balls.

In tandem, you can overwrite the multiball behavior to not assume that the "in play" ball has been replaced by the lock. The `replace_balls_in_play` setting set to True will instruct the multiball to eject the active ball **and** the additional balls.

```
##! mode: multiball_mode
multiball_locks:
  madnesslock:
    balls_to_lock: 3
    balls_to_replace: 2
    lock_devices: lockdevice
multiballs:
  madnessmball:
    ball_count: 3
    ball_locks: lockdevice
    start_events: multiball_lock_madnesslock_full
    replace_balls_in_play: true
```

With the above configuration, the final locked ball will start the multiball and the multiball will eject three balls from *lockdevice*.

Note: Be careful with with *balls_to_replace* and *replace_balls_in_play*. They will only work in exactly this combination. Used in isolation they will likely lead to incorrect ball counts.

Video about ball locks and multiballs:

https://youtu.be/2mFkgIlksC4

How to create a multiball with a virtual ball lock

If your machine does not have a physical ball lock you can use a counter to count how many times a ball has been "locked". This could be a ball device (as in the example) or any normal shot.

This is an example:

```
switches:
  s_middle_ramp:
    number:
coils:
  c_plunger:
    number:
ball_devices:
  bd_middle_ramp_ball_lock:
    eject_coil: c_plunger
    ball_switches: s_middle_ramp
##! mode: mb_mode
multiballs:
  3balls_multiball:
    ball_count: 3
    ball_count_type: total
    shoot_again: 30s
    start_events: logicblock_mb_counter_complete
counters:
  mb_counter:
    count_events: balldevice_bd_middle_ramp_ball_lock_ball_entered
    count_complete_value: 3
```

Video about ball locks and multiballs:

https://youtu.be/2mFkgIlksC4

How to create an "add-a-ball" style multiball

Help us to write it

How to create a multiball which uses multiple lock devices

Related Config File Sections
multiballs:
ball_devices:
multiball_locks:

Some machines have multiple locks and a multiball may use multiple source devices to eject balls. However, you have to unterstand that *multiball_locks:* and *multiballs:* are independent. A multiball can use one or multiple *ball device* as source devices to eject balls. Locks will keep balls inside ball devices and request new balls to the playfield. To use multiple locks in one multiball you need to combine those: Define locks to lock balls in your lock ball devices. Additionally, define a multiball device with your lock ball devices as source. Then, use the *multiball_lock_(name)_full* events to enable/start your multiball mode. When the multiball starts reset counters of your locks.

In the following example, you have to lock balls sequentially in your three locks. Every lock will enable the next lock using the *multiball_lock_(name)_full* event. The last lock will start the *multiball* mode. If you hit *s_target1* the multiball will start which will reset and disable all locks using the *multiball_(name)_started* event. After all the balls from the multiball drained all lock modes and the multiball mode will stop using the *multiball_(name)_ended* event.

```
coils:
  eject_coil1:
    number:
  eject_coil2:
    number:
  eject_coil3:
    number:
switches:
  s_lock1:
    number:
  s_lock2:
    number:
  s_lock3:
    number:
  s_target1:
    number:
ball_devices:
  bd_lock1:
    eject_coil: eject_coil1
    ball_switches: s_lock1
    eject_timeouts: 2s
  bd_lock2:
    eject_coil: eject_coil2
    ball_switches: s_lock2
    eject_timeouts: 2s
  bd_lock3:
    eject_coil: eject_coil3
    ball_switches: s_lock3
    eject_timeouts: 2s
# mode lock1
##! mode: lock1
mode:
```

(continues on next page)

(continued from previous page)

```
  restart_on_next_ball: true
  stop_events: multiball_my_multiball_started
multiball_locks:
  lock1:
    lock_devices: bd_lock1
    balls_to_lock: 1
    disable_events: mode_multiball_started
    reset_count_for_current_player_events: multiball_my_multiball_started
# mode lock2
##! mode: lock2
mode:
  restart_on_next_ball: true
  start_events: multiball_lock_lock1_full
  stop_events: multiball_my_multiball_started
multiball_locks:
  lock2:
    lock_devices: bd_lock2
    balls_to_lock: 1
    disable_events: mode_multiball_started
    reset_count_for_current_player_events: multiball_my_multiball_started
# mode lock3
##! mode: lock3
mode:
  restart_on_next_ball: true
  start_events: multiball_lock_lock2_full
  stop_events: multiball_my_multiball_started
multiball_locks:
  lock3:
    lock_devices: bd_lock3
    balls_to_lock: 1
    disable_events: mode_multiball_started
    reset_count_for_current_player_events: multiball_my_multiball_started
# mode multiball
##! mode: multiball
mode:
  start_events: multiball_lock_lock3_full
  stop_events: multiball_my_multiball_ended
multiballs:
  my_multiball:
    ball_count: 4
    ball_count_type: total
    shoot_again: 2s
    start_events: s_target1_active
    ball_locks: bd_lock1, bd_lock2, bd_lock3
```

Video about ball locks and multiballs:

https://youtu.be/2mFkgIlksC4

Player Variables

Related Config File Sections
player_vars:

MPF contains lots of features which make working with players easy including variables. If you are not a programmer, variables are just locations inside the computer's memory to store bits of information like numbers and text (aka strings). Programmers create variables to store and retrieve these bit of information for use in their programs. For example, You may want to create a player variable to store the number of times a bumper has been hit to award a bumper bonus.

Each player has "player variables" which are key/value pairs that are stored separately for each player.

Some simple examples of player variables include things like:

- `number`: The player's number (1, 2, etc.)
- `score`: The player's current score

There are two types of player variables that you can use; the default player variables provided by MPF and custom variables that you can create, update and reference.

Default Player Variables

There's a *Player Variables Reference* which lists the default player variables that MPF creates and uses.

MPF also uses player variables to keep track of all the built-in game logic elements that are tracked on a per-player basis, including achievement status, logic block states, extra balls, bonus, etc.

Custom Player Variables

You can also create your own custom player variables which can be called anything you want and can store anything you want. You can use them to track player's progress through the game, how many loops they've made, how many pop bumper hits they have, etc. See the *player_vars:* documentation for details and examples.

Data types

If you are a programmer, you likely know what datatypes are. If you are not a programmer but want to create your own player variables, you'll need to know a little bit about datatypes. To make this really simple, you may want to store the name of the current mode so that you can display the mode name on the display. Since the name of the mode is a piece of text, you'll need to create a player variable of type "str" to denote a string of characters. Here are the data types available in MPF.

Datatype	Description
str	a string of textual characters
int	an integer, a basic number with no decimal point
float	floating point, a more precise number with decimal point

Examples:

```
player_vars:
  current_mode:
    initial_value: Trees Attack
    value_type: str
  bumper_hits:
    initial_value: 0
    value_type: int
  super_bonus_multipler:
    initial_value: 1.25
    value_type: float
```

Player varaibles are essentially global in MPF, meaning that you can define them in config files and they are available to use in any location in your files. This makes them easy to use but also easy to introduce bugs or unintended consequences so be aware of every place that you use them if you are getting unanticipated results. A best practice would be to define all of your player variables in a common location such as the machine configuration file.

Setting Variables

MPF configuration files do not work with variables as easily as "real" programming languages. The primary method of changing a variable is by configuring the change you would like to make. In the current version of MPF, this is primarily done in the `variable_player:` section of your mode.

```
##! mode: my_mode
variable_player:
  # add 1 to bumper_hits
  bumper_1_active:
    bumper_hits: 1
```

The example below shows a player variable of type string being updated. A mode carousel (mode selection by the player) was used by the player to select a mode ladder (a set of modes played in a sequence similar to scenes in GhostBusters). The apostrophes are not required but allowed.

```
##! mode: my_mode
variable_player:
  carousel_left_scoop_scene_selected:
    current_ladder:
      action: set
      string: 'Scene 1'
```

The example below shows a player variable being updated after a conditional event. In this case, the base mode has received an event indicated that a mode has been complete. The conditional event checks to see which mode ladder was in play and increments the custom player variable ladder_scene_1 to indicate the progress towards completing the mode.

```
##! mode: my_mode
variable_player:
  mode_is_complete{current_player.current_ladder=="Scene 1"}:
    ladder_scene_1: 1
```

Displaying Custom Variables

Displaying your custom player variables on a slide can be confusing in the current version of MPF. The example below shows a text widget that is displaying 3 variables on the main scoring screen of the base mode. The first two variables are of type "str" and the last variable is of type "int".

```
player_vars:
  current_ladder:
    initial_value: "Initial Ladder"
    value_type: str
  current_mode:
    initial_value: "No Mode"
    value_type: str
  ladder_scene_1:
    initial_value: 1
    value_type: int

##! mode: base
slide_player:
  mode_base_started:
    widgets:
      - type: text
        text: (current_ladder) > (current_mode) > (ladder_scene_1)
```

Related How To Guides
Help us to write it

Related Events
player_add_request
player_added
player_turn_will_start
player_turn_starting
player_turn_started
ball_save_(name)_saving_ball
player_turn_will_end
player_turn_ending
player_turn_ended
multi_player_ball_started
single_player_ball_started

Replays

Help us to write it

Timed Switches

Related Config File Sections
timed_switches:

MPF includes functionality to manage "timed_switches" which are scenarios when a single switch is continuously active (or inactive, depending on the settings) for a set period of time.

A classic example of this is the flipper "cradling" where a player holds a flipper button in for a few seconds. In almost all modern machines, this is used to trigger a "player info" screen that shows the player's score, how much bonus they have built up, high scores, etc.

Flipper cradling is also used to reset (and pause) the ball search timer, since a player could be holding a ball and drinking a beer, meaning no switch hits will happen, but the ball search should not start.

In fact MPF's default config file (which is automatically used in all games) includes a `timed_switches:` section for flipper cradling and automatically creates *flipper_cradle* and *flipper_cradle_release* events (as long as you tag your flipper switches with *left_flipper* and *right_flipper*).

Note that timed switches are similar to, but not the same as *combo switches*.

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for timed switches is `device.timed_switches.<name>`.

active_switches List of switches that are currently active past the time that this timed_switches: section is set for.

Related How To guides

Todo: *Help us to write it*

Related Events

- *(name)_active*
- *(name)_released*

Timers

Related Config File Sections
timers:

MPF config files include the concept of "timers" which you can use to count towards a specific event based on time. Timers can be configured to count up or down, at whatever interval you want, at any speed you want. You can use events to start, stop, pause, reset, or change their speed.

Timers post events with each "tick" which you can use to update the display, play sounds, etc. They also post events when they complete which you can use to stop a mode, play a show, etc.

Example uses of timers might include:

- Hurry up count down to make a shot (with variable score based on how much time is left).
- Timer to end a timed mode.
- A timer which ticks periodically to rotate a lit shot left or right.
- Etc.

The example config files section of the documentation contains *examples of timers in modes*.

Displaying the value of a timer on a slide

If you want to use your timer in a slide you have to set the value to a player variable first:

```
##! mode: your_mode
# in your mode
timers:
  your_timer:
    start_value: 0
    end_value: 20
    control_events:
      - action: start
        event: mode_your_mode_started
variable_player:
  timer_your_timer_tick:
    your_timer_variable_times_100:
      int: device.timers.your_timer.ticks * 100
      action: set
slides:
  show_timer:
    widgets:
      - type: Text
        text: (player|your_timer_variable_times_100)
slide_player:
  mode_your_mode_started: show_timer
```

In this example we update the player variable `timer_your_timer_tick` every time the timer changes based on the tick event. The value is multiplied by 100 (but you can also omit this or do anything *Variable player* supports). Afterwards, you can use the variable in your slide.

Related Events

- *timer_(name)_started*
- *timer_(name)_stopped*
- *timer_(name)_paused*
- *timer_(name)_complete*
- *timer_(name)_time_added*
- *timer_(name)_time_subtracted*
- *timer_(name)_tick*

Service Mode

Related Config File Sections
settings:

MPF has a build in service mode which can be extended using settings (or in code). Usually you map your *service switches and door switches* to control service mode. Additionally, you might want to add keys of your *keyboard* during development.

This is an example:

```
# include service mode in your modes list
modes:
  - service
# add tags to your switches
switches:
  s_door_open:
    number: 1
    tags: service_door_open, power_off
  s_service_enter:
    number: 17
    tags: service_enter
  s_service_esc:
    number: 18
    tags: service_esc
  s_service_up:
    number: 19
    tags: service_up
  s_service_down:
    number: 20
    tags: service_down
# add a setting (not used here)
settings:
  replay_score:
    label: Replay Score
    values:
      500000: "500000 (default)"
      1000000: "1000000"
```

(continues on next page)

(continued from previous page)

```
      1500000: "1500000"
    default: 500000
    key_type: int
    sort: 100
# add keyboard switches
keyboard:
  right:
    switch: s_service_enter
  left:
    switch: s_service_esc
  up:
    switch: s_service_up
  down:
    switch: s_service_down
# you need to define a "sfx" sound track because the service mode brings some sounds (see the sound␣
↪documentation for details)
sound_system:
  tracks:
    music:
      type: standard
      simultaneous_sounds: 1
      volume: 0.5
    voice:
      type: standard
      simultaneous_sounds: 1
      volume: 0.7
    sfx:
      type: standard
      simultaneous_sounds: 8
      volume: 0.4
# additionally you need to define some slide styles which are used in the mode
widget_styles:
  medium:
    font_name: pixelmix
    font_size: 8          # for LCDs you need to increase this to 30-40. also change the font above
    adjust_top: 1
    adjust_bottom: 1
  small:
    font_name: smallest_pixel-7
    font_size: 9          # for LCDs you need to increase this to 30-40. also change the font above
    adjust_top: 2
    adjust_bottom: 3
```

Related How To guides
How to design a game in MPF using Modes

Shots

Related Config File Sections
shots:
shot_profiles:
shot_groups:

In MPF, a "shot" is a switch (or combination) of switches that the player shoots for. Examples include:

- A standup target, drop target, or rollover lane
- A ramp, loop, or orbit
- A toy, subway, or VUK

Most shots have lights or LEDs associated with them which are on, off, flashing, and/or certain colors to reflect what "state" the shot is in.

Broadly speaking, a shot is anything the player shoots at during a game. It could be a standup target, a lane, a ramp, a loop, a drop target, a pop bumper, a toy, etc.

In MPF, you define switches (or a sequence of switches) as a "shot". Then whenever that shot is made, MPF posts events which you can use to trigger scores, achievements, shows, etc.

Some shots are made up of a single switch (like a standup target). But you can also configure shots that are only considered to be hit based on series of switches that must be hit in the right order within a certain time frame. For example, you might have an orbit shot with three switches: *orbit_left*, *orbit_top*, and *orbit_right*. You could configure one shot called *left_orbit* that's triggered when the switches *orbit_left*, *orbit_center*, and *orbit_right* are hit (in that order) within 3 seconds, and you could configure a second shot called *right_orbit* that's triggered when the switches *orbit_right*, *orbit_center*, and *orbit_left* are hit within 3 seconds. (So, same switches, but two different shots depending on the order they're hit.)

The beauty of using shots is that you just define all the switches and timing once, and then every time you want to use that shot in your game, you just need to work with the "right_orbit" shot and not have to worry about all the details of the switches and timing.

You can also configure different "states" for shots, e.g. "What state is that shot in?" That can be things like lit, unlit, complete, flashing, etc. You can also configure shows for each state (the unlit state means the light is off, flashing means that the light is flashing, etc.), and you can configure different scoring based on whether state the shot is in (1,000 points if unlit, 5,000 if lit, etc.). All of this is completely configurable.

You can also group multiple shots into "shot groups" and then do certain things when all the shots in the group are in the same state. For example, you could have three standup targets configured as three separate shots that all start in the "unlit" state, but then once all three shots are advanced to the "complete" state, you could add 100,000 points and start another mode.

Shots are also are integrated into MPF's modes system, so you can configure a shot to do different things in different modes.

For example, a ramp shot might do nothing more than score 1,000 points in your base mode, but when the multiball mode is running, that same shot would score a jackpot. You can also configure whether notification of a shot being hit is passed down from one mode to the lower priority modes below it. (In the jackpot example we just mentioned, you probably just want to score the million points for the jackpot if that shot is made while the multiball mode is running and *not* score the 1,000 points for that shot from the base mode even though the base mode is still running under the multiball mode.

Example

This is an example of a shot in a mode:

```
##! mode: inlanes
shots:
  my_shot:
    switch: lane_l
    show_tokens:
      light: lane_l
```

The shot will use the default profile which has the states `unlit` and `lit`. It will start `unlit` and go to `lit` after the first hit. The first hit will post *shot_my_shot_unlit_hit* and the second hit will post *shot_my_shot_lit_hit*. Those events are commonly used to trigger logic based on the shot.

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for multiballs is `device.shots.<name>`.

state Index of the current state. Will start at 0 and increments when the shot advances.

state_name String representation of the state of the shot. Might be 'lit', 'unlit' or whatever is inside your shot_profile.

This is an example:

```
##! mode: inlanes
shots:
  my_shot:
    switch: lane_l
    show_tokens:
      light: lane_l
event_player:
  s_target_active{device.shots.my_shot.state_name=='lit'}: start_multiball
```

In the example the event `start_multiball` will be posted when the switch `s_target` is hit and the shot is in state `lit`.

Related Events

- *(name)_hit*
- *(name)_(profile)_hit*
- *(name)_(profile)_(state)_hit*
- *(name)_(state)_hit*

Grouping Shots for lane change, rotation, etc.

Related Config File Sections
shots:
shot_profiles:
shot_groups:

Example config for lane changing lights.

```
##! mode: inlanes
shots:
  shot_l_outlane:
    switch: lane_l
    show_tokens:
      light: lane_l
  shot_l_inlane:
    switch: lane_a
    show_tokens:
      light: lane_a
  shot_r_inlane:
    switch: lane_n
    show_tokens:
      light: lane_n
  shot_r_outlane:
    switch: lane_e
    show_tokens:
      light: lane_e
shot_groups:
  outlanes:
    shots: shot_l_outlane, shot_l_inlane, shot_r_inlane, shot_r_outlane
    rotate_left_events: s_flipper_left_active
    rotate_right_events: s_flipper_right_active
    reset_events: outlanes_profile_hit_lit_complete
    enable_events: ball_started
    disable_events: ball_ending
```

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for shot groups is `device.shot_groups.<name>`.

common_state The name of the common state of all shots in the group. Will be `None` if there is no common state. State names depend on the profile of your shots (by default `lit` and `unlit`).

Shot Group Overview:

Shot Group:

We're creating a shot group called "outlanes", which contains 4 shots that we defined in our Shots: section of a mode.

Rotate events:

These will cycle the lights thru your shots, based on which flipper button is pressed in this case.

Reset_Events:

Describes an event that will cause this shot group to reset back to its original state.

Enable/Disable Events:

Describe events that will cause this shot group to be enabled/disabled, in this case we are using Ball_Started and Ball_Ending.

Related Events

- *(name)_complete*
- *(name)_(state)_complete*
- *(name)_hit*
- *(name)_(state)_hit*

Shot Profiles

Related Config File Sections
shots:
shot_profiles:
shot_groups:

Shot profiles define how shots will behave when hit. This is an example:

```
##! mode: mode1
shot_profiles:
  my_default_profile:
    states:
      - name: unlit
        show: "off"
      - name: lit
        show: "on"
```

Normally, a shot will advance its profile (unless `advance_on_hit` is set to `False`) and will stay at its last step (unless `loop` is set to `True`). There can be a show with option for every state.

Sequence Shots

Related Config File Sections
sequence_shots:

A sequence of switches which need to be hit in order with a timeout.

This is an example:

```
switches:
  s_ramp_entry:
    number: 1
  s_ramp_success:
    number: 2
sequence_shots:
  ramp:
    switch_sequence: s_ramp_entry, s_ramp_success
    sequence_timeout: 3s
```

When both switches are hit in sequence `ramp_hit` (*(name)_hit*) will be posted. You can use that event to trigger further logic/shows/etc.

Using Sequence Shots in Shot Groups

Sequence shots got shots in their name but they cannot be used in *shot_groups*. If you want to use them in a shot groups create a *shot* which is triggerd on the *(name)_hit* event.

This is an example:

```
switches:
  s_ramp_entry:
    number: 1
  s_ramp_success:
    number: 2
sequence_shots:
  ramp:
    switch_sequence: s_ramp_entry, s_ramp_success
    sequence_timeout: 3s
##! mode: test_mode
# In your mode
shots:
  shot_ramp:
    hit_events: ramp_hit
shot_groups:
  your_group:
    shots: shot_ramp
```

Related How To guides

- *Loops / Orbits / Ramps*

Related Events

- *(name)_hit*

How to integrate shots with shows, lights, sounds, widgets, slides and more

Pinball games need to communicate with the player. Regarding shots this includes two typical things:

1. Indicate the current state of the shot. This is usually implemented by toggling a light or show an image on screen. Normally, the state indication stays permanently until the state changes. It might also permanently play of shot (i.e. to flash an arrow). Additionally, it has to be restored after a player change and sometimes on more restart.
2. On state change (or hit of a shot) the machine needs to communicate success to the player. Usually, this is implemented using a some light show, sound and some animation on screen. Additionally, this often involves scoring and might load start another mode.

To implement (1) we recommend to use *shot_profiles* and create a show per state. This gives you maximum flexibility. Additionally, it will automatically restore the previous state on player change or mode restart (lights/screen will never be out of sync with your state).

To implement (2) we recommend to use *Show player* on the *(name)_hit* event. This allows very flexible animations/scoring/video/sounds and will also automate all cleanup for you.

Config Example

Let us look at a very typical example (from *Rollover Switches*) for a typical inlane/outlane setup with rotation using flipper buttons. Additionally, we want to show the state of the lanes on screen. When shots are hit we want to play a sound, play a show on screen and flash the light of the shot.

First, we will define some switches and lights for inlanes and outlanes. In a mode we define all the shots and tie them to a light. Furthermore, we define an widget name to display for each shot. To color the light and display the widget we define a show and integrate it to our shots using a shot_profile. Eventually, we add a show_player to play animations and sounds when a shot is hit.

```
# this is in your machine-wide config
# first we define some switches + lights
switches:
  s_outlane_left:
    number: 0
  s_inlane_left:
    number: 1
  s_inlane_right:
    number: 6
  s_outlane_right:
    number: 7
lights:
  l_outlane_left:
    number: 0
  l_inlane_left:
    number: 1
  l_inlane_right:
    number: 6
  l_outlane_right:
    number: 7
  gi_left_sling:
    number: 8
  gi_right_sling:
    number: 9
```

(continues on next page)

(continued from previous page)

```
##! mode: my_mode
# put this into a mode
# shots each pass their led and widget to the show define in their shot_profile
shots:
  shot_outlane_left:
    switches: s_outlane_left
    profile: lane_profile
    show_tokens:
      leds: l_outlane_left
      widget: outlane_left
  shot_inlane_left:
    switches: s_inlane_left
    profile: lane_profile
    show_tokens:
      leds: l_inlane_left
      widget: inlane_left
  shot_inlane_right:
    switches: s_inlane_right
    profile: lane_profile
    show_tokens:
      leds: l_inlane_right
      widget: inlane_right
  shot_outlane_right:
    switches: s_outlane_right
    profile: lane_profile
    show_tokens:
      leds: l_outlane_right
      widget: outlane_right
# integrate shots with their show
shot_profiles:
  lane_profile:
    states:
      - name: unlit
        show: "off"           # a default show to turn of the led. change if you want to do something␣
↪on unlit shots
      - name: lit
        show: "shot_lit"      # our show to indicate an lit shot
          # you can add more states here
# to rotate shots and reset them when they are all lit
shot_groups:
  sg_lanes:
    shots: shot_outlane_left, shot_inlane_left, shot_inlane_right, shot_outlane_right
    rotate_left_events: s_flipper_left_active
    rotate_right_events: s_flipper_right_active
    reset_events:
      sg_lanes_lit_complete: 1s
# define a few widgets which show on screen. you can also use images or videos here
widgets:
  outlane_right:
    - type: text
      text: Outlane right lit
  outlane_left:
    - type: text
      text: Outlane left lit
```

(continues on next page)

(continued from previous page)

```
  inlane_right:
    - type: text
      text: Inlane right lit
  inlane_left:
    - type: text
      text: Inlane left lit
shows:
  shot_lit:    # define our show to indicate the state
    - duration: -1      # this show step will run permanently
      widgets:    # show the corresponding widget
        (widget):
          action: add
      lights:     # turn the light purple
        (leds): purple
  shot_hit:    # define our show to communicate success to the player
    - duration: 1s      # this show step lasts 1s
        # add sounds here or videos
        # add scoring here
      shows:      # run another (built-in) show to flash the light
        flash_color:
          show_tokens:
            color: red
            leds: (leds)
          speed: 4
  group_complete:    # define our show to communicate success on completing all shots
    - duration: 1s
        # add scoring, sounds and video
      shows:
        flash_color:
          priority: 10     # higher priority as the shots
          show_tokens:
            color: green
            leds: l_outlane_left, l_inlane_left, l_inlane_right, l_outlane_right, gi_left_sling, gi_
↪right_sling
          speed: 4
# on success flash the sling shot gi on the side of the lane hit and play a sound/video
show_player:
   # play a show once a each shot is lit
  shot_outlane_left_hit{state=="unlit"}:
    shot_hit:
      key: left
      show_tokens:
        leds: gi_left_sling
      loops: 0
  shot_inlane_left_hit{state=="unlit"}:
    shot_hit:
      key: left
      show_tokens:
        leds: gi_left_sling
      loops: 0
  shot_outlane_right_hit{state=="unlit"}:
    shot_hit:
      key: right
      show_tokens:
```

(continues on next page)

(continued from previous page)

```
      leds: gi_right_sling
    loops: 0
  shot_inlane_right_hit{state=="unlit"}:
    shot_hit:
      key: right
      show_tokens:
        leds: gi_right_sling
      loops: 0
  # play a show when the group completes
  sg_lanes_complete{state=="lit"}:
    group_complete:
      loops: 0
```

Related How To Guides
Tutorial step 18: Add your first shot
Shots on inlane/outlanes
Shots in game modes
Implement a Mode for Top Lanes with Multiplier and Scoring
Shots in other modes

Skill Shot

Related Config File Sections
mode:
shots:
shot_groups:
timers:
state_machines:

Types of skill shots:

- Time based
- Hit some target before another target
- Super skill shot
- How to create a lane-change skill shot

A simple skill shot mode:

```
##! mode: skill_shot
mode:
  start_events: ball_started
  stop_events:
    - skill_success
    - skill_failed
  priority: 500
shots:
  skill_l:
```

(continues on next page)

(continued from previous page)

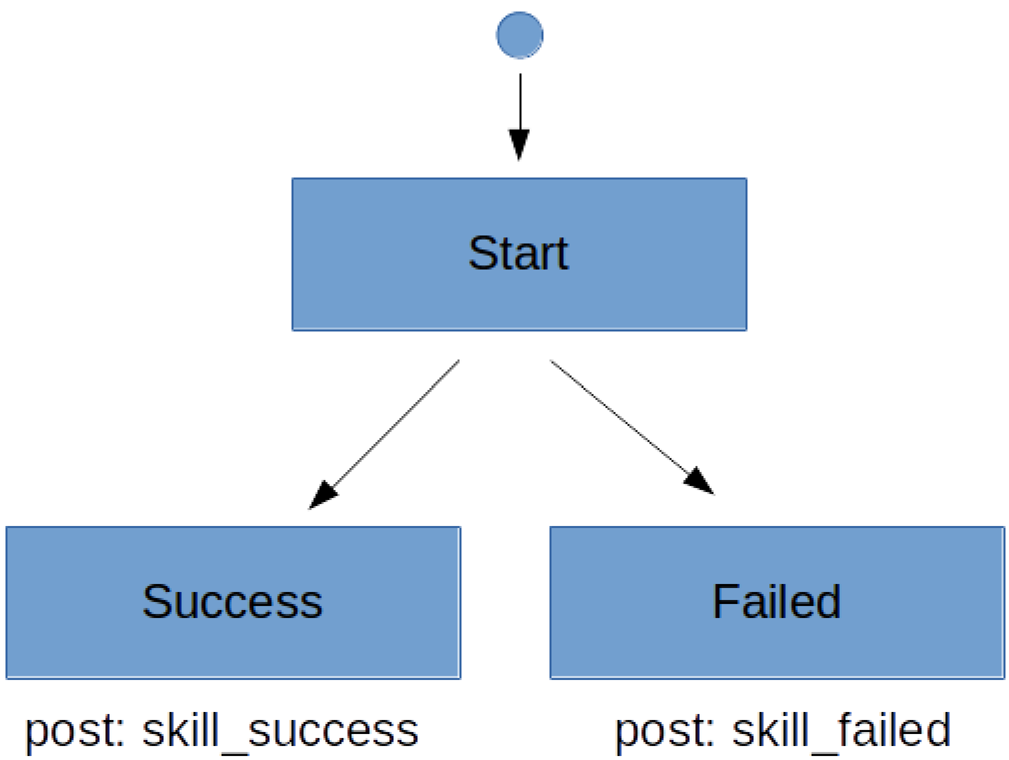
```
    switch: s_lane_l
    profile: skill_shot_profile
    advance_events: mode_skill_shot_started        # replace "skill_shot" with your mode name
    show_tokens:
      light: l_lane_l
  skill_m:
    switch: s_lane_m
    profile: skill_shot_profile
    show_tokens:
      light: l_lane_m
  skill_r:
    switch: s_lane_r
    profile: skill_shot_profile
    show_tokens:
      light: l_lane_r
shot_groups:
  skill_shot:
    shots: skill_l, skill_m, skill_r
    rotate_left_events: s_left_flipper_active
    rotate_right_events: s_right_flipper_active
shot_profiles:
  skill_shot_profile:
    states:
      - name: unlit
        show: off
      - name: flashing
        show: flash_color
        show_tokens:
          color: red
        speed: 4
      - name: lit
        show: on
    loop: true
variable_player:
  skill_success:
    score: 42
timers:
  skill_shot_timeout:
    start_value: 0
    end_value: 5     # set the timeout of your skill shot here
    direction: up
    tick_interval: 1s
    start_running: false
    control_events:
      - action: start
        event: balldevice_plunger_lane_ball_eject_success  # replace "plunger_lane" with the name of
↪your plunger device
state_machines:
  skill_shot_success:
    debug: true
    states:
      start:
        label: Skill shot ready
      success:
```

(continues on next page)

(continued from previous page)

```
      label: Skill successful
      events_when_started: skill_success
    failed:
      label: Skill failed
      events_when_started: skill_failed
  transitions:
    - source: start
      target: success
      events: skill_shot_flashing_hit
    - source: start
      target: failed
      events: skill_shot_unlit_hit, timer_skill_shot_timeout_complete
```

This works the following way: The three shots `skill_l`, `skill_m` and `skill_r` represent the three lanes. `skill_l` starts lit. The group `skill_shot` can be rotated using the flippers. When a lit shot it hit the group posts `skill_shot_lit_hit` and `skill_shot_unlit_hit` when a unlit shot is hit. To prevent races between the two events we use a state_machine called `skill_shot_success` which has three states:



When the mode started it starts at `start`. Then when either `skill_shot_lit_hit` or `skill_shot_unlit_hit` are posted in transitions to success or `failed`. Those states will post either `skill_success` or `skill_failed`. Additionally, there is a timer `skill_shot_timeout` which will fail the skill shot 5s after the ball left the plunger.

Usually, you want to create a modes which starts on `skill_success` and another mode which starts on `skill_failed` to play some shows.

Related How To guides
How to design a game in MPF using Modes

Video Modes

Help us to write it

Scoring

Related Config File Sections
variable_player:

The variable_player is commonly used to score points for the current player when a certain event is posted. This event could be a switch hit (i.e. for *s_your_switch* use the event *s_your_switch_active*).

```
##! mode: mode1
variable_player:
  s_your_switch_active:
    score: 100
```

Furthermore, you can add or set any other player or machine variable. You can also use *dynamic values* here.

It is very common to use multipliers in your game for scoring. The simplest way to implement multipliers is to use a *player variable* to keep the multiplier and multiply it to your scoring entries in *variable_player*. This is an example for simple scoring with multiplier:

```
# set initial value for your multiplier player variable (to have it start
# at 1 instead of 0)
player_vars:
  multiplier:
    value_type: int
    initial_value: 1
##! mode: my_mode
# in your mode:
variable_player:
  increment_multiplier:
    multiplier: 1
  score_something:
    score: 100 * current_player.multiplier
```

The multiplier will be tracked per player and carry over to the next ball. At start we set it to 1 using a :doc:player_vars </config/player_vars> entry in config for every player.

You can also reset the multiplier on every ball if you want:

```
##! mode: my_mode
# in your mode:
variable_player:
  # set initial state on mode start of mode "my_mode"
  mode_my_mode_started:
    multiplier:
      int: 1
      action: set
  increment_multiplier:
```

(continues on next page)

(continued from previous page)

```
    multiplier: 1
  score_something:
    score: 100 * current_player.multiplier
```

Sometimes you want to increase your multipliers after multiple events were posted. For instance, you might want to increase the multiplier after the player completed two shot_groups:

```
# set initial value for your multiplier player variable (to have it start
# at 1 instead of 0)
player_vars:
  multiplier:
    value_type: int
    initial_value: 1
##! mode: my_mode
# in your mode:
accruals:
  bonus_multiplier:
    events:
      - robo_lanes_shots_lit_complete
      - tech_lanes_shots_lit_complete
    events_when_complete: increment_multiplier, light_bonus_2x_led
    start_enabled: true
variable_player:
  increment_multiplier:
    multiplier: 1
  score_something:
    score: 100 * current_player.multiplier
```

You can also combine two (or more) multipliers (see *dynamic values* for details about other possible placeholders and math operators):

```
# set initial value for your multiplier player variables (to have it start
# at 1 instead of 0)
player_vars:
  multiplier:
    value_type: int
    initial_value: 1
  mode_multiplier:
    value_type: int
    initial_value: 1
##! mode: my_mode
# in your mode:
variable_player:
  increment_multiplier:
    multiplier: 1
  increment_mode_multiplier:
    mode_multiplier: 1
  score_something:
    score: 100 * current_player.multiplier * current_player.mode_multiplier
```

You may also just add multipliers instead of multiply them. For instance you could use: `score: 100 * (1 + current_player.multiplier + current_player.mode_multiplier)` and set `initial_value: 0` in `player_vars:` to have them start at 0.

Another option is to use a counter as multiplier using `score: 100 *` (device.counters.multiplier_counter.value + 1). See *dynamic values* for details about possible placeholder.

Sometimes just using math is getting too complicated. For instance, you want to have some special scoring under certain *conditions*. In this case, it is sometimes better to use *conditional events* instead of complicated math formulas in a variable_player.

In this example, we enable special scoring if the super_multiball mode is active and the player made more than two loops (just for the sake of the example - you could also move the scoring into super_multiball and remove the first condition):

```
# set initial value for your multiplier player variables (to have it start
# at 1 instead of 0)
player_vars:
  multiplier:
    value_type: int
    initial_value: 1
  loops_made:
    value_type: int
    initial_value: 0
##! mode: super_extraball
##! mode: my_mode
# in your mode:
variable_player:
  made_loop:
    loops_made: 1
  score_something:
    score: 100 * current_player.multiplier
  score_something{mode.super_extraball.active and current_player.loops_made > 2}:
    score: 1000000
```

Related How To Guides
High Scores
Scoring Based on Logic Blocks
Implement a Mode for Top Lanes with Multiplier and Scoring
How to implement solid state game style score queues in MPF

How to implement solid state game style score queues in MPF

Related Config File Sections
score_queues:
score_queue_player:

When scoring in solid state games the game will typically play chimes while adding the player score and wait after each digit. You can use *score_queues:* and *score_queue_player:* to implement this in MPF.

```
coils:
  c_chime_1000:
    number:
  c_chime_100:
```

(continues on next page)

(continued from previous page)

```
    number:
  c_chime_10:
    number:
score_queues:
  score:
    chimes: c_chime_1000, c_chime_100, c_chime_10,  None
##! mode: my_mode
# in your mode
score_queue_player:
  score_2k:
    score: 2000
  score_200:
    score: 200
```

Tilt

Tilt is a built-in mode. To enable it, just add the `tilt` mode to your machine config list of modes. Additionally, add the `tilt_warning` tag to your *tilt bob switch* and the `slam_tilt` to your slam tilt switch. Tilt runs at all times, since the machine has to look for slam tilts while games are not running.

The tilt mode contains three logic paths:

- Slam tilt (slam_tilt)
- Instant tilt (tilt)
- Tilt warnings (tilt_warning)

You can provide a switch tag or list events for each of them. Let us go over all of them quickly:

The slam tilt is usually triggered by the slam tilt switch at the coin door. It clears all credits and ends the current game.

The *normal* tilt is usually triggered by a *tilt bob switch*. It will give `warnings_to_tilt` warnings until it ends the current ball. The remaining warnings are reset by the `reset_warnings_events` events. By default they are reset on ball end but you can also change it to game end. The warnings count is stored in the player variable configured in `tilt_warnings_player_var` (which defaults to `tilt_warnings`) and you can mess with them using *Variable player* if you like.

Instant tilt is rarely used in normal machines but it might be useful for custom tilt logic or special modes.

Minimal config

The minimal example is to just load the default `tilt` mode:

```
modes:
  - tilt
```

Change defaults

If you want to customize the mode you can also create a tilt mode inside your mode folder (config would be in modes/tilt/config/tilt.yaml):

```
# in your machine config
modes:
  - tilt
##! mode: tilt
# in your tilt mode
tilt:    # the following are the defaults only copy those if you want to change them
  multiple_hit_window: 300ms
  settle_time: 5s
  warnings_to_tilt: 3
```

Add operator settings to service mode

```
# in your machine config
modes:
  - tilt
settings:
  warnings_to_tilt:
    label: Number of tilt warnings
    values:
      0: "no warnings"
      1: "1"
      2: "2"
      3: "3"
      5: "5"
      10: "10"
    default: 3
    key_type: int
    sort: 600
  settle_time:
    label: Time to wait on tilt to settle bob
    values:
      3000: "3s"
      5000: "5s"
      10000: "10s"
    default: 5000
    key_type: int
    sort: 610
  multiple_hit_window:
    label: Tilt sensitivity
    values:
      150: "sensitive"
      300: "normal"
      500: "insensitive"
      1000: "very insensitive"
    default: 300
    key_type: int
    sort: 620
##! mode: tilt
```

(continues on next page)

(continued from previous page)

```
# in your tilt mode
tilt:
  multiple_hit_window: settings.multiple_hit_window
  settle_time: settings.settle_time
  warnings_to_tilt: settings.warnings_to_tilt
```

The tilt modes contains default slides but you can *change them*.

Monitorable Properties

For *dynamic values* and *conditional events*, the prefix for ball devices is `mode.tilt.<name>`.

tilt_settle_ms_remaining Milliseconds until the tilt bob is considered settled.

tilt_warnings_remaining Remaining warnings until the game is tilted.

Related How To guides
How to design a game in MPF using Modes
Overwriting Tilt Slides

Overwriting Tilt Slides

The *tilt mode* comes with very basic slides. You can overwrite them using the following config:

```
##! mode: tilt
# in your modes/config/tilt.yaml
slides:
  _overwrite: true   # this is important to overwrite the existing slides
  tilt_warning_1:
    widgets:
      - type: text
        text: "STOP IT"
    expire: 1s
  tilt_warning_2:
    widgets:
      - type: text
        text: WARNING
        y: top-2
        anchor_y: top
      - type: text
        text: "SERIOUSLY STOP IT"
        y: top-18
        anchor_y: top
        expire: 1s
    expire: 2s
  tilt:
    - type: text
      text: TILT
```

By setting the `_overwrite: true` you will overwrite the complete `slides:` section of the built-in tilt mode. The slides above are the default slides.

Note: You can add a slide for the `slam_tilt` event. However, by default the `tilt` slide is also shown at the same time so you have to make sure that your slide has a higher priority than that slide.

CHAPTER 9

How to design a game in MPF using Modes

This section assumes that you already configured all your hardware devices (especially all your *ball device*). If you did not configure your hardware please do that first. You can go through the *tutorial* or have a look a the *mechs section*.

Video about how to structure your modes:

https://youtu.be/JLgeGBc03bM

Video about *state machines* (often used to implement logic in your mode):

https://youtu.be/qakxTF1H57E

Video about events in MPF:

https://youtu.be/G3UbVP8gFU0

This section is about laying out your modes and actually designing your game logic. It is structured into the following subsections:

Mode Selection and Game Startup

Questions answered in this section:

- How to select modes/players during start?
- How to implement a (timed) skill shot?
- How does a player qualify for a mode?
- How to start the mode?
- Can multiple modes run at once?

Mode Selection

In most machines there are multiple modes which can start but you need to shoot and/or select them first. This usually serves multiple purposes: First, it gives the player options and allows different play styles. Second, it prevents all modes from starting at once. We will create a selection/qualification mode which then starts a *game mode* (or sometimes two). This selection mode usually runs all the time and provides the following functionality:

- Track whether a mode can be qualified/selected or not. Usually you cannot qualify for a mode while a game mode is running.
- **When modes can be qualified:**
  - Indicate the progress on qualification of modes
  - or: Indicate the current selection which would be started
- Start a mode and wait until it is done (no more selection/qualification possible in the meantime)
- Indicate which modes are already completed (often also active during game modes)

If you got multiple modes which can be selected AND started independently you probably need two selection modes which run independently.

We assume that you already defined your *switches* of your shots. Additionally, we assume that you defined *sequence_shots* in case your shots require multiple shots to be hit in order. You should be famililar with the events posted by a successful hit of your playfield shots (those do not have to be defined as `shots` in your config). Usually you will use either `my_switch_active` for a single switch called `my_switch` (e.g. a standup target) or `my_sequence_shot_hit` for a *sequence_shots* called `my_sequence_shot`.

There are generally two types of mode selection:

- Selection by making a shot. This ususally happens during the game.
- Selection using flipper/action/start buttons after hitting a scoop or on ball start. In those cases you have to delay the eject of the ball (see below for an example how to do that).

Please let us know if you got a snippet which might be useful for other users and is missing here. We would be very happy to *include it*.

Common types of selection modes:

Skill shot at ball start

Skill shots typically run on ball start only. See *Skill Shot*.

Select by hitting shot X times

A very common style to qualify and select modes is to light a few shots and once a player has made them a few times start the mode which belongs to the shot. This selection style is used in machines such as Stern Batman DK (2008) or Stern Starwars (2017).

This is an example:

```
##! mode: left_ramp
# mode: left_ramp
mode:
  start_events: start_mode_left_ramp
  stop_events: stop_mode_left_ramp
event_player:
  left_ramp_complete: stop_mode_left_ramp, enable_qualify
##! mode: right_ramp
# mode: right_ramp
mode:
  start_events: start_mode_right_ramp
  stop_events: stop_mode_right_ramp
event_player:
  right_ramp_complete: stop_mode_right_ramp, enable_qualify
##! mode: qualify
# mode: qualify
mode:
  start_events: ball_started
counters:
  left_ramp_qualify_counter:
    starting_count: 0
    count_complete_value: 3
    events_when_complete: disable_qualify, start_mode_left_ramp
    enable_events: enable_qualify
    disable_events: disable_qualify
    start_enabled: true
    persist_state: true
    reset_on_complete: false
    restart_events: reset_qualify_modes
    count_events: left_ramp_hit
  right_ramp_qualify_counter:
    starting_count: 0
    count_complete_value: 3
    events_when_complete: disable_qualify, start_mode_right_ramp
    enable_events: enable_qualify
    disable_events: disable_qualify
    start_enabled: true
    persist_state: true
    reset_on_complete: false
    restart_events: reset_qualify_modes
    count_events: right_ramp_hit
```

This very basic example should be sufficient for a lot of machines. Another option here is to add achievments and have those enable/disable the counters. The advantage of that is that you can use *achievement_groups:* to track completion of combinations modes (e.g. completions of rows in Stern Starwars). You can also do that with *condition events* or *accruals:*.

You probably want to *integrate shows with the logic blocks* next.

Select mode and start by shot

There are multiple options to implement a selection carousel.

Using a carousel

One way to achieve mode selection you use a carousel mode which looks like this:

```
##! mode: carousel
#config_version=5
mode:
  start_events: start_selection_mode
  stop_events: carousel_item_selected
  code: mpf.modes.carousel.code.carousel.Carousel
mode_settings:
  selectable_items: character1, character2, character3
  select_item_events: s_start_active
  next_item_events: s_left_flipper_active
  previous_item_events: s_right_flipper_active
# TODO: add some slides. If you have a nice example please send it to us (or create a PR).
variable_player:
  carousel_character1_selected:
    selected_character:
      string: "character1"
  carousel_character2_selected:
    selected_character:
      string: "character2"
  carousel_character3_selected:
    selected_character:
      string: "character3"
```

A carousel will not currently track which modes are already completed. Also this in this example the carousel will stop after a selection was made. Therefore, we advise to create a second mode to track the progress of your modes.

This might be useful for cases where you want to select characters or general awards which then might influence how fast your modes start. For instance this might be combined with the example above by influencing the *starting_count:* or *count_complete_value* using conditional events:

```
##! mode: qualify
counters:
  left_ramp_qualify_counter:
    starting_count: 2 if current_player.selected_character == "character1" else 0
    count_complete_value: 3
    count_events: left_ramp_hit
```

Using Achivement Groups

You can define multiple *groups* of *achievements* and rotate them:

```
##! mode: left_ramp
# mode: left_ramp
mode:
  start_events: start_mode_left_ramp
  stop_events: stop_mode_left_ramp
event_player:
  left_ramp_complete: stop_mode_left_ramp, enable_qualify
```

(continues on next page)

(continued from previous page)

```
##! mode: right_ramp
# mode: right_ramp
mode:
  start_events: start_mode_right_ramp
  stop_events: stop_mode_right_ramp
event_player:
  right_ramp_complete: stop_mode_right_ramp, enable_qualify
##! mode: qualify
# mode: qualify
mode:
  start_events: ball_started
achievements:
  left_ramp:
    show_tokens:
      leds: l_left_ramp
    show_when_enabled: off
    show_when_selected: flash
    show_when_completed: on
    complete_events: stop_mode_left_ramp
    events_when_started: start_mode_left_ramp
  right_ramp:
    show_tokens:
      leds: l_right_ramp
    show_when_enabled: off
    show_when_selected: flash
    show_when_completed: off
    complete_events: stop_mode_right_ramp
    events_when_started: start_mode_right_ramp
achievement_groups:
  all_achievements:
    achievements: left_ramp, right_ramp
    auto_select: true
    start_selected_events: hit_scoop
    rotate_right_events: s_action_button_active
    enable_events: enable_qualify, ball_started
    debug: true
```

This is a very flexible way to achieve this.

Select a mode at the start of ball 1

Use this to delay the start of a player's first ball until they select a mode:

```
##! mode: start_selecton_on_ball_one
#config_version=5
mode:
  start_events: ball_ended
  stop_events: ball_started
  priority: 100
  game_mode: false   # this is needed to interfere with game start
queue_relay_player:
  player_turn_starting{player.ball==0}:
    post: show_mode_selection       # use this event to enable selection
```

(continues on next page)

(continued from previous page)

```
    wait_for: selection_mode_ended  # make sure you post this event is posted when a selection was made
```

You can replace `player_turn_starting{player.ball==0}` with just `player_turn_starting` to have the selection on every ball (but not on extra balls). If you also want to trigger it on extra balls use `ball_starting`.

Using the start button to select modes

Normally, pressing the start button will cause MPF to add another player. To suppress this during mode selection you can do the following:

```
# Add the following to the game section of your machine's config.yaml
# This will disable the start button for adding players
game:
  add_player_switch_tag: add_player
##! mode: attract
# Add this to your attract.yaml
event_player:
  s_start_active: sw_add_player
##! mode: game_running
# Have something in your base mode to trigger another mode (e.g. the carousel above)
# and in that mode have the following (to reenable the start button):
event_player:
  s_start_active: sw_add_player
```

Game Mode

Questions answered in this section:

- How to track progress inside a mode?
- How does it end?
- Will it always succeed?
- Can it timeout?
- Can it restart if it failed?
- Where will it continue on restart?
- How to implement roll over lanes in a mode?
- How to implement a mystery award mode?
- How to implement a stand-up target bank mode?

Game Modes

Here is a selection of game modes:

- *Weak flipper*
- *Lighting Multiple Timed Shots*

- *Simple Lane Mode*
- *Top Lanes with Multiplier*
- *End the current game by long-pressing start*
- *Delay Ball Start/Stop to Show Slides/Sounds*
- *Provide a Random Mystery Award*.
- *Skill Shot Mode at Ball Start*
- *Carousel Mode*
- *Drain all balls on the playfield and serve one back without ending the current ball*

If you created a unique game mode in your machine which is missing here please consider *contributing a tutorial or example*.

Lighting Multiple Timed Shots at the Same Time

Related Config File Sections
timed_switches:
timers:
event_player:

In this mode you can active shots for 3s by hitting a target. We assume that those shots post `timerx_start`. The mode succeeds when all three shots are active at the same time. Every shot starts a timer and checks if the other two are running.

This is a basic example:

```
##! mode: my_mode
mode:
  start_events: start_my_mode
  stop_events: my_mode_succeeded
timers:
  t1:
    start_value: 3
    end_value: 0
    direction: down
    control_events:
      - action: restart
        event: timer1_start
  t2:
    start_value: 3
    end_value: 0
    direction: down
    control_events:
      - action: restart
        event: timer2_start
  t3:
    start_value: 3
    end_value: 0
    direction: down
    control_events:
```

(continues on next page)

(continued from previous page)

```
    - action: restart
      event: timer3_start
event_player:
  timer_t1_started{device.timers.t2.running and device.timers.t3.running}: my_mode_succeeded
  timer_t2_started{device.timers.t1.running and device.timers.t3.running}: my_mode_succeeded
  timer_t3_started{device.timers.t1.running and device.timers.t2.running}: my_mode_succeeded
```

Implement a Mode for Top Lanes with Multiplier and Scoring

Related Config File Sections
mode:
shots:
shot_groups:
variable_player:
show_player:

This example shows how to make a classic rule used in many games. By making the three top lanes light (J, A, and M), the playfield multiplier is increased from 1X to 2X, 3X, 4X, 5X, and then to 10X. The Right and Left Flipper buttons are used to control a lane change, and ending the ball resets the mode. This example is based on Bally's Heavy Metal Meltdown. The example below creates a new mode called JAM_rollover, and uses a machine-wide player variable named pf_multiplier. This variable is what can be used in other parts of the game logic to multiply values based on the current multiplier value, for example, when calculating end of ball bonuses. The counter value lb_JAM_complete_count is used as the count value in the JAM_lanes_done{count==2} within the variable_player conditional event statements.

```
# in your machine config
##! mode: JAM_rollover
# in modes/JAM_rollover
mode:
  start_events: ball_started
  priority: 110
counters:
  lb_JAM_complete_count:
    count_events: JAM_lanes_lit_complete
    events_when_hit: JAM_lanes_done
    starting_count: 1
    direction: up
    persist_state: false
shots:
  top_lane_J:
    switch: s_top_lane_J
    show_tokens:
      light: l_jam_J
  top_lane_A:
    switch: s_top_lane_A
    show_tokens:
      light: l_jam_A
  top_lane_M:
    switch: s_top_lane_M
```

(continues on next page)

(continued from previous page)

```
    show_tokens:
      light: l_jam_M
shot_groups:
  JAM_lanes:
    shots: top_lane_J, top_lane_A, top_lane_M
    rotate_left_events: s_left_flipper_active
    rotate_right_events: s_right_flipper_active
    reset_events:
      JAM_lanes_lit_complete: 1s
variable_player:
  mode_JAM_rollover_started:
    pf_multiplier:
      int: 1
      action: set
  JAM_lanes_done{count==2}:
    pf_multiplier:
      int: 2
      action: set
  JAM_lanes_done{count==3}:
    pf_multiplier:
      int: 3
      action: set
  JAM_lanes_done{count==4}:
    pf_multiplier:
      int: 4
      action: set
  JAM_lanes_done{count==5}:
    pf_multiplier:
      int: 5
      action: set
  JAM_lanes_done{count==6}:
    pf_multiplier:
      int: 10
      action: set
  JAM_lanes_complete:
    score: 1000 * current_player.pf_multiplier
show_player:
  JAM_lanes_lit_complete:
    flash:
      loops: 4
      speed: 4
      show_tokens:
        lights: JAM_lanes
  JAM_lanes_done{count==2}:
    Playfield_2x_on:
      show_tokens:
        lights: Playfield_2X
  JAM_lanes_done{count==3}:
    Playfield_3x_on:
      show_tokens:
        lights: Playfield_2X, Playfield_3X
  JAM_lanes_done{count==4}:
    Playfield_4x_on:
      show_tokens:
```

(continues on next page)

(continued from previous page)

```
      lights: Playfield_2X, Playfield_3X, Playfield_4X
  JAM_lanes_done{count==5}:
    Playfield_5x_on:
      show_tokens:
        lights: Playfield_2X, Playfield_3X, Playfield_4X, Playfield_5X
  JAM_lanes_done{count>=6}:
    Playfield_10x_on:
      show_tokens:
        lights: Playfield_2X, Playfield_3X, Playfield_4X, Playfield_5X, Playfield_10X
```

Ending the Current Game by Long-pressing Start

Related Config File Sections
timed_switches:

The following snippet will end a running game by long-pressing the start button:

```
timed_switches:
  game_cancel:
    switch_tags: start
    time: 5s
    events_when_active: end_game
```

Please note that this will also work on ball one and will not inhibit bonus nor high_score mode. Let us know in the forum if you need this.

Mystery Awards

Related Config File Sections
ball_holds:
event_player:
random_event_player:
slide_player:
slides:

Mystery awards provide a random award from a list of options while holding the ball.

Holding the Ball

Any *ball_device* can be used to hold a ball while the mystery award display runs with *ball_holds*.

Here is an example of how to use a scoop to hold a ball during a mystery award animation:

```
##! mode: mystery_mode
event_player:
  upper_lanes_complete: enable_mystery

```

(continues on next page)

(continued from previous page)

```
ball_holds:
  mystery_scoop:
    balls_to_hold: 1
    hold_devices: bd_low_scoop
    enable_events: enable_mystery
    disable_events: end_mystery, multiball_active
    release_one_events: end_mystery
```

In the above example, the scoop will only hold the ball when it is enabled with the enable_mystery event. For example, the player needs to complete upper lanes to light mystery. Only when those conditions are met will the scoop hold the ball.

Under disable_events, you can see that the example also prevents the mystery award during multiball. These events allow you to control when you don't want the device to hold on to the ball.

At the end of the mystery award, the ball_hold is disabled and releases a ball.

Providing Random Awards

Once mystery has been lit and the ball enters the device, you can use *random_event_player* to control which awards are chosen.

In the below example, there are four possible awards and the game will make sure each one is provided to avoid doubling-up.

```
##! mode: mystery_mode
random_event_player:
  ball_hold_mystery_scoop_held_ball:
    events:
      mystery_award_1_event: 30 #numbers show probability of event
      mystery_award_2_event: 20
      mystery_award_3_event: 20
      mystery_award_4_event: 30
    force_all: true
```

A random award will only be selected after a ball has been held in the scoop.

Displaying Awards

You can use anything to display an award such as a slide or video. In the below example, a video is used for each award and the scoop will eject the ball after the video has completed.

```
##! mode: mystery_mode
event_player:
  slide_award_1_slide_removed: end_mystery
  slide_award_2_slide_removed: end_mystery
  slide_award_3_slide_removed: end_mystery
  slide_award_4_slide_removed: end_mystery

slide_player:
  mystery_award_1_event:
    award_1_slide:
```

(continues on next page)

(continued from previous page)

```
      expire: 5s
  mystery_award_2_event:
    award_2_slide:
      expire: 5s
  mystery_award_3_event:
    award_3_slide:
      expire: 5s
  mystery_award_4_event:
    award_4_slide:
      expire: 5s

slides:
  award_1_slide:
    - type: video
      video: award_1
  award_2_slide:
    - type: video
      video: award_2
  award_3_slide:
    - type: video
      video: award_3
  award_4_slide:
    - type: video
      video: award_4
```

Full Mystery Award Example

Here is the full example you can use in a mode as a template to start working on your own mystery award.

```
##! mode: mystery_mode
event_player:
  upper_lanes_complete: enable_mystery
  slide_award_1_slide_removed: end_mystery
  slide_award_2_slide_removed: end_mystery
  slide_award_3_slide_removed: end_mystery
  slide_award_4_slide_removed: end_mystery

ball_holds:
  mystery_scoop:
    balls_to_hold: 1
    hold_devices: bd_low_scoop
    enable_events: enable_mystery
    disable_events: end_mystery, multiball_active
    release_one_events: end_mystery

random_event_player:
  ball_hold_mystery_scoop_held_ball:
    events:
      mystery_award_1_event: 30 #numbers show probability of event
      mystery_award_2_event: 20
      mystery_award_3_event: 20
```

(continues on next page)

(continued from previous page)

```
    mystery_award_4_event: 30
  force_all: true

slide_player:
  mystery_award_1_event:
    award_1_slide:
      expire: 5s
  mystery_award_2_event:
    award_2_slide:
      expire: 5s
  mystery_award_3_event:
    award_3_slide:
      expire: 5s
  mystery_award_4_event:
    award_4_slide:
      expire: 5s

slides:
  award_1_slide:
    - type: video
      video: award_1
  award_2_slide:
    - type: video
      video: award_2
  award_3_slide:
    - type: video
      video: award_3
  award_4_slide:
    - type: video
      video: award_4
```

More examples

See *How to design a game in MPF using Modes* and *Other Game Modes* in particular for more examples.

Lane Mode

Related Config File Sections
mode:
shots:
shot_groups:
variable_player:
show_player:

In this How To guide, we're going to look at how you can set up a series of lanes with lights (or standup targets) which you can rotate with the flipper buttons. We'll also look at how you can play a light show when they're complete and assign scoring. "Lane change" is a fairly popular thing in pinball machines, typically with a set of lanes at the top of the machine. They start all off, and then as you roll over them they light up. You can use the flippers to cycle through which lanes are lit, and when they're all lit, you get a score (or increase the bonus multiplier, etc.) For this how to guide we'll use a Williams Indiana Jones machine. Here's a video that shows the final result of building everything we outline in this guide.

See it in action:

https://youtu.be/4Ip60PVe-oQ

Let's begin!

(A) Configure your devices

We'll assume that you already have your switches and lights defined in the `switches:` and `lights:` section of your machine-wide config. (If you have RGB LEDs, you can follow this tutorial also—just substitute `leds:` for `lights:`.)

Next you need to define your `shots`, which is where you pair your switches and lights so you know that Switch A is associated with Light B, and so on.

Do this in your base mode configuration (in `/modes/base/config/base.yaml`), following the documentation for the `shots:` section in the configuration file reference. In Indiana Jones, we've given the lights and switches the same names (which is ok since they're different types of devices), so our `shots:` section looks like this:

```
##! mode: base
shots:
  indy_i:
    switch: indy_i
    show_tokens:
      light: indy_i
  indy_n:
    switch: indy_n
    show_tokens:
      light: indy_n
  indy_d:
    switch: indy_d
    show_tokens:
      light: indy_d
  indy_y:
    switch: indy_y
    show_tokens:
      light: indy_y
```

Next, configure a `shot group`, which is where you can group individual shots together so you can interact with as a single group, like this:

```
##! mode: base
shot_groups:
  indy_lanes:
    shots: indy_i, indy_n, indy_d, indy_y
```

Note that the order of your shots is important since that's how MPF knows the order of them in order to do shot rotation (more on that later.) At this point if you run MPF and start a game, if you hit one of your shots then you should see the light turn on. (How does MPF know this? Because you haven't specified a shot profile for these shots, so MPF uses the default `shot profile` which has them in an unlit state at first and then lights them once they're hit.) Notice that if you hit the flippers they don't rotate, and once you light all the shots they just stay on. We'll change both those behaviors next! Also notice that the states of the shots are stored per-player. If you play and drain a ball, when you start the next ball, the shots will be in the same state before they drained. Also note that if you start a multi-player game, the shots will reset when the second player starts since that player hasn't hit any yet, and when the first player goes to Ball 2, MPF will reset the shots back to what the first player had.

(B) Configure shot rotation

Next, let's configure the shots so that their lit/unlit states rotate (or shift) to the left or right when the player hits the flipper. This step is optional of course. In some situations you might not want your shots to rotate (like the ADVENTURE standups in Indiana Jones where the player has to hit all the shots to light the Path of Adventure). To do this, we have to configure the shot group for rotation events. We configure two different events—one to rotate left and one to rotate right. You can actually configure rotation events in either your machine-wide config or in a mode-specific config. If you do it machine-wide, then the rotation events will always be active. If you configure it in a mode config, then they're only active as long as that mode's active. In this tutorial we're going to configure them in the base mode as well but you could put that group in any other mode and load/unload it as you need it.

```
##! mode: base
shot_groups:
  indy_lanes:
    shots: indy_i, indy_n, indy_d, indy_y
    rotate_left_events: left_flipper_active
    rotate_right_events: right_flipper_active
```

You can specify whatever event name(s) you want for your rotation events. By default, MPF will post *(switch_name)_active* when every switch in the game activates. So in our case, our flipper buttons from the machine-wide switches: section are named `left_flipper` and `right_flipper`. If you named your switch `s_lower_left_flipper_button`, then your event name would be `s_lower_left_flipper_button_active`. Some older pinball machines only rotate lane shots to the right, regardless of which flipper button is pressed. In that case you'd only have an entry for rotate_right_events, but you'd add both the left and right flipper events, like this:

```
##! mode: base
shot_groups:
  indy_lanes:
    shots: indy_i, indy_n, indy_d, indy_y
    rotate_right_events: left_flipper_active, right_flipper_active
```

Of course you can use whatever event(s) you want to rotate the shots. Many System 11 machines had lit shots in the inlanes and outlanes that rotate based on slingshot hits, so in that case you'd set them up and then use `left_slingshot_active` and `right_slingshot_active` as your events (changed based on your actual switch names, of course). Now if you run MPF and start a game, you should be able to light a shot by hitting it and then see it rotate when you hit the flippers. (Note that you have to actually start a game. shots are not active when a game is not in progress.)

(C) Configure your shots to reset when they're complete

If you played with this, you most likely noticed that the shots didn't actually reset once they were all complete. So that's what we'll do in this step. The way we'll do that is to add an entry for *reset_events:* which specifies what events will cause the shots to reset. To do that, go back into your *base.yaml* file and add another setting to your *indy_lanes* shot group for *reset_events:*, like this:

```
##! mode: base
shot_groups:
  indy_lanes:
    shots: indy_i, indy_n, indy_d, indy_y
    rotate_left_events: left_flipper_active
    rotate_right_events: right_flipper_active
    reset_events:
      indy_lanes_lit_complete: 1s
```

There are a few things going on here. First, notice that the name of our event is `indy_lanes_default_lit_complete`. That seems like a mouthful, but it's logical if you break it down! MPF automatically posts events from shot groups based on what's happening in that group. What happens is that every time a shot changes state, the shot group it belongs to checks the state of all the shots in the group. If they are all the same, then it posts a "complete" event which we can use to assign scores, trigger effects, and reset the group. The format of that event is *(name)_(state)_complete*. In our case, our shot group name is `indy_lanes`, and the state of the shots that we're interested in is called *lit*. Also notice that instead of adding `indy_lanes_lit_complete` to the same line as `reset_events`, we put it on its own line along with a time entry of `1s`. This format is available for every device configuration setting where we specify events, and it means that when that event is posted, it will wait for the specified time to pass before actually performing its action. The reason we did this is because without it, the shots will reset themselves instantly when they complete, which might be confusing to the player since it will look like they have 3 of the 4 shots complete, they hit the 4th one, and then they all go out. The player will think, "Wait, what just happened? Did I get it?" So by adding this delay, we wait 1 second after completing all the shots before they're reset. At this point you should be able to launch MPF, start a game, hit a shot, rotate it with the flippers, and when you complete all the shots, they should wait a second and then reset. Cool!

(D) Add some scoring

Next lets add some scoring to your shots. We're going to make it so the player gets 5,000 points if they hit and unlit shot (which will then light), 100 points if they hit a shot that's already lit (since they failed to rotate or nudge the ball into an unlit lane), and 10,000 points when they complete all the shots in the group. To do that, add a scoring section to your base.yaml mode configuration. (Or you can add it to your machine-wide config if you want to keep all your scoring entries in one place.) It should look like this:

```
##! mode: base
variable_player:
  indy_lanes_unlit_hit:
    score: 5000
  indy_lanes_lit_hit:
    score: 100
  indy_lanes_lit_complete:
    score: 10000
```

Again, these event names might seem crazy, but they're all very logical if you break them down. The shot group will post events any time one of its member shots is hit. This is similar to the *complete* event from the previous step, except the *hit event* ends in `_hit` and is posted with every hit to any shot versus the *_complete* event which is only posted when all the shots in the group have made it to the same state. Remember that since we haven't assigned any shot profiles (nor will we), we're using the default shot profile which has two steps: `unlit` and `lit`, with the `unlit` step running a light script that turns off the associated light or LED and the *lit* step running a light script that turns on the light. One anomaly with the scoring is that when you hit the last shot to complete the group, you'll actually get 15,000 points instead of 10,000. That's because when you hit that final unlit shot, you get 5,000 points for hitting an unlit shot plus the 10,000 points for completing the group. If you really only want 10,000 points total on the last hit, then you could just change the `complete` event to 5,000 points, or setup a logic block to track the count and trigger the scoring.

(E) Add a light show to play a cool effect on completion

As it is now, when you complete the lanes, you get the points which is cool, but after 1 second the lights just sort of unceremoniously reset. Boring! So let's create a light show that flashes the lane lights when you complete the lanes. To do this, let's first create a light show (details in Steps A and B *here*) called *indy_lanes_complete.yaml*:

```
##! show: indy_lanes_complete
- duration: 1
  lights:
    indy_i: ff
    indy_n: 00
    indy_d: ff
    indy_y: 00
- duration: 1
  lights:
    indy_i: 00
    indy_n: ff
    indy_d: 00
    indy_y: ff
```

Obviously you can make this show do whatever you want; I opted for a simple one that sort of alternates the lights. Then to run the light show, go back to your *base.yaml* mode config and add a *light_player:* entry which plays this show when the lanes are complete, like this:

```
##! show: indy_lanes_complete
##! mode: base
show_player:
  indy_lanes_default_lit_complete:
    indy_lanes_complete:
      speed: 20
      loops: 10
      priority: 1
```

If you've worked with shows before, these settings should be pretty straightforward. Running this show at 20x the speed means that it runs really fast. We set `loops: 10` so it loops 10 times and then stops. The only slightly confusing thing might be the `priority: 1` setting. Any time priority settings are added to mode config files, the setting is added to the priority of the mode. For example, if you configure your base mode to run at priority 100, that means that everything it does has a priority of 100—slide shows, lights, sounds, etc. Adding `priority: 1` to this light_player entry just means that this light show will run with a priority of 101 instead of 100, ensuring that it shows up "on top" of anything else this mode is doing with those lights.

(F) Revisit your reset delay

At this point you should be all set and your machine's shots should work like the shots in the video at the beginning of this guide. The only loose end to tie up is `reset_events` entry of `indy_lanes_lit_complete: 1s`. As it is now, when the lanes complete (and while the light show is playing), your lanes will still be in their "lit complete" state, meaning if the ball hits a lane within that first second, the player won't get credit for it towards the second round of lighting the lanes. You might want to remove the 1s and just change that entry to `reset_events: indy_lanes_lit_complete`. If you do that and the player's ball hits a lane while the show is playing, then they will get the score and credit towards the next round of lighting the lanes (even though they won't see the lane light until after the show stops since the show is running at a higher priority). Whether you do this is a matter of personal taste. You could also set a stop event for the light show and cancel it right away if the lane is hit again, or you could not have a `priority` entry in the light_player entry so lighting the lane shows up while the show plays around it. Really there are lots of options you can play with.

This is a full example:

```
# switches and lights in your machine config
switches:
  indy_i:
    number: 1
  indy_n:
    number: 2
  indy_d:
    number: 3
  indy_y:
    number: 4
lights:
  indy_i:
    number: 1
  indy_n:
    number: 2
  indy_d:
    number: 3
  indy_y:
    number: 4
##! show: indy_lanes_complete
# the show on complete
- duration: 1
  lights:
    indy_i: ff
    indy_n: 00
    indy_d: ff
    indy_y: 00
- duration: 1
  lights:
    indy_i: 00
    indy_n: ff
    indy_d: 00
    indy_y: ff
##! mode: base
# your base mode
mode:
  start_events: ball_started
```

(continues on next page)

(continued from previous page)

```
shots:
  indy_i:
    switch: indy_i
    show_tokens:
      light: indy_i
  indy_n:
    switch: indy_n
    show_tokens:
      light: indy_n
  indy_d:
    switch: indy_d
    show_tokens:
      light: indy_d
  indy_y:
    switch: indy_y
    show_tokens:
      light: indy_y
shot_groups:
  indy_lanes:
    shots: indy_i, indy_n, indy_d, indy_y
    rotate_left_events: left_flipper_active
    rotate_right_events: right_flipper_active
    reset_events: indy_lanes_lit_complete
variable_player:
  indy_lanes_unlit_hit:
    score: 5000
  indy_lanes_lit_hit:
    score: 100
  indy_lanes_lit_complete:
    score: 10000
show_player:
  indy_lanes_default_lit_complete:
    indy_lanes_complete:
      speed: 20
      loops: 10
      priority: 1
```

Carousel

Related Config File Sections
mode:
mode_settings:
event_player:
slide_player:
slides:

Changed in version 0.33.

- *Doctor Who Carousel*

A carousel allows you to create process for the player to select from a list of items such as selecting a mode to play. The carousel is implemented as a mode. The player can move through a list of items that you provide on the display or cycle through playfield inserts.

This is just one way to select modes. More ways to implement mode selection are described in the *mode selection section* of the *game design documentation*.

A common use of the carousel is to create a mode selection process. For example, the player can scroll through a list of modes on the display. Each mode could be presented to the user as a slide. The player can move from slide to slide using the flippers. Once the player decides which mode to play, he can select the mode by hitting the start button or both flippers at once. This is just one example of how you could implement a carousel as a mode selection process.

There is a reference to a code file in here so be careful to include that reference. You don't need to download any code as it is already in you MPF installation. Here is the process of configuring a carousel:

- Create a mode folder and config file `<machine>/modes/carousel/config/carousel.yaml`
- Add the code to `mode:` section:

```
code: mpf.modes.carousel.code.carousel.Carousel
```

- Create your selectable items. These could be your mode names but you can name them anything for now.

```
selectable_items: terra, pyro, space, liquid
```

- Select the event(s) that choose the item. For example, the start button. You could think of this an the "enter key"

```
select_item_events: s_start_inactive
```

- Select the event that moves to the next item in the list of items

```
next_item_events: s_right_flipper_inactive
```

- Select the event that moves back to the previous item in the list of items

```
previous_item_events: s_left_flipper_inactive
```

Note: It is recommended to use the *flipper_inactive* events to rotate, rather than *flipper_active*. This allows the use of *flipper_cancel* to select items without accidentally rotating before the selection occurs.

Depending on your situation, especially if you use *flipper_cancel* as the select event, you may notice that after cancelling the subsequent *flipper_inactive* events still play sounds or change slides after selection has been made. The Carousel's `block_events` can be used to prevent carousel event handling until one of the `release_events` is posted. If you don't need to do anything after selection, just specifying *block_events* without any *release_events* will help.

```
block_events: flipper_cancel
release_events: both_flippers_inactive
```

There are two events of importance here:

- carousel_<item>_highlighted
- carousel_<item>_selected

You can use the carousel_<item>_highlighted event to display a slide showing the name of the mode to the player.

You can then use the carousel_<item>_selected event to start the mode that was selected by the player.

```
##! mode: my_carousel
# in mode my_carousel
#config_version=5
mode:
  start_events: ball_starting
  stop_events: my_carousel_item_selected
  code: mpf.modes.carousel.code.carousel.Carousel
  use_wait_queue: true
mode_settings:
  selectable_items: terra, pyro, space, liquid
  select_item_events: s_start_inactive
  next_item_events: s_right_flipper_inactive
  previous_item_events: s_left_flipper_inactive
slide_player:
  my_carousel_terra_highlighted: select_terra
  my_carousel_liquid_highlighted: select_liquid
  my_carousel_space_highlighted: select_space
  my_carousel_pyro_highlighted: select_pyro
slides:
  select_liquid:
    widgets:
      - type: text
        text: LIQUID METAL
        font_size: 100
        color: yellow
    transition:
      type: move_in
      direction: right
  select_terra:
    widgets:
      - type: text
        text: TERAFORM
        font_size: 100
        color: yellow
    transition:
      type: move_in
      direction: right
  select_space:
    widgets:
      - type: text
        text: SPACE OUT
```

(continues on next page)

(continued from previous page)

```
        font_size: 100
        color: yellow
    transition:
      type: move_in
      direction: right
  select_pyro:
    widgets:
      - type: text
        text: PYRO
        font_size: 100
        color: yellow
    transition:
      type: move_in
      direction: right
```

Doctor Who Carousel

The following example is based around Bally's Doctor Who. When the player starts a game, the player is shown via a carousel the option to pick eight modes, each representing a certain Doctor. The flipper buttons control the carousel right and left. When the Launch Button is pressed, the game starts the mode selected by the player and launches the ball.

```
#config_version=5
##! mode: carousel
# put this in your modes/carousel/config/carousel.yaml
mode:
  start_events: ball_starting
  stop_events: carousel_item_selected
  code: mpf.modes.carousel.code.carousel.Carousel
  priority: 125
  use_wait_queue: true
mode_settings:
  selectable_items: Doctor1, Doctor2, Doctor3, Doctor4, Doctor5, Doctor6, Doctor7, Doctor8
  select_item_events: sw_launch_active
  next_item_events: sw_right_flipper_inactive
  previous_item_events: sw_left_flipper_inactive
slide_player:
  carousel_Doctor1_highlighted: select_Doctor1
  carousel_Doctor2_highlighted: select_Doctor2
  carousel_Doctor3_highlighted: select_Doctor3
  carousel_Doctor4_highlighted: select_Doctor4
  carousel_Doctor5_highlighted: select_Doctor5
  carousel_Doctor6_highlighted: select_Doctor6
  carousel_Doctor7_highlighted: select_Doctor7
  carousel_Doctor8_highlighted: select_Doctor8
slides:
  select_Doctor1:
    widgets:
      - type: text
        text: Doctor 1
        font_size: 10
        color: yellow
```

(continues on next page)

(continued from previous page)

```
    transitions:
      type: move_in
      direction: right
  select_Doctor2:
    widgets:
      - type: text
        text: Doctor 2
        font_size: 10
        color: yellow
    transitions:
      type: move_in
      direction: right
  select_Doctor3:
    widgets:
      - type: text
        text: Doctor 3
        font_size: 10
        color: yellow
    transitions:
      type: move_in
      direction: right
  select_Doctor4:
    widgets:
      - type: text
        text: Doctor 4
        font_size: 10
        color: yellow
    transitions:
      type: move_in
      direction: right
  select_Doctor5:
    widgets:
      - type: text
        text: Doctor 5
        font_size: 10
        color: yellow
    transitions:
      type: move_in
      direction: right
  select_Doctor6:
    widgets:
      - type: text
        text: Doctor 6
        font_size: 10
        color: yellow
    transitions:
      type: move_in
      direction: right
  select_Doctor7:
    widgets:
      - type: text
        text: Doctor 7
        font_size: 10
        color: yellow
```

(continues on next page)

(continued from previous page)

```
    transitions:
      type: move_in
      direction: right
  select_Doctor8:
    widgets:
      - type: text
        text: Doctor 8
        font_size: 10
        color: yellow
    transitions:
      type: move_in
      direction: right
event_player:
  select_Doctor1: mode_Doctor_1_start
  select_Doctor2: mode_Doctor_2_start
  select_Doctor3: mode_Doctor_3_start
  select_Doctor4: mode_Doctor_4_start
  select_Doctor5: mode_Doctor_5_start
  select_Doctor6: mode_Doctor_6_start
  select_Doctor7: mode_Doctor_7_start
  select_Doctor8: mode_Doctor_8_start
```

Then, each mode that the carousel can start is set up with the following.

```
#config_version=5
##! mode: Doctor_1
##Example:  Doctor_1.yaml
mode:
  start_events: carousel_Doctor1_selected
  stop_events: ball_ended
  priority: 130
##Then the rest of the mode's code.
```

Related How To guides
How to design a game in MPF using Modes

How to Drain All Balls on the Playfield and Serve One Back Without Ending the Current Ball

Related Config File Sections
mode:
ball_saves:
autofire_coils:
event_player:
show_player:
shows:
queue_event_player:
queue_relay_player:

You might want to a have a mode that does not end the current ball when a timer expires, a jackpot is collected, or some other event happens.

When this happens you might want the flippers and possibly other coils to disable (slings, pops, etc) in order to collect the ball/balls.

The first thing you need to do it make sure your flippers, sling, pops, etc have an enable and disable event in their devices config file.

This is an example for left sling:

```
autofire_coils:
  left_slingshot:
    switch: s_left_slingshot
    coil: c_left_slingshot
    disable_events: ball_ending, service_mode_entered, disable_sling
    enable_events: ball_started, enable_sling
```

In the mode that you want to end but not end the ball add a "fake ball save".

In this example this mode had a multiball and the fake ball save is enabled when a multiball ends. In your mode it can be enabled whenever you want it to, mode start or when a shot is hit, etc.

```
##! mode: your_mode
ball_saves:
  fake_MODE_NAME_ball_save:
    enable_events:
      - multiball_<multiball_name>_ended
    auto_launch: false
    balls_to_save: 1
    debug: true
```

When the mode ends either when timer expires or another event happens you should have 2 other modes, another ball save mode and an end_mode mode.

In the ball save mode you will have another ball_save and maybe a show_player that flashes the shoot again light.

We will call this mode ball_save_end_mode.

```
##! mode: your_mode
mode:
  start_events:
    - mode_end_MODE_NAME_started
  stop_events:
    - mode_end_MODE_NAME_stopped
  priority: 9100

ball_saves:
  end_mode_ball_save:
    enable_events: mode_ball_save_end_mode_started
    auto_launch: false
    balls_to_save: 1
    debug: true

show_player:
  ball_save_end_mode_ball_save_enabled:
    fast_flash_show:
      key: end_modes_ball_save_flash
      speed: 3
```

(continues on next page)

(continued from previous page)

```
      show_tokens:
        leds: l_shoot_again
        color: red
      action: play
      priority: 9999
  mode_ball_save_end_mode_stopping:
    end_modes_ball_save_flash:
      action: stop
```

This ball save mode is started when end_mode is started. The end_mode is started by whatever you want the mode you don't want ball to drain end. For example a timer expired or some other event happened.

This is the end_mode. It will disable the flippers and drain the balls. You can display a message on screen or play a video, etc. explaining what just happened. The queue_relay_player will hold the ball until the show is over. When this mode is ending you should enable the coils you disabled.

```
##! mode: your_mode
mode:
  start_events:
    - start_end_MODE_NAME_mode
  stop_events:
    - player_continue_show_ended
  priority: 8150

event_player:
  mode_end_MODE_NAME_started:
    - flipper_off
    - disable_Upper_Left_pop_bumper
    - disable_Upper_Right_pop_bumper
    - disable_Lower_Left_pop_bumper
    - disable_Lower_Right_pop_bumper
    - disable_sling
  player_continue_show_ended:
    - flipper_on
    - enable_Upper_Left_pop_bumper
    - enable_Upper_Right_pop_bumper
    - enable_Lower_Left_pop_bumper
    - enable_Lower_Right_pop_bumper
    - enable_sling
    - start_ANOTHER_MODE

queue_event_player:
  mode_end_MODE_NAME_started:
    queue_event: my_queue_end_MODE_NAME
    events_when_finished: end_end_MODE_NAME

queue_relay_player:
  my_queue_end_MODE_NAME:
    post: start_end_MODE_NAME_intro
    wait_for: end_show_ended
  balldevice_bd_trough_ball_eject_attempt:
    post: wait_for_instruction
    wait_for: player_continue_show_ended
```

(continues on next page)

(continued from previous page)

```
shows:
  end_MODE_NAME_ball_over:
    - duration: 11
      slides:
        end_MODE_NAME_ball_over_slide:
          widgets:
            - type: text
              text: "BALL LOST"
              color: white
              font_size: 80
              y: center + 300
            - type: video
              video: end_mode_video
            - type: text
              text: "DON'T MOVE"
              font_size: 80
              color: red
              x: center
              y: center - 300
              animations:
                show_slide:
                  - property: opacity
                    value: 1
                    duration: .5s
                  - property: opacity
                    value: 0
                    duration: .5s
                    repeat: true
  player_continue_show:
    - duration: 3
      slides:
        end_mode_player_continue_slide:
          widgets:
            - type: text
              text: PLAYER (number)
              color: blue
              font_size: 120
              y: center + 90
            - type: text
              text: Keep Shooting
              color: red
              y: center - 10
              font_size: 90


show_player:
  start_end_MODE_NAME_intro:
    end_MODE_NAME_ball_over:
      loops: 0
      events_when_stopped: end_show_ended
  end_show_ended:
    player_continue_show:
      loops: 0
```

(continues on next page)

(continued from previous page)

```
    events_when_stopped: player_continue_show_ended
```

This is just an example of how I did it in my game. Every game is different.

If you have any questions about how to do this in your game please post to MPF Users Google Group.

Wizard Modes

Questions answered in this section:

- How to track achievements towards one or multiple wizard modes?
- How to start a wizard mode?
- What to do after wizard mode?

Wizard Modes

Unlockable game modes that take over the playfield are typically referred to as "wizard" modes and are often considered milestones for a player's progress. Here we will outline some common approaches to tracking, starting, and completing wizard modes in MPF.

Achievements to Qualify Wizard Modes

The simplest way to track a player qualifying, attempting, and completing wizard modes is through *achievements*, a special type of player variable that progress through a series of pre-defined states.

One common approach for wizard modes is to have a counter that tracks shots to qualify the mode. In this example, hitting three shots will enable and immediately start the wizard mode:

```
##! mode: wizard_qualify
# mode: wizard_qualify
mode:
  start_events: mode_base_started
achievements:
  winterhascome:
    enable_events: logicblock_winteriscoming_count_complete
    start_events: achievement_winterhascome_state_enabled
counters:
  winteriscoming_count:
    starting_count: 0
    count_complete_value: 3
    count_events: winteriscoming_shot_hit
##! mode: winterhascome
# mode: winterhascome
mode:
  start_events: achievement_winterhascome_state_started
```

The different achievement states allow you to fine-tune when and how modes can be qualified. Achievements can be *enabled/disabled*, *selected*, *started/stopped*, and *completed*, and have some rules to help control the flow:

- *disabled* achievements must be enabled before any other state changes will work
- *enabled* achievements can be disabled, selected, or started
- *selected* achievements can be disabled, started, or completed
- *started* achievements can be stopped or completed
- *completed* achievements cannot change state anymore

By combining these state flows with your qualifying shots, mode selections, and end-of-ball events, there is a lot of flexibility for using achievements to track wizard modes. For example, a one-time-only wizard mode could distinguish between an *enabled* achievement (i.e. it hasn't been played) and a *stopped* achievement (i.e. it has been played), while a wizard mode awarding a bonus for accomplishing some goal could distinguish between a *stopped* achievement and a *completed* achievement.

Of course, there's no requirement that achievements be used to start and stop wizard modes. Achievements are a convenience for tracking progress, your own game design may have other approaches.

Starting and Stopping Wizard Modes

Most wizard modes will be started and stopped like any other game mode, using the mode *start_events* and *stop_events*. There will usually be a close relationship between the start/stop events and the achievement state events, as in these typical examples:

```
# Use a counter to enable an achievement
achievements:
  captainschair:
    enable_events: completed_missions_count_hit{value==6}

# Enable an achievement to start a mode [direct event]
mode:
  start_events: achievement_captainschair_state_enabled

# Enable an achievement to start a mode [indirect event]
event_player:
  achievement_captainschair_state_enabled: start_mode_captainschair

# Start an achievement when its wizard mode starts
achievements:
  captainschair:
    start_events: mode_captainschair_started

# Complete an objective to complete an achievement [direct event]
achievements:
  captainschair:
    complete_events: logicblock_captainshots_counter_complete

# Complete an objective to complete an achievement [indirect event]
achievements:
  captainschair:
    complete_events: captainschair_complete
event_player:
  logicblock_captainshots_counter_complete: captainschair_complete
```

(continues on next page)

(continued from previous page)

```
# Stop an achievement when a mode stops
achievements:
  captainschair:
    stop_events: mode_captainschair_will_stop
```

For wizard modes that stop other game modes, disable qualifier shots or ball locks, and/or have other "takeover" behaviors, consider using *Mode Layering* to handle the transitions in and out of wizard modes.

After a Wizard Mode

Most wizard modes are only played once and have a "completion" goal for the player to accomplish. Mid-game wizard modes (also called "mini-wizard" modes) will usually end if the goal is completed, while end-of-game wizard modes play until the ball drains. Similarly, end-of-game wizard modes typically restart immediately on the players next ball while mid-game wizard modes usually do not. Multiball wizard modes usually remain active until only one ball is left in play.

Achievement states are an excellent way to track how a wizard mode ended and whether it impacts future game behavior. If you're using the *achievement_(name)_started* event to start your wizard mode the `restart_after_stop_possible:` setting determines whether a "stopped" achievement can be started and the `restart_on_next_ball_when_started:` setting will post the *achievement_(name)_started* event when its parent mode starts. If the wizard mode has a "completion" goal, the achievement's "completed" state can be used to track whether a player accomplished it.

Ball End Modes

Questions answered in this section:

- How to start a mode after the ball for a player drained?
- How to implement a bonus mode?

Ball End Modes

Certain modes typically run on game end. MPF has a lot of built-in modes for this purpose. You can omit any of them or replace them with your own mode.

Ball end modes delay the ball ending process. If you want your own mode to delay the ball ending process you can start the with the following config:

```
##! mode: custom_bonus
#config_version=5
mode:
  start_events: ball_ending     # start on ball ending process
  use_wait_queue: true          # delay ball ending
  priority: 500                 # determines the order of ball end modes
  stop_events: stop_my_mode     # post this event to stop the mode and continue the ball ending process
```

Ball ending will be delayed until your mode stops so make sure that your mode ends eventually or the game will be stuck. In the example above your config need to post stop_my_mode or, if you are writing code, stop your mode in code.

Showing slides on mode end

See *Ball Start and End Behaviour*.

Bonus Mode

Score multipliers and evaluate them into a bonus at the end of the ball. See *End of Ball Bonus* for details.

Game End Modes

Questions answered in this section:

- How to start a mode after the last player drain his ball?
- How to implement a highscore mode?
- How to implement a match mode?

Game End Modes

After the last ball of the last player ended (and all modes which blocked ball ending ended) the game ending sequence will run. A few modes typically exist which delay game ending and are built-in to MPF.

If you want to implement your own game end mode use this template:

```
##! mode: custom_high_score
#config_version=5
mode:
  start_events: game_ending     # start on game ending process
  use_wait_queue: true          # delay ball ending
  game_mode: false              # the game is no longer running at this point
  priority: 500                 # determines the order of game end modes
  stop_events: stop_my_mode     # post this event to stop the mode and continue the game ending process
```

This example will block the game ending process until you post stop_my_mode in your config or stop the mode from code.

Start Mode After Last Ball of Every Player

Alternatively, you can use *Queue Relay player* to achieve the same as above. In this example we start a mode after the last ball of every player (but you can also use game_ending as above). Put this into your base mode to start your custom mode on the end of ball three (or remove the condition to start if after every ball):

```
##! mode: base
queue_relay_player:
  ball_ending{current_player.ball==3}:
    post: start_your_mode
    wait_for: mode_your_mode_stopped
```

Ending the Game by Long-Pressing Start

See *Ending the Current Game by Long-pressing Start*.

Highscore Mode

Allow players to enter their initials on high score. See *High Scores* for details.

Match Mode

Evaluates a match with the end of the player score. Typically awards a credit on match, See *Match Mode* for details.

Other modes

Questions answered in this section:

- Which modes run outside of a game?
- How to control attract?
- How do credits work?
- How does tilt work?
- What is the service mode?

Other Game Modes

There are a few very typical modes in almost all machines. Those either run at start or all the time.

All the time/Before ball start

Those modes run all the time or before ball start.

Credits Mode

Count coins and denies game start on insufficient credits. See *Coins & Credits* for details.

Attract Mode

Attract mode stop on game start. See *Attract (mode)*.

Tilt Mode

Tilt usually run the whole time. It will end the game on tilt and might remove credits on slam tilt outside of a game. See *Tilt* for details.

Service Mode

See *Service Mode* for details.

Ball End Modes

See *Ball End Modes*.

Game End Modes

See *Game End Modes*.

Layering Modes Example

Examples given in this section:

- How to define mode categories and helper modes
- How to move in and out of game and wizard modes
- How to track and persist progress outside of modes

Layering Modes Example

One of the major difficulties in designing a new game is managing the interrelationship between different game modes. When considering how to structure your game, it can be helpful to categorize your modes based on how much they "take over" the playfield. When your modes are categorized, you can create helper modes to manage the starting and stopping of game modes.

For the purposes of demonstration and to help you start thinking about how you might layer your own game, let's look at a breakdown of one approach to mode layering.

Gameplay Modes

While every pinball game has unique characteristics, many games can be analysed based on three categories of game modes:

- **Field Modes** are nonintrusive modes that run when no wizard or mission modes are active and are typically used for accruals, multipliers, and shots to qualify for other modes. All field modes are run together.
- **Mission Modes** are "partial takeover" modes that ask for the player's attention but allow other gameplay mechanics to continue. Typically, a mission mode will disable qualification/starting of other mission modes but won't impact multiball locks, pop bumper awards, and other progressions. Examples include:

- Attack Wave, Shoot the Martians *(Revenge from Mars)*
- Trolls *(Medieval Madness)*
- Catch the Robbers *(Dirty Harry)*
- House Challenges *(Game of Thrones)*

- **Wizard Modes** are "complete takeover" modes that stop nearly all gameplay mechanics and force the player to focus on that mode exclusively. Examples include:
  - Multiball modes and video modes *(all games)*
  - Rooftop Chase *(Whodunit)*
  - Khan Battlefield *(the Shadow)*
  - G-R-E-Y Attack *(Congo)*
  - Hand of the King, Iron Throne, Winter Has Come *(Game of Thrones)*

Note: a partial takeover mode is commonly referred to as a "game mode", but here we will call it a "mission mode" because **"game" is a very specific mode in MPF** *and game.yaml is a file that we don't want to interfere with.*

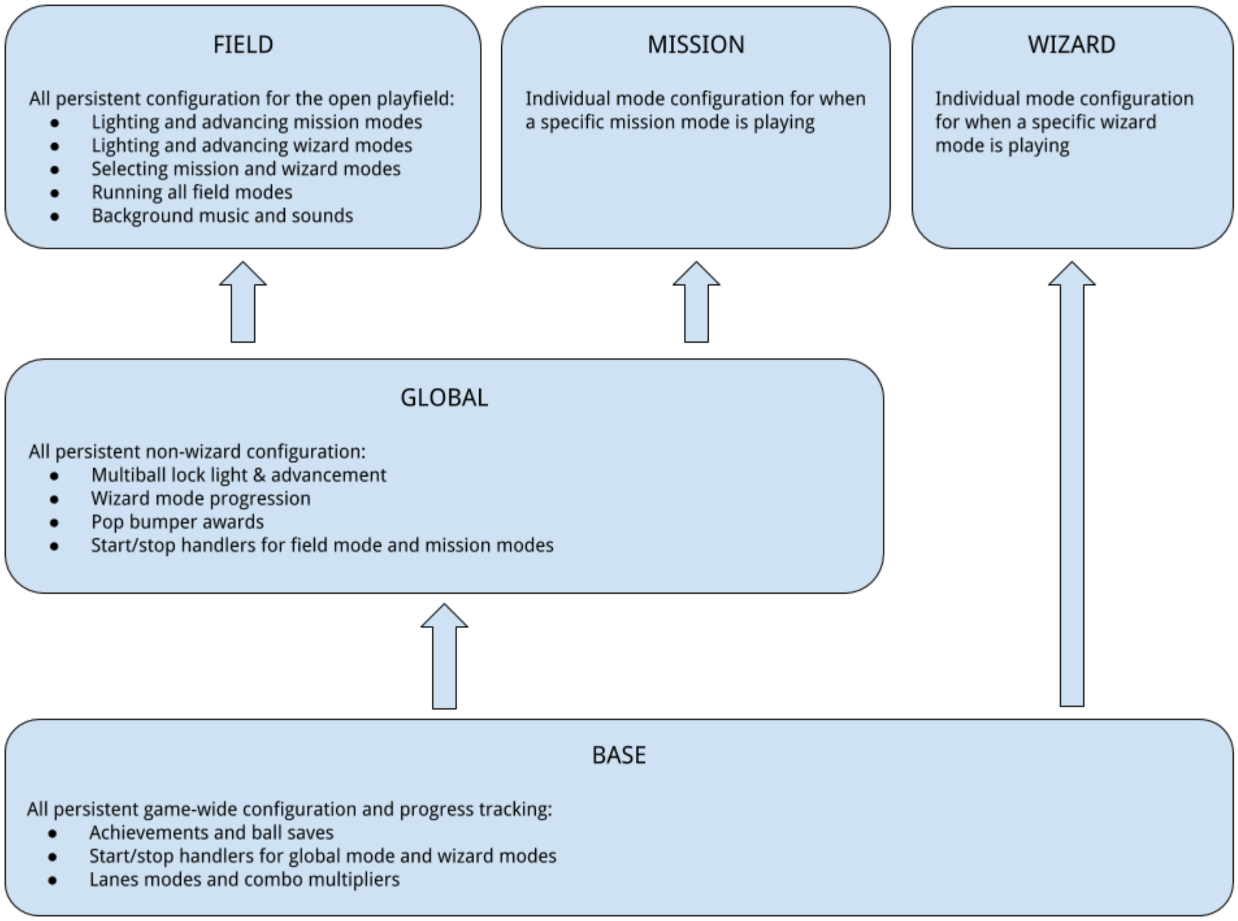
Helper Modes

To facilitate the transition between Field, Mission, and Wizard modes, three helper modes can run underneath the current gameplay:

- **Field Mode** *(field.yaml)* consolidates all of the field modes so that starting/stopping them can be managed by a single event handler. For clarity, it helps to consider the helper mode and all the individual field modes to be one single mode (and we will structure the code in this way).
- **Global Mode** *(global.yaml)* manages transitioning between field mode and mission modes and tracks any accruals/qualifiers that can be advanced while a mission mode is running, for example:
  - Pop bumper countdowns/awards
  - Multiball lighting & locking
  - Mission mode qualifying and selection
- **Base Mode** *(base.yaml)* is the default MPF background mode and manages transitioning between global mode and wizard modes. Base mode is also responsible for any always-persistent tracking, for example:
  - Achievements & specials
  - Ball saves
  - Combo multipliers

Mode Relationship Diagram

In this typical layering configuration, the base mode starts when a player's turn starts and ends when that player's turn ends. By default, the global mode starts when the base mode starts and the field mode starts when the global mode starts. As a result, the typical player turn starts with field mode (a.k.a. on an open playfield).



Field and Mission modes are mutually exclusive: the field mode stops when a mission mode starts, and starts again when the mission mode stops. The global mode runs throughout and manages this transition.

Global and Wizard modes are mutually exclusive: when a wizard mode starts the global mode stops (and with it, the field or any mission modes also stop), and global starts again when the wizard mode ends. Base mode runs throughout and manages this transition.

Starting and Stopping Layers

For a successful layering, each helper mode depends on some particular coding.

Field modes always run together, so the simplest way to manage them is to separate the various field mode behaviors into different yaml files and import all of them into the field helper mode. This keeps each file small while giving just a single mode to start and stop.

```
##! mode: field
# modes/field/config/field.yaml

mode:
  start_events: start_mode_field
  stop_events: stop_mode_field
```

(continues on next page)

(continued from previous page)

```
config:
  # add your mode parts here. For instance:
  # - field_mission_qualifier_shots.yaml
  # - field_miniwizard_qualifier_shots.yaml
  # - field_chase_advancement.yaml
  # - field_dropbank_special.yaml
```

Mission modes replace field mode, usually on their own but you may want to allow two or more missions to run concurrently. Giving every mission mode a few common event handlers allows the global mode to easily manage the transitions into and out of mission modes.

```
##! mode: trolls
# modes/trolls/config/trolls.yaml

mode:
  start_events: start_mode_trolls
  stop_events: stop_mode_trolls, stop_missions
  events_when_started: mode_type_mission_started
  events_when_stopped: mode_type_mission_stopped
```

Global mode can import global-specific config files to consolidate all persistent behavior (just like field mode), and uses special events to handle transitioning between field mode and mission modes. Global will automatically attempt to restart field when a mission mode stops, so we add a special handler: stop global mode when the ball ends, and only restart field mode if global isn't stopping.

```
##! mode: global
# modes/global/config/global.yaml

mode:
  start_events: start_mode_global
  stop_events: stop_mode_global, ball_will_end

config:
  # add your configs here. For instance:
  # - global_multiball_madness_light_and_lock.yaml
  # - global_pop_bumpers.yaml
  # - global_wizard_qualifier.yaml

event_player:
  mode_global_started:
    - start_mode_field
  mode_global_will_stop:
    - stop_mode_field
    - stop_missions
  mode_type_mission_started:
    - stop_mode_field
  mode_type_mission_stopped{not mode["global"].stopping}:
    - start_mode_field
```

Wizard modes replace global, and use a special set of event handlers just like the mission modes.

```
##! mode: madness
# modes/madness/config/madness.yaml
```

(continues on next page)

(continued from previous page)

```
mode:
  start_events: start_mode_madness
  stop_events: stop_mode_madness, stop_wizards
  events_when_started: mode_type_wizard_started
  events_when_stopped: mode_type_wizard_stopped
```

Base mode runs for the player's entire turn and includes special handlers to manage the transition between global mode and wizard modes. Just like with global restarting field, base mode restarts global mode when a wizard mode stops (unless base mode itself is stopping).

```
##! mode: base
# modes/base/config/base.yaml
event_player:
  mode_base_started:
    - start_mode_global
  mode_base_will_stop:
    - stop_mode_global
  mode_type_wizard_started:
    - stop_mode_global
  mode_type_wizard_stopped{not mode["base"].stopping}:
    - start_mode_global
```

CHAPTER 10

Displays, DMDs, & Graphics

Every electronic pinball machine has some type of display, whether it's 1980s-style 7-segment numeric displays, an early '90s-style alphanumeric display, a mono dot matrix display (DMD), a full color "RGB" DMD, or a modern LCD (which itself can either be a small LCD, like a "color DMD", or a huge one like what Jersey Jack has in the backbox of The Wizard of Oz and The Hobbit).

The MPF media controller is designed so that it can support all types of these displays, including multiple different types of displays at the same time. It supports text, drawing shapes, images, and videos. You can position any combination of these on the display at any time, and you can set layering and transparencies. You can use standard TrueType fonts. You can also apply animations, motions, and transitions to your displays and their widgets. And, like just everything else in MPF, you can do most of your display configuration via the config files.

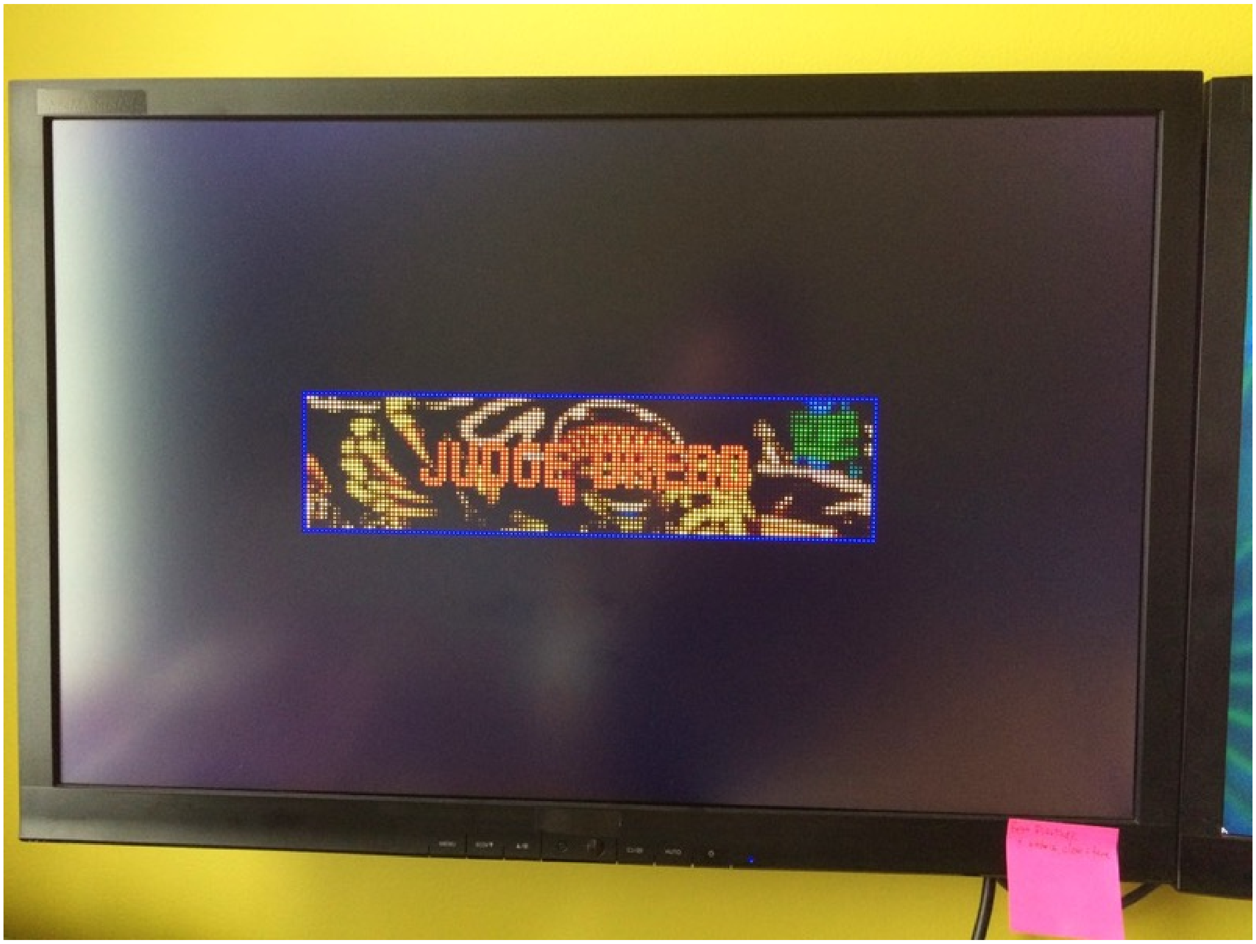
Note: Everything in this "Displays & Graphics" section is about the default MPF Media Controller. See the *media controller* section for details and alternative implementations.

Here are a few photos of the MPF Media Controller's display system in action. These were all created with configuration files and without manual programming.

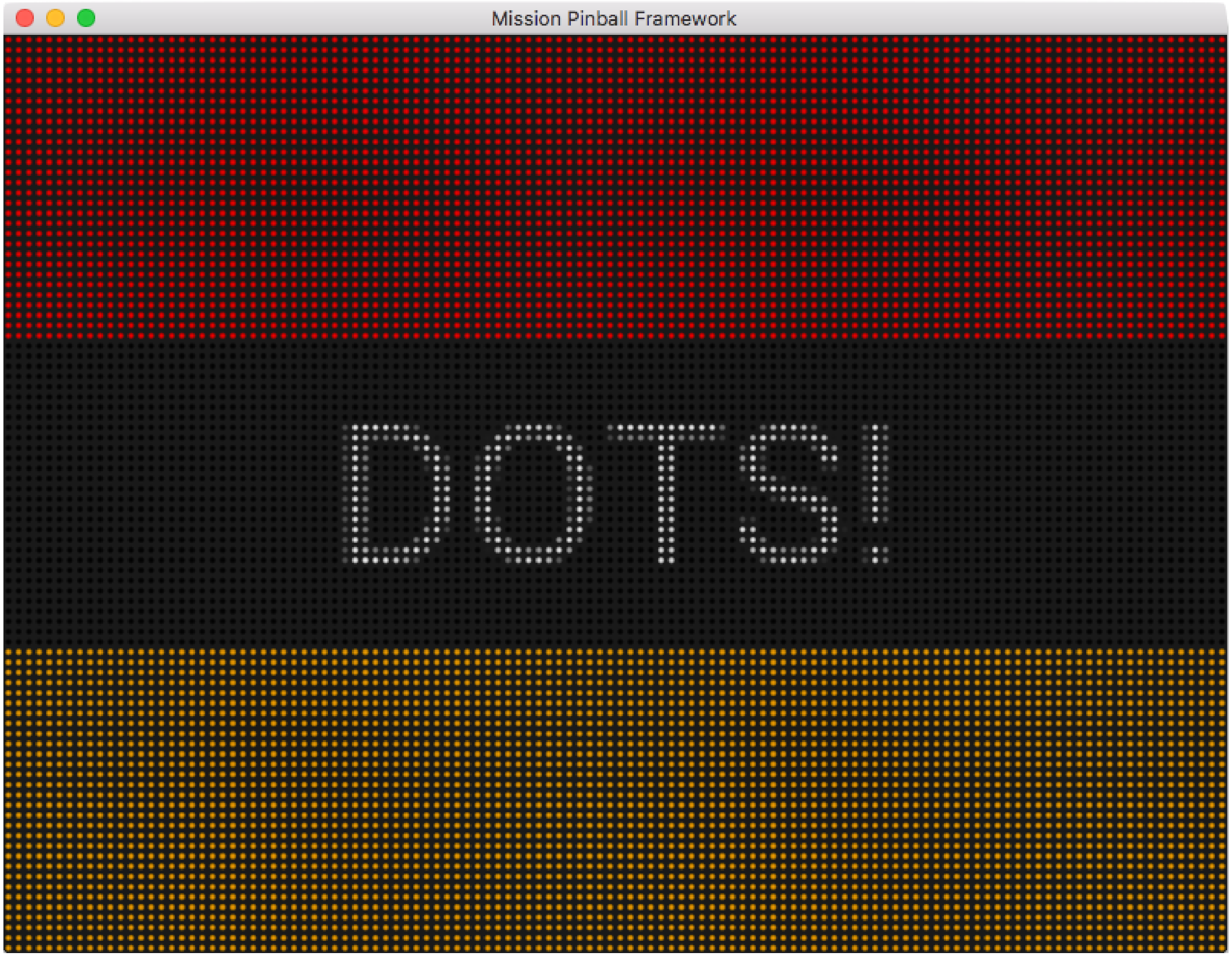
Here's a traditional *single-color / mono DMD*:

Here's an on-screen window (or what many people called an "LCD" display. In this case, it's showing on single *color DMD* virtually with no "dot" filter applied, along with other on-screen content:

Here's a "color" DMD on an LCD monitor. It's showing a 128x32 window of color content, with a "dot look" filter to make it look like dots.



Here's a full-size window with the dot filter applied:



Here's a full-color RGB DMD LED matrix. (So it's like a color DMD, but a matrix of 2.5mm RGB LEDs rather than an LCD):

Before we go into the details of all the various display components, let's start with an overview of how the MPF display architecture works. (If you don't care about the details and just want to start using your display, you can jump directly into our *step-by-step tutorial* which covers how to get your display running.)

Additionally, MPF also supports *segment displays and alpha numeric displays*. Both physically and virtually. Another type of displays are *score reels* which can also be controlled.

Related Events

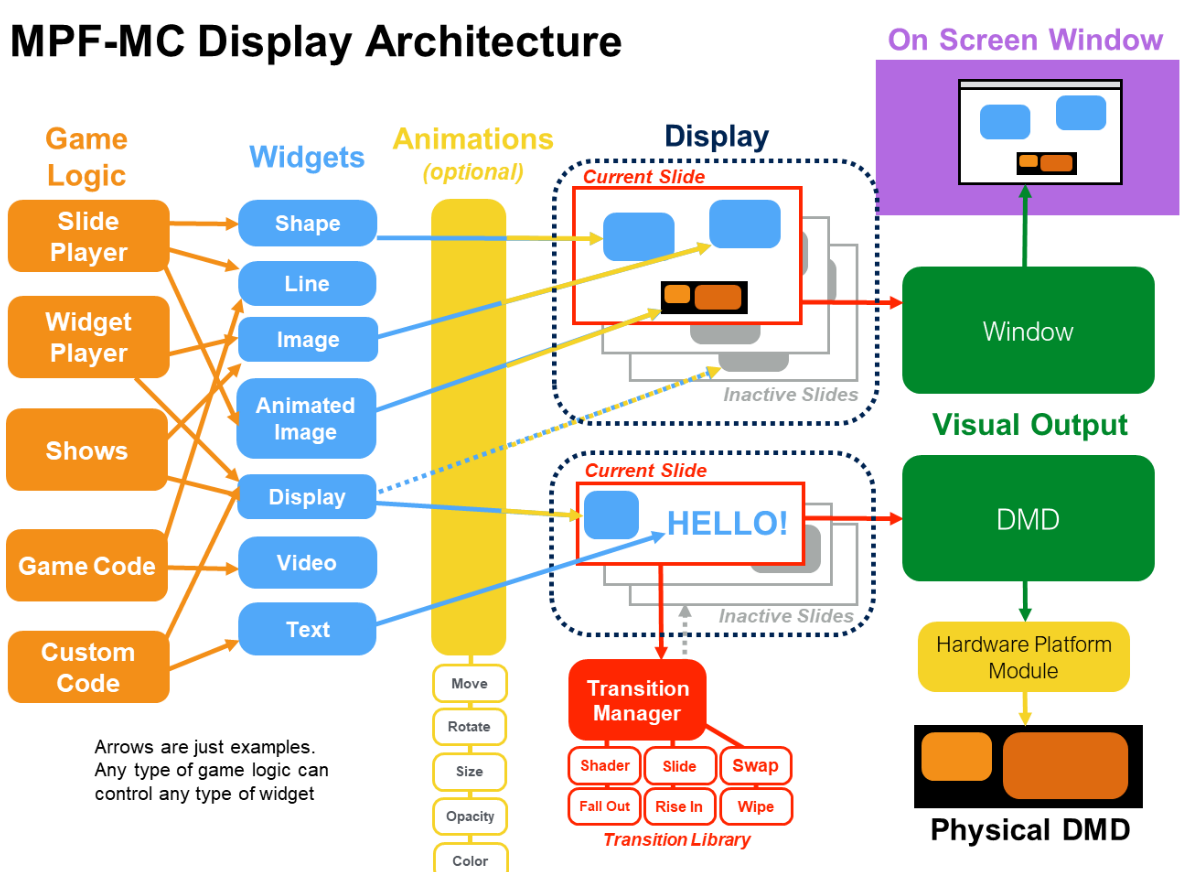
- *display_(name)_initialized*
- *display_(name)_ready*

Display Concepts & Architecture

The MPF Media Controller uses the same core architecture to power all kinds of displays, regardless of whether it's a DMD (physical or virtual, monochrome or color), an LCD (on screen window displays), or a combination of both.

The MPF Media Controller's display system is based on Kivy (a multimedia programming library) and uses technologies like SDL2 and Gstreamer under the hood.

Here's an architecture diagram which details how the MPF Media Controller's display system works. It's kind of complex to look at, but we'll to step through it piece-by-piece. The good news is that you don't have to understand all of it to use MPF. (You can follow our step-by-step tutorial to get your display up and running just with a few config file entries.) But as you start to create more advanced display effects, it will be helpful to understand how everything fits together.



The major components of the MPF Media Controller's display system are:

Window

Every MPF-MC application has one (and only one) window. It is the fundamental graphical element that maps directly to a graphical window on the host operating system. If you do not provide a "window:" section in your config, a default window will be created for you (800 x 600 pixels). The size settings (width, height) control the dimensions (in pixels) of the host operating system window that will be created. Various settings in the *window:* section control the appearance and behavior of the main on-screen window which is created by MPF-MC. These settings include things such as whether or not the window has a border, is full screen, or whether special image processing is applied to the window using effects. These effects perform image processing to the source image of the window and can be used to get an old-school "DMD look" or "color DMD" look to your window as well as other special effects. Windows can be used in any monitor configuration (portait or landscape) and will attempt to center themselves left/right and top/bottom. Windows always use the left lower corner as the 0,0 location.

Displays

Before anything can actually be shown on the window, it must first be drawn in a display. Displays are an internal representation of a blank canvas that holds graphical content. It is important to not confuse these displays with physical hardware displays (like an LCD monitor or a DMD). These displays can be shown on such physical devices, but there is not necessarily a one-to-one mapping between them. One of the most import features of displays are they are targets for showing slides (you can think of them as slide managers). The MPF-MC can have multiple display canvases at the same time, and you can map different ones to different physical displays. You can even create sub-displays where one display has a small region which is another display (kind of like picture-in-picture). The most important setting for a display is its size (with, height) in pixels. If you do not specify any displays in your "displays:" section of your config files, a default display (800 x 600 pixels) will be automatically created for you. It is important to remember that displays always use the left lower corner as the 0,0 location.

Slides

Every display has a list of "slides", (which are the same height and width of the display). One slide is "active" at a time, meaning it's the slide that's showing. Think of these like slides in a slide projector. You'll probably end up with hundreds of slides, but only one is showing at a time. You can use transition effects to switch from one slide to another (these are things like sliding in, pushing, fading, flipping, etc.).

Widgets

Widgets are the "things" you actually put on slides. There are lots of different types of widgets, including text, images, videos, shapes, etc. Different widgets have different properties, like their x,y position on the slide, their size, color, etc. You can position widgets on slides with pixel-level accuracy, or you can use relative positions like "10% down from the top edge", or "centered", or "25% to the left of center", etc. Using relative positions means that your display will be resolution independent.

You can also animate the properties of a widget. For example, a widget could start out at the bottom of the display and then move to the center, or you can animate the size, color, scale, rotation, or the opacity, or pretty much anything other widget property you desire. You can chain together multiple animations to run back-to-back, or you can configure multiple animations to happen at the same time. You can even configure the "curve" of the formula that's used to animate widgets, so you can have them smoothly accelerate and decelerate, or slow down as they're animating, or pop into place, etc.

Display Widget

One widget type of special importance (and deserving of its own introductory paragraph) is the display widget. The display widget is used to show the contents of a display on the screen (remember from above a display is just an in-memory drawing canvas, the display widget allows its contents to be shown). The main window automatically creates a display widget that has the same dimensions of the window. Now if you only require a simple graphical layout where one slide is shown at a time and fills the entire window, then you need not concern yourself any further with the display widget. However if you require a more advanced layout, the display widget will enable you to accomplish that. Display widgets can also be configured to apply special image processing to the contents of the display. These effects can be used to get an old-school "DMD look" or "color DMD" look to your window as well as other special effects.

All these concepts come from PowerPoint. :)

The original creators of MPF have day jobs that require them to spend a lot of time with PowerPoint! If you've ever used PowerPoint, you should notice that we used PowerPoint (or Keynote or whatever presentation software you like) as the conceptual model for MPF's display system. In PowerPoint, your content is a series of "slides." Each slide contains one or more "elements (widgets)". Those elements can be text, images, videos, drawing shapes, etc. Each element has a "size" (length & width), a "position" on the slide (x,y coordinates), a "layer" which controls how it overlaps with other elements, alpha transparencies, and animation effects (blink, sparkle, move, etc).

And even though your entire PowerPoint presentation is made of of lots of slides, only one slide is active on your "display" at a time. Then when you change to another slide, you can have nice animated "transitions" from one slide to the next.

So if the MPF display system seems kind of complex, just think of it like a giant PowerPoint presentation and it should all hopefully make sense. Now let's start digging into some of the details of each of the parts of the display system.

You can read more about the *internals of the media controller*.

Working with Displays

The first step to setting up a display in MPF is to use the `displays:` section of your machine-wide config to create a list of displays.

Note that the Tutorial includes a walk-through of setting up your first display. So if you just want to get it up and running quickly, check out the *tutorial* instead and then come back here for the nitty-gritty details later.

Here's a very simple example that creates a display called "window" with a height and width of 800x600:

```
displays:
  window:
    width: 800
    height: 600
```

You can name your display whatever you want. For example, here's a display called "potato" which is 100x100:

```
displays:
  potato:
    width: 100
    height: 100
```

You can add multiple displays to your config. Here's an example with a display called "lcd" which is 1366x768, and a second display called "playfield" which is 640x480:

```
displays:
  lcd:
    width: 1366
    height: 768
    default: true
  playfield:
    width: 640
    height: 480
```

The "lcd" display above also has a setting `default: true`. As you can imagine, when you have more than one display, then when you are setting up content to be shown on the display, you have to specify which display you want it to show up on. Picking one display to be your default is the display that's used for content where you don't explicitly set which display you're using.

Note: Full details and options for these displays are available in the *displays: section* of the config file reference.

These "displays" are logical, not physical!

One concept that's somewhat confusing for new users is that the displays you set up here are not yet tied to physical displays in your pinball machine. You can think of these as "logical" displays which you can use in your config files and game code. But when it comes to using a physical display, you have to "link" the physical display hardware to one of these logical displays.

One final note about the displays you specify in your `displays:` section: The size (height and width) of your displays here are independent from the actual physical displays (windows and DMDs). For example, the size of the on-screen window is specified in the `window:` section of the machine config (which is 800x600 by default). So if you change the size of your display here (perhaps to 320x240), then the on-screen window will still be 800x600, and the content of the display canvas will be 320x240 (but scaled up to the 800x600 window). This means that MPF is "resolution independent", in that you can build your game for a certain display size and then scale it up or down to fit on whatever physical display is there later.

Let's walk through some examples of how you can actually configure various displays. You don't have to read through all of these—just pick whichever display type you want to use in your machine.

Using an LCD for a display

This guide will show you how to use an on-screen LCD window for your main display. This would be like what Jersey Jack does in Wizard of Oz or The Hobbit.

Here's what the final version of the relevant sections of your machine config file will look like. We'll step through everything one-by-one.

```
displays:
  window:
    height: 600
    width: 800
window:
  height: 600
  width: 800
  title: Mission Pinball Framework
  resizable: true
  fullscreen: false
  borderless: false
  exit_on_escape: true
  source_display: window
```

1. Add your display

The first part of the config file is where you create your display called "window" and set its size:

```
displays:
  window:
    width: 800
    height: 600
```

This is just like we covered in the *Working with Displays* section.

2. Add your window configuration

Next you need to add a section to your machine config file which has the settings for the actual on-screen popup window. This is configured in the `window:` section.

Most of the settings here are pretty self-explanatory. The most important thing is the `source_display: window` section which is where you specify which display (from the `displays:` section of your config) will provide the actual source content for your on-screen window.

(That said, if you only have one display, or if you have a display called "window", then the on-screen window will automatically use that display for its source, but we're just including it here for completeness.)

The other important thing to point out is that you have to specify the size of your display and your window separately. In the example above, we have an 800x600 window showing the content from an 800x600 display. But we could, for example, set the display to 400x300 while keeping the window at 800x600. In that case, the display content would be "scaled up" to fit the window, meaning that each source pixel would be 2x2. This would be how you'd do a low-res old-school look on a modern high-def window.

You can play with the other settings to see how they affect things. The full list of window options is in the *window:* section of the config file reference. (Just be sure that you add them to the `window:` section of your config, not the "window" entry in the `displays:` section.) Check that out to see what else you can do.

Note: At this time, the MPF Media Controller only supports a single LCD window at a time. If you want more than one LCD window, MPF 0.31 will let you run multiple instances of the MPF-MC at the same time—one for each window.

Now you have a working config, so you can read through the rest of the display documentation to see how you can add slides and widgets to your display.

Also, if you want to make the content on your window look like dots, or if you want to show a "virtual" DMD in your window, check out the other guides in this section.

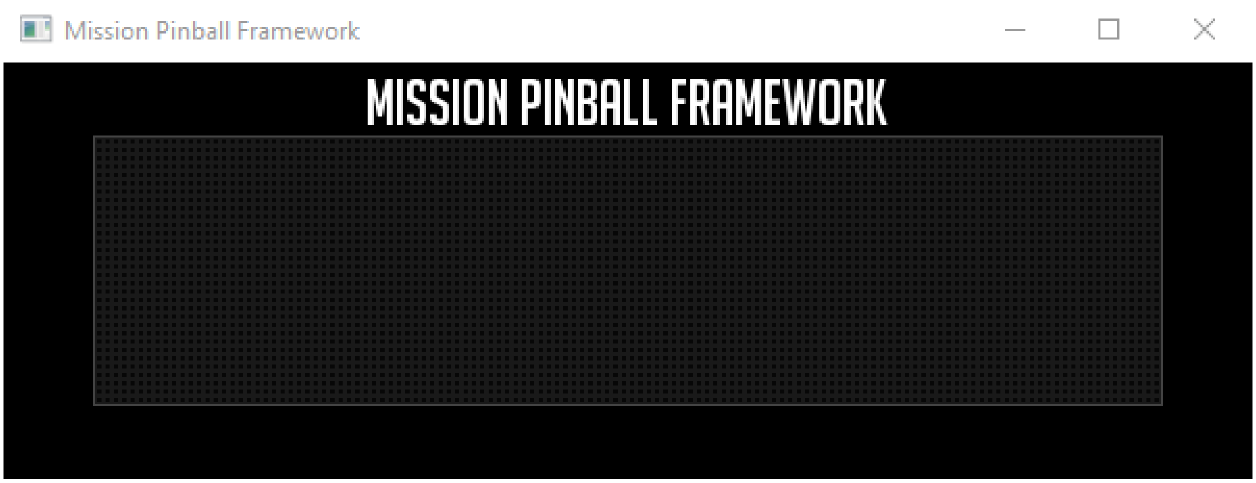
See *Using multiple screens* for informations about using two or more LCDs.

Using a traditional (single color) physical DMD

This guide will show you how to use a traditional, physical DMD with MPF, like this:

This is supported for all *Monochome DMD platforms*.

It will also show you how to create an on-screen popup window which will show the contents of the DMD, like this (with a blank DMD):



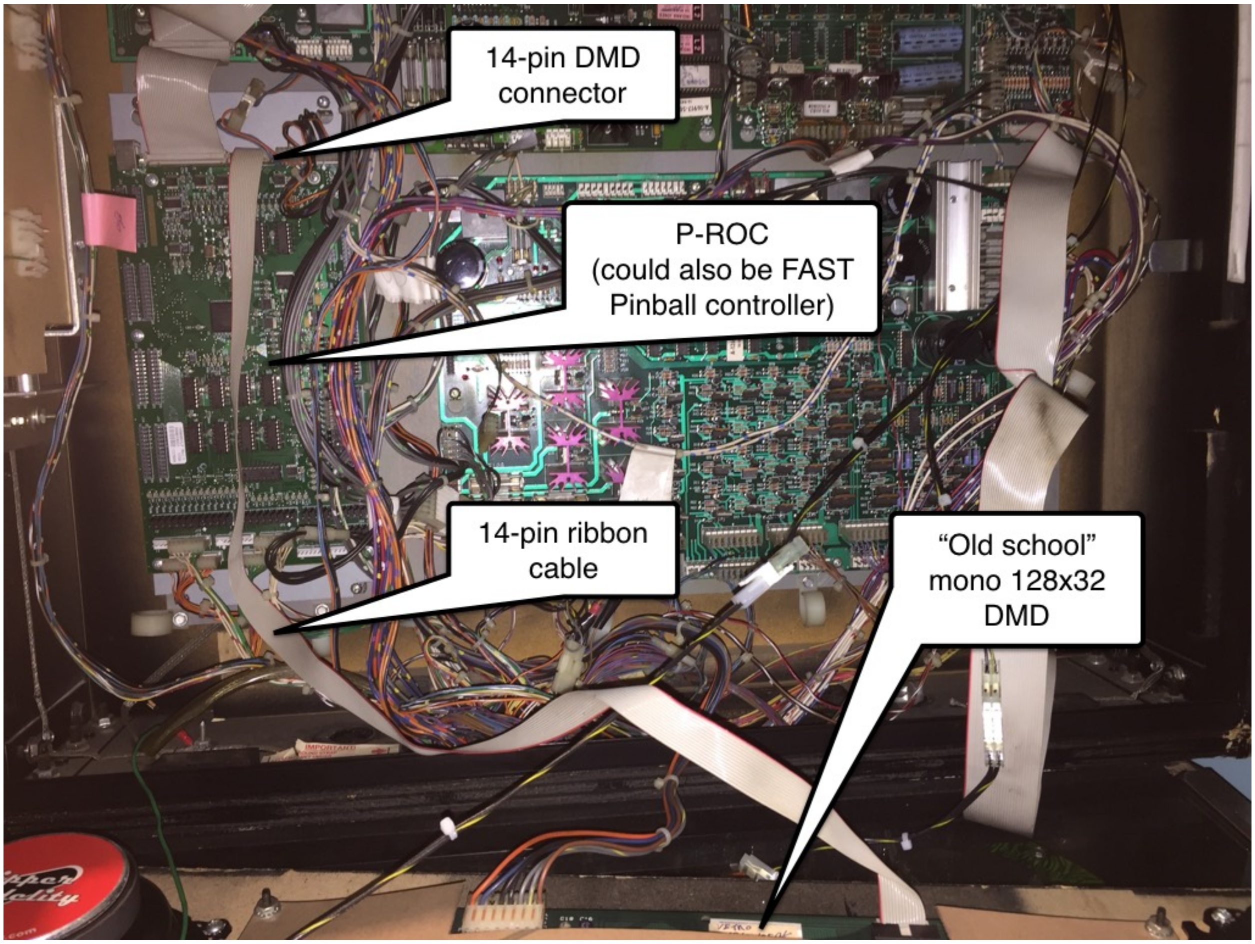
If you want to use a physical DMD without the on-screen equivalent, we'll show you how to do that at the end.

If you want to only have an on-screen DMD without the physical one, like if you want to replace the DMD with an LCD screen but still have it look like a DMD, then read *this guide* instead.

The final version of the relevant sections of your machine config for a physical DMD with an on screen window too will look like this:

1. Understand how physical, mono DMDs work

This guide explains how to config physical single-color (mono) DMDs. These are DMDs that are connected to your FAST Pinball or P-ROC controller via the 14-pin ribbon cable, like this:



It makes no difference whether you’re using an LED or an original plasma gas DMD. (Also it doesn’t matter what color it is.)

2. Add your displays to your MPF config

The first part of the config file above is where you create your logical displays like we covered in the *Working with Displays* section.

```
displays:
  window:
    width: 600
    height: 200
  dmd:
    width: 128
    height: 32
    default: true
```

We’re creating two displays here. The first is called “window” and has a size of 600x200. This will be the display that shows up on the computer screen.

The second display, which we’re calling “dmd”, will be the display that provides the content for the physical DMD. This display is 128x32, which is the pixel size of the DMD.

Notice that we set `default: true` for the dmd display. This is because as we’re creating display content in our game, we want it (by default) to show up in the DMD (since that will be the primary display in our game).

Note that you don't set colors or anything here—this is just setting up the logical displays which we'll use next.

3. Add your window configuration

Next, we have a `window:` section which holds the settings for the actual on screen window itself. In this case we're just configuring it to be 800x600, with a window title of "Mission Pinball Framework".

```
window:
  width: 600
  height: 200
  title: Mission Pinball Framework
```

Check out *Step 2. of the LCD guide* for more details on this window section, and be sure to check out all the window options in the *window:* section of the config file reference.

Notice that in this case, we did not add the `source_display: window` setting to this section. That's because we have a logical display called "window", and when you have that, the on-screen window will automatically use that display as its source.

4. Configure a window slide to show the on screen DMD

Now we have a working on-screen window and a working physical RGB DMD. But if you run `mpf both` now, your on screen window will be blank because we haven't built any slides to show up.

So in this step, we're going to build a slide for the on-screen window that will be shown when MPF starts. We'll add some widgets to that slide to make it look like the screen shot at the beginning of this guide.

First, create a `slides:` section in your machine config (if you don't have one already), and then create an entry for the slide that we want to show. In this case, we've decided to name that slide "window_slide_1". (Of course you can call this slide whatever you want.)

```
slides:
  window_slide_1:
```

Next we have to add some widgets to that slide. (Refer to the *documentation on widgets* if you're not familiar with widgets yet.)

The first widget will be a *display widget* with a *dmd effect* which is a widget which renders a logical display onto a slide in a way that makes it look like a DMD:

```
slides:
  window_slide_1:
    - type: display
      effects:
        - type: dmd
          dot_color: ff5500
      width: 512
      height: 128
```

Again, there are lots of options here. Note that we're adding a `height:` and `width:` of 512x128. This is the on-screen pixel size of the DMD as it will be drawn in the window. In this case we chose an even multiple of the source display for the DMD (which is 128x32), meaning that each pixel of the original DMD will be rendered on screen as 4 pixels by 4 pixels. This is big enough to get the circular "dot look" filter to look good, and being an even multiple means that we won't have any weird moire patterns.

For the on screen DMD, we *are* able to select the pixel color, because this is how the DMD will be drawn on the computer screen, and MPF has no idea what color the actual DMD is. So you can pick any color you want here. We chose `ff5500` which is a classic DMD orange color.

There are other options listed in the *dmd effect* documentation to control settings like how big the circles are versus the space in between them, the ability to not have the "dot" filter, and the ability to set the "glow" radius of each dot, color tint, limiting the color palette, etc.

Note that in this case, we did not have to add the `source_display:` option because we have a display called "dmd" which will automatically be used as the source for the color DMD widget.

Next, we also added two more widgets to this slide—a text widget with the title of the machine, and a gray rectangle that's slightly larger than the DMD to give it a nice border.

```
slides:
  window_slide_1:
    - type: display
      effects:
        - type: dmd
          dot_color: ff5500
      width: 512
      height: 128
    - type: text
      text: MISSION PINBALL FRAMEWORK
      anchor_y: top
      y: top-3
      font_size: 30
    - type: rectangle
      width: 514
      height: 130
```

5. Configure the slide to show when MPF starts

Now we have a nice slide with the virtual DMD on it, but if you run MPF, you still won't see it because we didn't tell MPF to show that slide in the window. So that's what we're doing here:

```
slide_player:
  init_done:
    window_slide_1:
      target: window
```

If you don't have a slide_player: entry in your machine-wide config, go ahead and add it now. Then create an entry for the *init_done* event. This is the event that the media controller posts when it's ready to be used, so it's a good event for our use case.

Then under that event, create an entry to show the slide you just created in the previous step. Notice that we also have to add the `target: window` entry to tell the slide player that we want this slide to show on the "window" target. We need to do this because the default display (from Step 2) is the

DMD, so if we don't specify a target, this slide will show on the default, which would be the DMD, instead of being shown on the window. (In this case, we would show a slide on the DMD which contains a DMD widget whose source is the DMD, and we'd probably open up some kind of wormhole and destroy the universe. So don't do that.)

And this point, you're all set! Of course there's no content on the DMD yet because we haven't set up any slide_player entries to add content to it, but that's something you can do by following the tutorial or looking at the guides for the slides and widgets here.

6. What if you don't want the on-screen window?

There might be some scenarios where you just want the physical DMD with no on-screen DMD. (For example, maybe you're using a low-power single board computer and you don't have enough horsepower to run a graphical environment.)

This is fine. To do it, just remove the window-related components from the config.

In this case, you wouldn't need the `default: true` entry for the dmd in the `displays:` section because you only have one display, so it will automatically be the default.

7. Configure the physical DMD

At this point you have two displays configured, and you have default content showing up in both of them. The final step is to add the configuration for your physical DMD so that MPF can talk to your hardware.

The exact steps to do that vary depending on which DMD hardware platform you've chosen, so click on the one you have from the list below and follow the final instructions there to get everything set up.

- *FAST Pinball Core & WPC controllers*
- *P-ROC*

Using an RGB full-color LED DMD

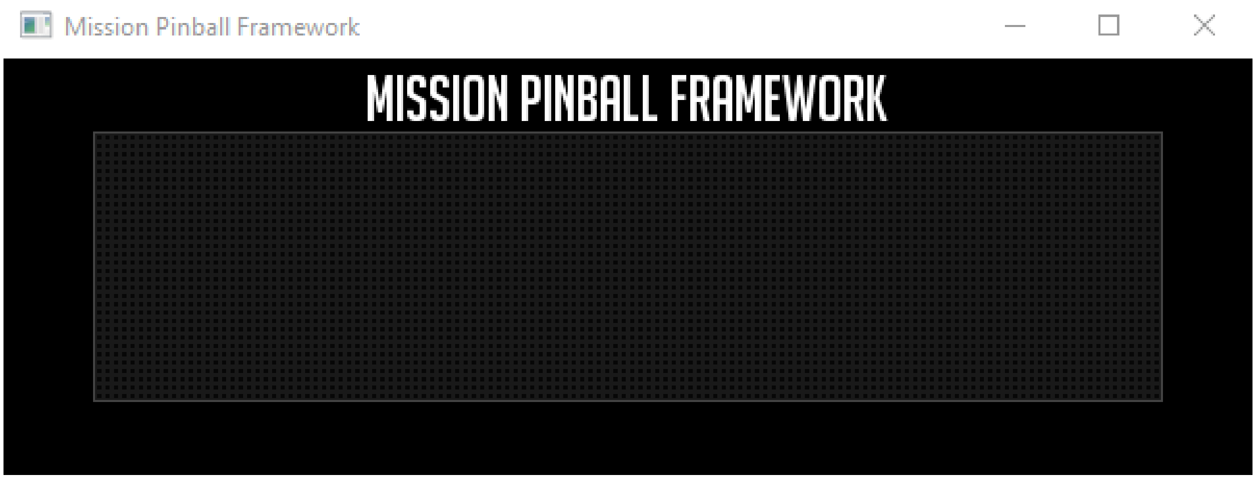
Related Config File Sections
displays:

MPF supports RGB full-color LED DMDs. There are several hardware options you can use for this: *DMD Platforms in MPF*.

This guide shows you how to configure MPF to use one of these displays.

By the way, these RGB LED DMDs have been called "real" Color DMDs in the forums since the displays are arrays of RGB LEDs rather than an LCD monitor running a display that is made to look like a DMD. Many people like these better than LCD-based displays because they're brighter and more vibrant, and the blacks are actually black since the LEDs are off versus LCD displays which have blacks are are actually dark gray.

We will also show you how to create an on-screen popup window which will show the contents of the DMD, like this (with a blank DMD):



If you want to use a physical RGB DMD without the on-screen equivalent, we'll show you how to do that at the end of this guide.

If you want to only have an on-screen DMD without the physical one, like if you want to replace the DMD with an LCD screen but still have it look like a color DMD, then read *this guide* instead.

The final version of the relevant sections of your machine config for a physical RGB DMD with an on screen window too will look like this:

1. Add your displays to your MPF config

Next, add the DMD display to your list of displays in your machine-wide config file:

```
displays:
  window:
    width: 600
    height: 200
  dmd:
    width: 128
    height: 32
    default: true
```

The example above contains two displays. The first is named "window" and has a size of 600x200. This will be the display that shows up on the computer screen. (Again, if you just want the DMD without an on-screen window, we'll show you how to do that later, but for now it's probably easiest to create a screen window so you can see what's happening with the display if you're working on your game without a physical machine attached.)

The second display, which we're calling "dmd", will be the display that provides the content for the physical RGB DMD. This display is 128x32, which is the pixel size of the DMD. If you have a different size DMD, enter the size (in pixels) here.

Notice that we set `default: true` for the DMD display. This is because as we're creating display content in our game, we want it (by default) to show up in the DMD (since that will be the primary display in our game).

2. Add your window configuration

The `window:` section of the machine-wide config holds the settings for the on-screen display window. If you don't have this section, add it now.

You can make the width and height anything you want. In this case we're just configuring it to be 600x200 with a window title of "Mission Pinball Framework".

```
window:
  width: 600
  height: 200
  title: Mission Pinball Framework
```

Check out *Step 2. of the LCD guide* for more details on this window section, and be sure to check out all the window options in the *window:* section of the config file reference.

Notice that in this case, we did not add the `source_display: window` setting to this section. That's because we have a logical display called "window", and when you have that, the on-screen window will automatically use that display as its source.

3. Configure a window slide to show the on screen DMD

Now we have a working on-screen window and a working physical RGB DMD. But if you run `mpf both` now, your on screen window will be blank because we haven't built any slides to show up.

So in this step, we're going to build a slide for the on-screen window that will be shown when MPF starts. We'll add some widgets to that slide to make it look like the screen shot at the beginning of this guide.

First, create a `slides:` section in your machine config (if you don't have one already), and then create an entry for the slide that we want to show. In this case, we've decided to name that slide "window_slide_1". (Of course you can call this slide whatever you want.)

```
slides:
  window_slide_1:
```

Next we have to add some widgets to that slide. (Refer to the *documentation on widgets* if you're not familiar with widgets yet.)

The first widget will be a *display widget* with a *color_dmd effect* which is a widget which renders a logical display onto a slide in a way that makes it look like a DMD:

```
slides:
  window_slide_1:
    - type: display
      effects:
        - type: color_dmd
      width: 512
      height: 128
```

Again, there are lots of options here. Note that we're adding a `height:` and `width:` of 512x128. This is the on-screen pixel size of the DMD as it will be drawn in the window. In this case we chose an even multiple of the source display for the DMD (which is 128x32), meaning that each pixel of the original DMD will be rendered on screen as 4 pixels by 4 pixels. This is big enough to get the circular "dot look" filter to look good, and being an even multiple means that we won't have any weird moire patterns.

There are other options listed in the *color_dmd effect* documentation to control settings like how big the circles are versus the space in between them, the ability to not have the "dot" filter, and the ability to set the "glow" radius of each dot, color tint, limiting the color palette, etc.

Note that in this case, we did not have to add the `source_display:` option because we have a display called "dmd" which will automatically be used as the source for the color DMD widget.

Next, we also added two more widgets to this slide—a text widget with the title of the machine, and a gray rectangle that's slightly larger than the DMD to give it a nice border.

```
slides:
  window_slide_1:
    - type: display
      effects:
        - type: color_dmd
      width: 512
      height: 128
    - type: text
      text: MISSION PINBALL FRAMEWORK
      anchor_y: top
      y: top-3
      font_size: 30
      color: white
    - type: rectangle
      width: 514
      height: 130
      color: 444444
```

4. Configure the slide to show when MPF starts

Now we have a nice slide with the virtual DMD on it, but if you run MPF, you still won't see it because we didn't tell MPF to show that slide in the window. So that's what we're doing here:

```
slide_player:
  init_done:
    window_slide_1:
      target: window
```

If you don't have a slide_player: entry in your machine-wide config, go ahead and add it now. Then create an entry for the *init_done* event. This is the event that the media controller posts when it's ready to be used, so it's a good event for our use case.

Then under that event, create an entry to show the slide you just created in the previous step. Notice that we also have to add the `target: window` entry to tell the slide player that we want this slide to show on the "window" target. We need to do this because the default display (from Step 2) is the DMD, so if we don't specify a target, this slide will show on the default, which would be the DMD, instead of being shown on the window. (In this case, we would show a slide on the DMD which contains a DMD widget whose source is the DMD, and we'd probably open up some kind of wormhole and destroy the universe. So don't do that.)

And this point, you're all set! Of course there's no content on the DMD yet because we haven't set up any slide_player entries to add content to it, but that's something you can do by following the tutorial or looking at the guides for the slides and widgets here.

5. What if you don't want the on-screen window?

There might be some scenarios where you just want the physical DMD with no on-screen DMD. (For example, maybe you're using a low-power single board computer and you don't have enough horsepower to run a graphical environment.)

This is fine. To do it, just remove the window-related components from the config.

In this case, you wouldn't need the `default: true` entry for the dmd in the `displays:` section because you only have one display, so it will automatically be the default.

6. Configure your RGB DMD Hardware

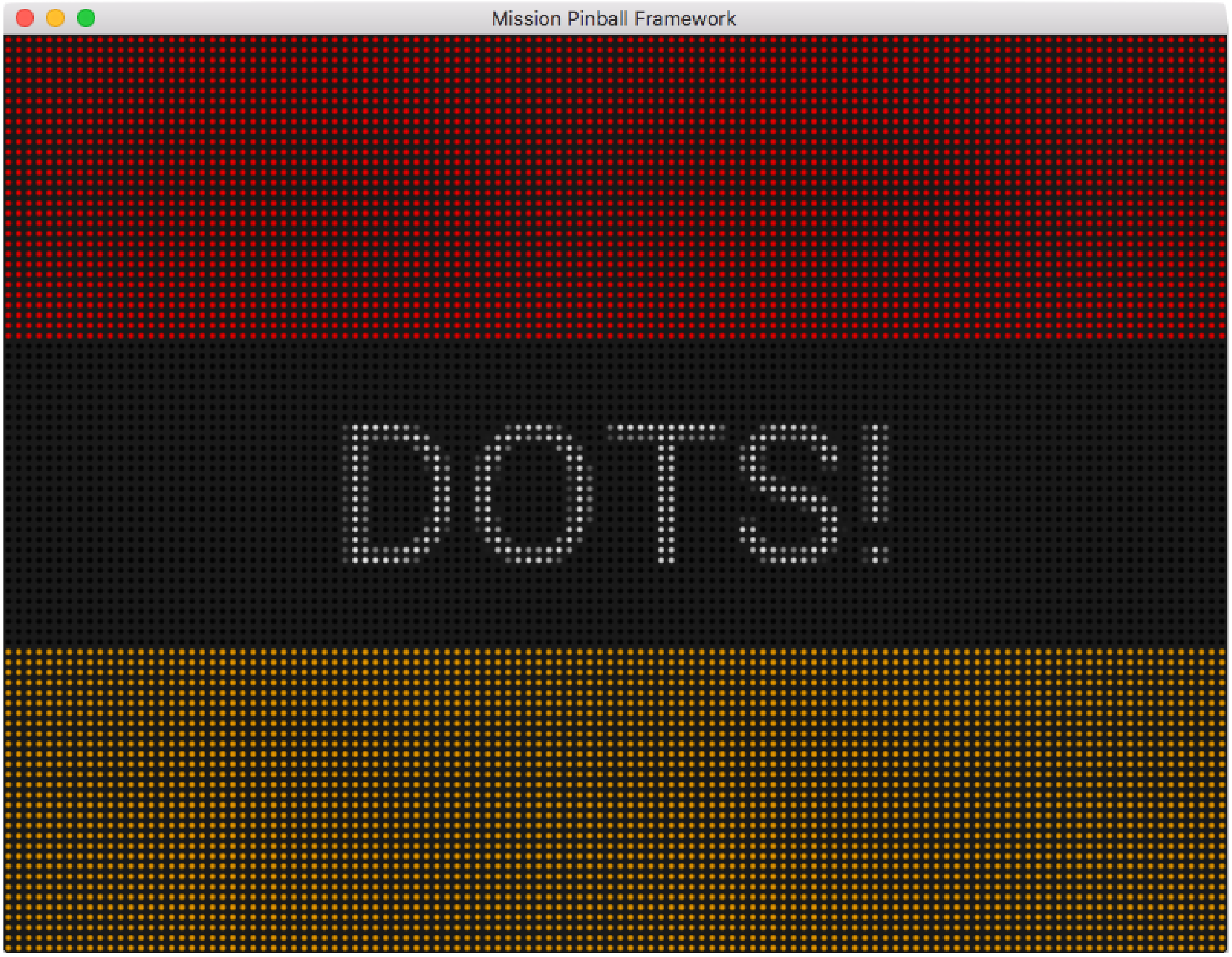
At this point you have two displays configured, and you have default content showing up in both of them. The final step is to add the configuration for your physical RGB DMD so that MPF can talk to your hardware.

The exact steps to do that vary depending on which DMD hardware platform you've chosen, so click on the one you have from the list below and follow the final instructions there to get everything set up.

- *SmartMatrix*
- *RGB.DMD*
- *FAST Pinball RGB DMD*
- *Raspberry Pi DMD*
- *PIN2DMD RGB DMD*

How to give your on-screen window the DMD "dot look"

This guide will show you how to configure a full screen "dot look" display, like this:



The final sections of the machine config to make this happen are here:

```
displays:
  window:
    width: 800
    height: 600
  dmd:
    width: 120
    height: 90
    default: true
slides:
  window_slide:
    - type: display
      effects:
        - type: color_dmd
          dot_size: .5
      width: 800
      height: 600
  dmd_slide:
    - type: text
      text: DOTS!
    - type: rectangle
      width: 120
```

(continues on next page)

(continued from previous page)

```
      height: 30
      color: orange
      y: 0
      anchor_y: bottom
    - type: rectangle
      width: 120
      height: 30
      color: red
      y: top
      anchor_y: top
slide_player:
  init_done:
    window_slide:
      target: window
    dmd_slide:
      target: dmd
```

Let's step through this step-by-step.

1. Create your displays

To understand how this works, you have to understand the concepts of MPF *displays* and *widgets*.

What's actually happening under the hood is that you set up two MPF displays. The first is the "window", which is the display that represents your on-screen window. This should be set to the size of the screen window at the native resolution of the monitor or LCD where it's being shown.

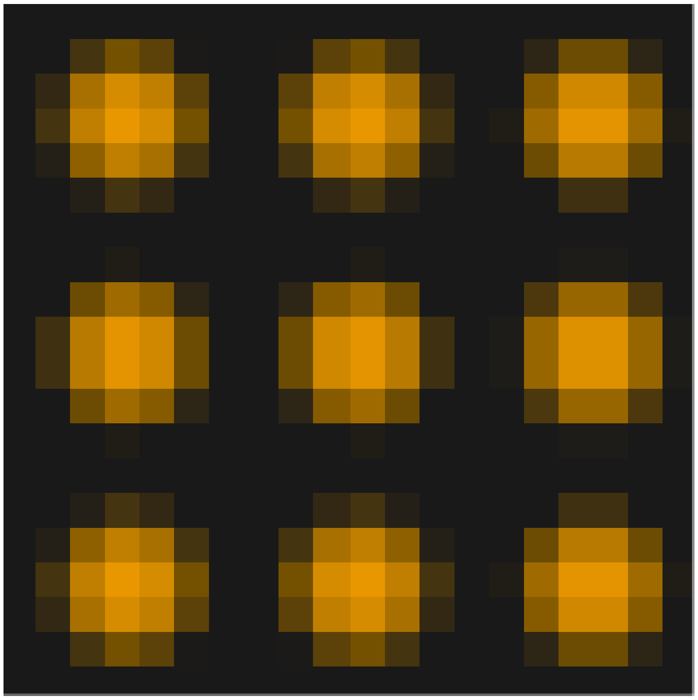
```
displays:
  window:
    width: 800
    height: 600
  dmd:
    width: 120
    height: 90
    default: true
```

In the example above, this is 800x600, but on your actual machine, it will probably be something like 1024x768, 1280x1024, 1600x1200, etc.

The second MPF display represents the virtual DMD itself, and you set that to the number of pixels (or dots) you want to be drawn in your window. In the example above, this is set to 120x90, meaning the virtual DMD is 120 dots wide and 90 dots tall. You can make this anything you want.

The key to remember is that the parent window will be using its pixels to draw the individual dots that make up the virtual DMD. So a smaller DMD resolution means the window has more pixels to use per-dot, resulting in a better overall image.

For example, if we zoom in on the 120x90 virtual DMD being shown on an 800x600 window, we'll see that it looks like this:



This works because there is about a 6x6 grid of pixels in the window for each virtual pixel in the DMD.

But if you increased the virtual DMD to 400x300 (instead of 120x90), that would mean you only had a 2x2 window area to render each pixel, and it wouldn't really work because you can't draw a circle with space around it in a 2x2 pixel.

Also note that we added `default: yes` to the dmd display, since as we get deeper into the machine config, we want all the content (the slide_player, etc.) to show up in the DMD display.

2. Create your window slide

Once you have your displays configured, the next step is to create the slide that will be shown in the window. In this case, the slide will only have a single widget, and that widget will be the Color DMD widget which will be used render the virtual DMD into the window.

```
slides:
  window_slide:
    - type: display
      effects:
        - type: color_dmd
          dot_size: .5
      width: 800
      height: 600
```

We decided to name this slide "window_slide", though you can name it whatever you want.

Note that in this case, we set the width and height of the color_dmd widget so that it's the same size as the window itself. This is what causes it to be scaled to the full size of the window.

We do *not* set the number of dots in the DMD here, as that's automatically pulled in from the dmd display setting.

We also do not need to set a source display for the color_dmd widget since it will automatically use a display called "dmd".

3. Create your DMD slide

Next, we need a slide to show in the DMD itself. This is just something we're setting up here as an example "first slide". In your actual game, this slide will be ever changing and will reflect what's happening in your machine.

We're calling our first slide "dmd_slide":

```
slides:
  dmd_slide:
    - type: text
      text: DOTS!
    - type: rectangle
      width: 120
      height: 30
      color: orange
      y: 0
      anchor_y: bottom
    - type: rectangle
      width: 120
      height: 30
      color: red
      y: top
      anchor_y: top
```

There's nothing special about this slide. We just added a text widget and two colored rectangles.

4. Configure your slides to show up

Finally, we need to create a slide_player entry which will cause the two slides we just created to be shown. In this example, we're using the *init_done event* since that's the event that's posted by the media controller once it's been initialized and ready to go.

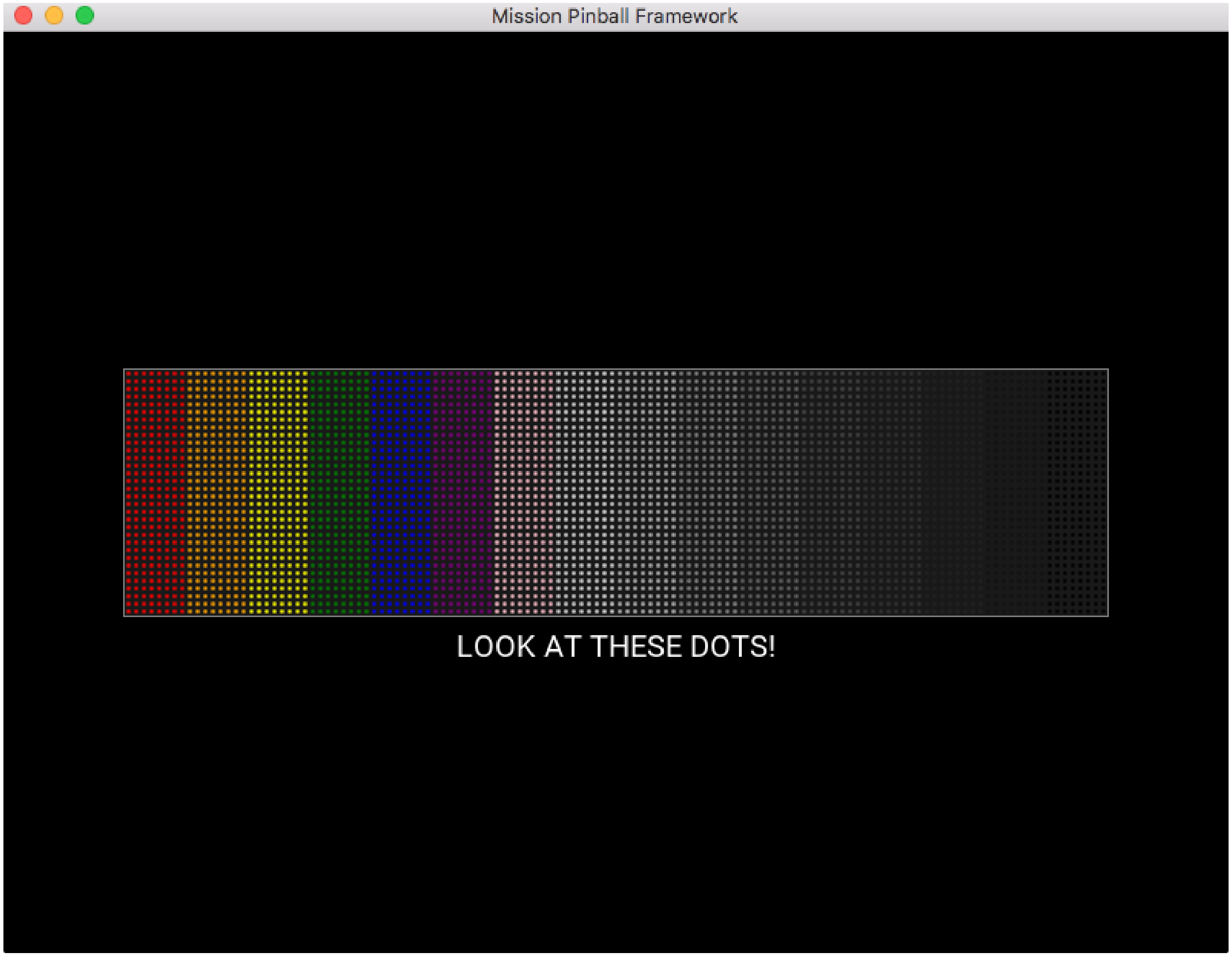
```
slide_player:
  init_done:
    window_slide:
      target: window
    dmd_slide:
      target: dmd
```

Since the DMD display is configured to be the default, when you use the slide_player in the rest of your game, you won't have to specify `target: dmd`. We just included it here to make it clear that we were targeting the window slide to the window display and the dmd slide to the dmd display.

5. Other options & positioning your DMD

Finally, remember to check the documentation for the *display widget* and the *color_dmd effect* for a full list of the options you can use to fine-tune how the DMD looks in the window. For example, you can configure the pixel size, the glow radius, the color of the space between the pixels, gain, tint, etc.

Also, you don't have to make the virtual DMD be the full size of the display. For example, if you set your dmd display to be 128x32 and then set the color_dmd widget to be 640x160, you'll get a display like this:



You can also use the *widget sizing and positioning* to create a DMD widget that is pre-positioned at a certain spot on the display. This is useful if you have a standard size LCD monitor in your backbox but only part of it is visible to the player. In that case you could make a color_dmd widget that was the size of the viewable area and use the widget positioning settings to align it to the area of the display that was visible.

You can also use the various *window:* options (such as full screen) to properly align the content of the display with the visible area.

Finally, even though this example was using the color_dmd widget, you could replace it with the *display widget* with a *dmd effect* for a single color look instead of full color.

Alpha-Numeric / Segment Displays

Related Config File Sections
segment_displays:
segment_display_player:

MPF supports segment displays and alpha numeric displays. There are several hardware options available: *Segment Display Platforms in MPF*.

1. Configure your segment displays in MPF config

You can use the following tested config snippet as a starting point to implement segment displays (make sure to use the correct numbers for your hardware).

Listing 1: your_machine_folder/config/display.yaml

```
#config_version=5

segment_displays:
  display1:
    number: 1
  display2:
    number: 2
  display3:
    number: 3
  display4:
    number: 4
  display5:
    number: 5

segment_display_player:
  # empty all displays on game start and setup display5
  game_start:
    display1:
      text: ""
    display2:
      text: ""
    display3:
      text: ""
    display4:
      text: ""
    display5:
      text: "{current_player.ball:d}"

  # clear only display5 after game
  game_ended{machine.player1_score > 0}:
    display1:
      text: "{machine.player1_score:d}"
  game_ended{machine.player2_score > 0}:
    display2:
      text: "{machine.player2_score:d}"
  game_ended{machine.player3_score > 0}:
    display3:
      text: "{machine.player3_score:d}"
  game_ended{machine.player4_score > 0}:
    display4:
      text: "{machine.player4_score:d}"
  game_ended:
    display5:
      text: ""
```

(continues on next page)

(continued from previous page)

```
# flash display on player turn
player_turn_started.1{number==1}:
  display1:
    action: flash
player_turn_ended.2{number==1}:
  display1:
    action: no_flash
player_turn_started.3{number==2}:
  display2:
    action: flash
player_turn_ended.4{number==2}:
  display2:
    action: no_flash
player_turn_started.5{number==3}:
  display3:
    action: flash
player_turn_ended.6{number==3}:
  display3:
    action: no_flash
player_turn_started.7{number==4}:
  display4:
    action: flash
player_turn_ended.8{number==4}:
  display4:
    action: no_flash

# show score when adding players
player_added.1{num==1}:
  display1:
    text: "{players[0].score:d}"
player_added.2{num==2}:
  display2:
    text: "{players[1].score:d}"
player_added.3{num==3}:
  display3:
    text: "{players[2].score:d}"
player_added.4{num==4}:
  display4:
    text: "{players[3].score:d}"
```

2. Implement virtual segment displays

If you don't have or want phyiscal segment displays you can also emulate them using the following slides:

```
slides:
  segment_displays:
    widgets:
      - type: text
        text: (player1|score)
        number_grouping: true
```

(continues on next page)

(continued from previous page)

```
        min_digits: 2
        font_name: ten_segment
        color: blue
        x: 620
        y: 724
        font_size: 240
        anchor_x: right
        anchor_y: bottom
        z: 2
# show slide on game start
slide_player:
  game_started: segment_displays
```

Using multiple screens

This section covers using multiple displays and screens.

Distinction between displays, windows and screens

The MPF media controller knows windows and displays. A `window` is the window where MPF-MC pops up on your desktop using kivy. Internally, MPF can have multiple `displays` which are internal viewports and can be targeted by slides. A `display` can either be displayed on a `window` or on one or more DMDs. Additionally, a `display` can also show one or more other `displays` using *display widgets*. MPF does not know about screens which are phyiscal monitors connected to your machine. However, the kivy window can span multiple screens.

Using multiple screens on your PC

Some machines use more than one screen. Unfortunately, kivy (the graphics framework we use) does only support one screen and cannot easily be started two times. This is mostly caused by OpenGL which is rarely used to render multiple windows. The simplest solution to this problem is to extend the MC window to span both (or more) screens. This can be achieved using the following config:

```
kivy_config:
  kivy:
    desktop: 1
    exit_on_escape: true
    pause_on_minimize: 0
    log_dir:
    window_icon:
  graphics:
    borderless: true
    window_state: visible # visible, hidden, maximized, minimized
    fbo: hardware # hardware, software, force-hardware
    fullscreen: false
    left: 0
    top: 0
    width: 3840   # width of display1 + display2
    height: 1080  # common height (or the maximum of both)
```

(continues on next page)

(continued from previous page)

```
    maxfps: 30
    multisamples: 2
    position: custom # auto, custom
    show_cursor: true
    resizable: false
    rotation: 0
displays:
  display1:
    width: 1920
    height: 1080
  display2:
    width: 1920
    height: 1080
  combined_display:
    width: 3840
    height: 1080
slides:
  base_slide:
    - type: display
      source_display: display1
      width: 1920
      height: 1080
      x: left
      anchor_x: left
    - type: display
      source_display: display2
      width: 1920
      height: 1080
      x: right
      anchor_x: right
slide_player:
  mc_ready:
    base_slide:
      target: combined_display
```

Use `width` and `height` to set the size of the window. `left` and `top` are used to position the window.

We created one window which spans both screens. Then we define a display `combined_display` which will be displayed on startup by the slide `base_slide` spanning both screens. `base_slide` contains two widgets which show the displays `display1` and `display2`. You can now target any slides to those two displays. See *Display Widget* for details.

Using multiple displays

You can easily use two DMDs and one LCD (or two LCDs with the solution above). To implement that you need to define multiple displays. One diplay per DMD and one for your LCD. If you want to show your DMDs on the LCD (i.e. during development) you can also define a fourth display and create a slide which contains three *display widgets* to show the other three displays.

A note on performance with "displays" and "dmds"

If you have a physical DMD defined (in the *dmds:* or *rgb_dmds:* of your machine config) and are emulating the DMD's slides and widgets in your window, be aware that the MPF media controller will process the graphics data for the physical DMD *even when MPF is running in "virtual" mode*.

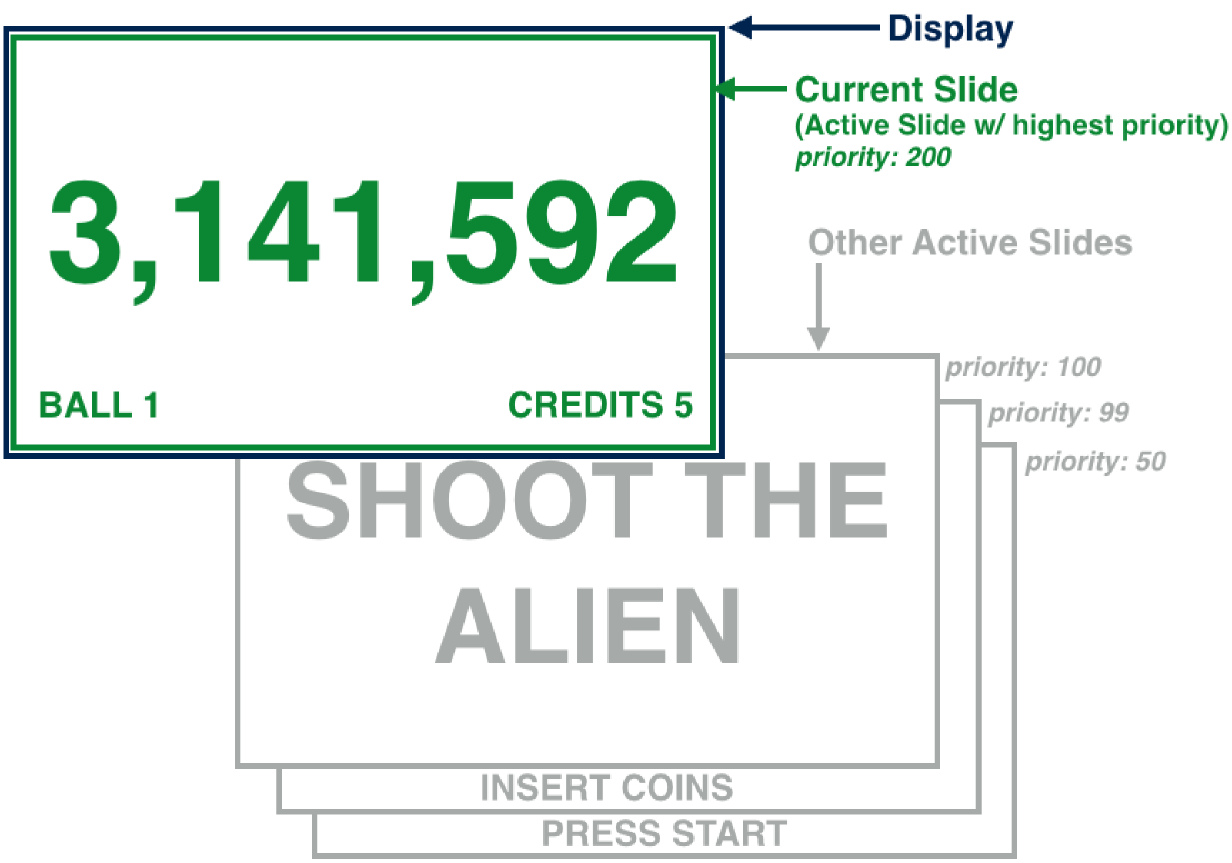
Although that graphics data will not be sent to a physical DMD, processing it provides a more realistic MPF experience because of the considerable CPU power required to convert on-screen graphics to DMD data.

If you are planning to use a physical DMD at some point on your project, it's recommended to configure one *before* you start designing your slides and widgets. Especially if you will be running virtually for the bulk of your early game design: you don't want to spend time designing intricate slides and high-resolution graphics only to find your CPU crumble when you finally attach a physical DMD.

Slides

Now that you know what a display is, the next concept you need to understand is "slides". Slides in MPF are just like slides in a PowerPoint presentation or slides in an old-fashioned slide projector.

You create multiple slides (each with its own content), and then you tell MPF when to activate certain slides. Every slide has a priority, so if multiple slides are active at the same time, the one with the highest priority will be shown. You can also set "transitions" which control what visual effect is used to transition from the current slide to the new slide. (Transitions are things like cross-fade, move in, push out, etc.)



Slide Priorities

Every slide in MPF has a priority, which is simply a numeric value. Bigger numbers equal higher priority.

Since only one slide is shown at a time, whenever there is more than one active slide, whichever slide has the highest priority will be the one that's shown.

For example, you might have a general score slide at priority 100 which shows the current player's score, the ball, the credits, and maybe the scores of the other players.

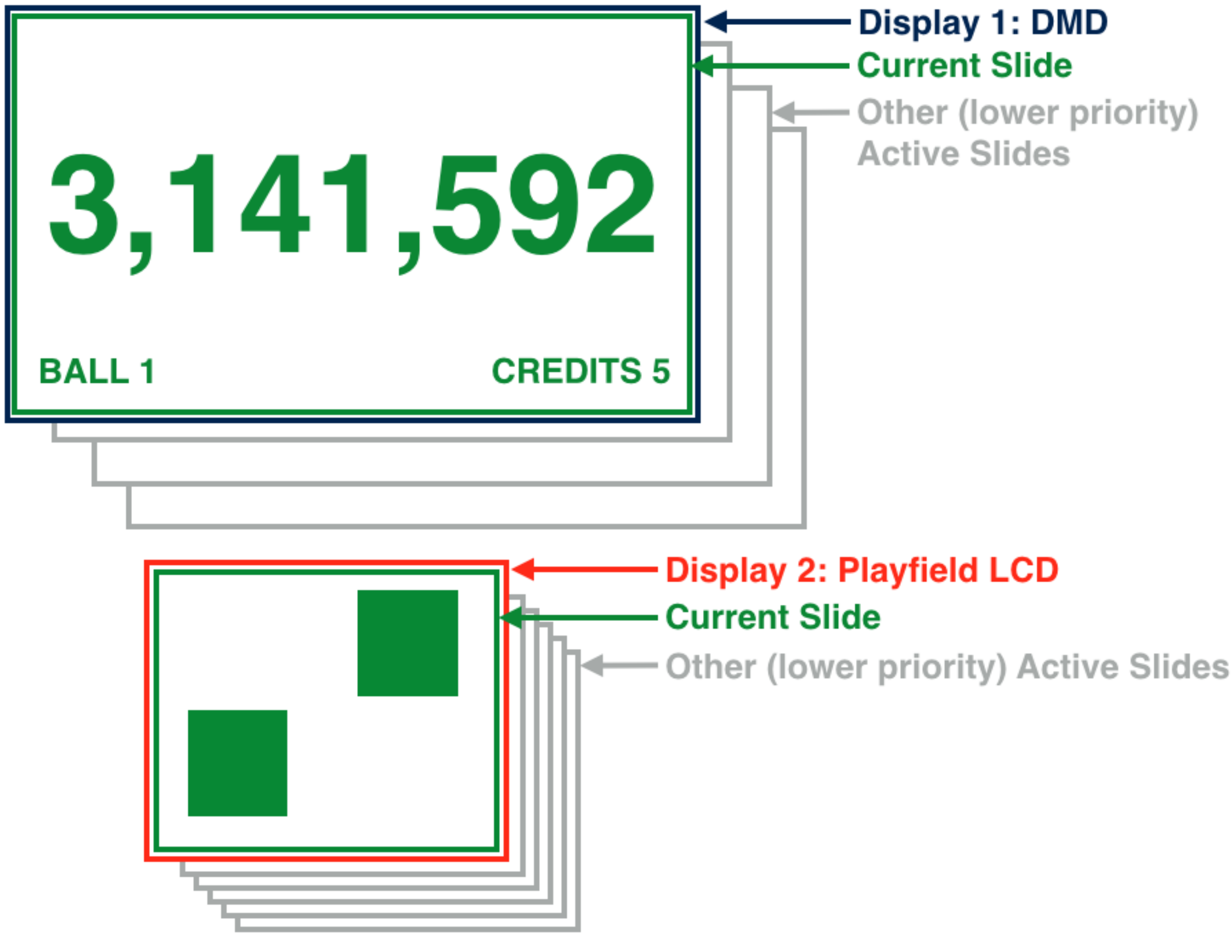
If the player shakes the machine too hard and a tilt warning slide is shown, then that tilt warning slide might be activated at a priority of 10,000, meaning that it would be shown instead of the general score slide.

Then after a few seconds, the tilt warning slide might be removed, and MPF will then show the next-highest active slide which would most likely be the general scoring slide that was showing before.

The slide priority system is integrated into MPF's mode system, meaning that slides created by modes automatically inherit the priority of the mode that's showing them. Put another way, a slide from a higher priority mode would show in place of a slide from a lower priority mode (though every mode doesn't need to have slides). You can also tweak the priorities of slides (higher or lower) to make sure the slide you want to show is the one that's showing at any given time. We'll dig into that later in the documentation.

Slides with Multiple Displays

When MPF is used with *multiple displays*, each display maintains its own stack of active slides. The priorities of the slides in the stack and the priority of the current slide on one display has nothing to do with the active and current slides of another display.



How to create slides

Since slides are so critical in MPF's display system, let's look at how you actually create slides. You can test slides and widgets interactively using *Interactive MC (iMC)*.

There are several ways you can define and create slides:

- In a *slides:* section of a config file.
- Dynamically in the *slide_player:* section of your config.
- Dynamically in a show config or show file.

Let's look at each of these options.

Defining slides in the slides: section of a config file

The main way to do it is in the "slides" section of a config file, like this:

```
slides:
  some_slide:
    - type: text
```

(continues on next page)

(continued from previous page)

```
    text: THIS IS MY SLIDE
some_other_slide:
  - type: text
    text: THIS IS ANOTHER SLIDE
  - type: text
    text: WITH MORE WORDS
    y: bottom
    anchor_y: bottom
tilt_warning_1:
  - type: text
    text: WARNING
tilt_warning_2:
  - type: text
    text: WARNING WARNING
```

In the example above, we have four main sub-entries in the slides section:

- some_slide
- some_other_slide
- tilt_warning_1
- tilt_warning_2

Each of the above listed subsections represents a different slide, and the names of those sections are used as the names of those slides. In other words, this config has a slide called "some_slide", another slide called "some_other_slide", etc.

You can list slides in a *slides:* section of either your machine-wide or a mode config. The most important thing to know about slide names is that they are GLOBAL throughout MPF. That means that MPF has a single master list of all the slide names used in the entire game. (So don't use the same slide name twice or it will get confused.)

The configuration entries under each slide name are the widgets that will be added to that slide. (Each slide can have one or more widgets. You can read about all the different types of widgets, as well as the options for widget positioning and sizing, in the *widgets* section of the documentation.

You'll probably end up creating hundreds of slides in your machine by the time you're done with it.

Note: The slides defined in the *slides:* section are just the configurations that are used to create the slides when they're needed. In other words, no memory is used to "hold" the slides, so you can create lots and lots of them without worrying about running out of memory.

At this point, you're just creating the slides. Deciding when to show which slide will come later.

Since MPF maintains a single global list of slides, it doesn't technically matter whether you define your slides in the *slides:* section of your machine-wide config or your mode config. Obviously though if you define the slides a mode will use in that mode's config file, then that will help you keep everything more organized.

Dynamically defining slides in a slide_player: section of a config file

The *slide_player:* section of a machine-wide or mode config is where you tell MPF to show (or "play") a specific slide when some event occurs. Full documentation for the slide_player is in the *How to Show a Slide on a Display* section of the documentation.

You can define slides in the slide_player like this:

```
slide_player:
  some_event:
    my_slide_1:
      - type: text
        text: THIS IS MY SLIDE
```

In the above example, when the event *some_event* is posted, the slide player will respond and show the slide called *my_slide_1* which will include that single text widget.

It doesn't really matter whether you pre-define a slide in the *slides:* section of a config versions dynamically defining it in the *slide_player:* section. Really it comes down to personal preference. Some people like to have all their slides in one location (all in the *slides:* section), whereas others prefer to have the configuration for the slides closer to where they will be used (by defining them in the *slide_player:* section). Most people end up mixing-and-matching, with some quick-and-dirty one-time use slides in the slide_player with other slides you might reuse in the slides: section.

Dynamically defining slides in a show config

As you'll learn in other parts of this documentation, anything that's in one of the "_player" sections of the config (like the "slide_player" above), can also be defined in a show configuration (from a show file or a show configuration section of a config file).

So here's an example of a slide created within a show for use within a specific step in that show:

```
##! show: my_show
# show_version=5
- time: 0
  slides:
    my_show_slide_1:
    - type: text
      text: MISSION PINBALL
      color: red
    - type: rectangle
      width: 128
      height: 32
```

Again, see the *show documentation* for details. Here we're just showing that it's also possible to define a slide in a show config.

How to Show a Slide on a Display

Once you have your *slides created*, you need to decide which slides you show when. (Just remember you can test slides and widgets interactively using *Interactive MC (iMC)*.)

Using the slide_player

The most common option is to use the *slide_player:* section of a config file. This can be in either your machine-wide or in mode-specific config files. (Like all mode settings, slides in a mode-based config file will only play when that mode is active.)

The slide player is based on MPF's *events system*, meaning that you basically say, "play THIS slide when THAT event happens".

For example, if you want to play a slide named "good_job" when the event "left_lane_hit" is posted, you would set your config like this:

```
slide_player:
  left_lane_hit: good_job
```

You can have as many event/slide combinations as you want, like this:

```
slide_player:
  left_lane_hit: good_job
  right_lane_hit: good_job
  left_ramp_hit: ramp_champ
```

The above examples are what we call the "express" config option since each event specifies a slide name, but no other options. (It just uses the default options for showing each slide. But instead of putting the slide name after the event name, you can also create a sub-entry with the slide name, then *another* sub-entry with additional options, like this:

```
slide_player:
  right_ramp_hit:
    ramp_hit_slide:
      expire: 2s
      target: dmd
```

You can mix-and-match all of these in a single config, like this:

```
slide_player:
  left_lane_hit: good_job
  right_lane_hit: good_job
  left_ramp_hit: ramp_champ
  right_ramp_hit:
    ramp_hit_slide:
      expire: 2s
      target: dmd
```

In the example above, when the event "left_ramp_hit" happens, the slide "ramp_champ" is shown. When the event "right_ramp_hit" happens, the slide "ramp_hit_slide" is shown, but with the additional options of setting the slide to expire (to be removed) after 2 seconds, and for that slide to show on the "dmd" display target instead of the default display.

There are many options for the slide_player in addition to the "expire" and "target" options shown above. Refer to the *slide_player:* section of the config file reference for full details.

Adding slides to a show

The slide_player is one of MPF's many *Config Players* (so called because they use a "config" section to "play" things). Config players can be used in a config file (as shown above) and also in a show step. To use the slide player in a show, you add a *slides:* section to a show step.

For example, if you want a slide called "happy_face" to play in a step in a show, you can do it like this (this is a snippet of a single step in a show):

```
##! show: my_show
- duration: 3s
  slides: happy_face
```

Again, you can use the sub-entry format to specify additional options:

```
##! show: my_show
- duration: 3s
  slides:
    happy_face:
      target: playfield_screen
```

Creating new slides in the slide_player

Both of the options we've show so far (using the *slide_player:* section of a config file and using the *slides:* section of a show) have used existing named slides that you would have already defined in the *slides:* section of a config. You also have the option to define new slides directly in each of these sections. See the *How to create slides* section of the documentation for instructions on how to do that.

Slide Transitions

When MPF switches the current slide on a display with another slide, you can set a transition effect that controls what this slide transition looks like. You can use these transitions with the slide_player and within shows. You can set transitions as a property of the new slide that comes in, or as a property of the outgoing transition when the current slide is removed. You can also control the duration (speed) of the transition.

Here's a list of all the types of transitions that MPF supports. Note that if you're reading the PDF or Epub version of this documentation, if you visit the documentation website (docs.missionpinball.org) then this page contains animated GIFs which show each of these transitions in action.

none

Setting a transition type of `none` means that no transition will be used, and the incoming slide instantly replaces the current slide.

push

The push transition means that the incoming slide "pushes" the outgoing slide out of the way. (e.g. the outgoing slide moves out while the incoming slide moves in)

Options for the push transition:

- duration: MPF *time string* Default is 1 second.
- easing: See the *easing instructions* for details.
- direction: `left`, `right`, `up` or `down`.

move_in

The move in transition means that the incoming slide moves in on top of the outgoing slide. The outgoing slide is not animated.

Options for the move_in transition:

- duration: MPF *time string* Default is 1 second.
- easing: See the *easing instructions* for details.
- direction: `left`, `right`, `top` or `bottom`.

move_out

Not working yet.

wipe

The wipe transition means that the display is wiped from the outgoing slide to the incoming one. Neither slide is animated.

Options for the wipe transition:

- duration: MPF *time string* Default is 1 second.

swap

The swap transition similates an app screen swap like on a mobile device. The outgoing slide moves out of the way and the incoming slide comes in on top of it.

Options for the swap transition:

- duration: MPF *time string* Default is 1 second.

fade

The fade transition is a classic crossfade from the outgoing slide to the incoming one.

Options for the fade transition:

- duration: MPF *time string* Default is 1 second.

fade_back

The fade_back transition causes the outgoing slide to shrink and fade away, revealing the incoming slide.

Options for the fade_back transition:

- duration: MPF *time string* Default is 1 second.

rise_in

The rise in transition causes the incoming slide to fade in and rise up from the center of the display. It's essentially the opposite of the fade_back transition.

Options for the rise_in transition:

- duration: MPF *time string* Default is 1 second.

Configuring Transitions

Transitions are specified as an additional property of a `slide_player:` config or the `slides:` section of a show config. For example:

```
slide_player:
  left_ramp_hit:
    slide1:
      transition:
        type: push
        duration: 2s
        direction: right
```

Hopefully the above example is obvious by now. When the event "left_ramp_hit" happens, MPF will show the slide called "slide1:, using the push transition, with a transition time of 2 seconds, pushing the new slide in from the right.

Transitions can be combined with other slide settings, like this:

```
slide_player:
  left_ramp_hit:
    slide1:
```

(continues on next page)

(continued from previous page)

```
      transition:
        type: push
        duration: 2s
        direction: right
```

You can also configure `transition_out:` settings which are transitions that will be applied to a slide when it is removed, like this:

```
slide_player:
  left_ramp_hit:
    slide1:
      transition:
        type: push
        duration: 2s
        direction: right
      transition_out:
        type: fade_back
```

Note: If the current slide has a `transition_out:` setting, and the new slide has a `transition:` setting, then the new slide's transition setting will take precedence.

How to configure a multiplayer display

This is an example walkthrough of creating a "score" slide that dynamically adjusts as additional players are added to the game, and displays custom player variables in addition to ball number and score.

1. Game Modes vs Player Modes

Unlike most events in MPF, adding players can occur both during a player's turn and prior to it, so it's a good idea to run your base slide as part of the "game" mode rather than inside a player-specific mode. This is easy to do by creating a mode folder and config file for the game mode, which MPF will automatically append to the default game configuration. You do not need to add any mode start/stop events to game.yaml, as those are all taken care of internally.

2. Create a base "single_player" slide in game.yaml

We'll start by creating the default single-player slide and showing it when the game starts, which will always be single-player because a second player can't be added until after the game starts.

We'll create each element of this slide as a separate widget, so we can remove different pieces one-by-one as the number of players increases. In this example, we have a large score for the current player, show the player's custom-variable "level" in the lower-left, and their current ball number in the lower-right

```
slides:
  base_slide: []
slide_player:
  single_player_ball_started: base_slide
widgets:
  base_score_widget:
    - type: text
      text: (score)
  base_level_widget:
    - type: text
      text: LVL (player|level)
  base_ball_widget:
    - type: text
      text: BALL (ball)
widget_player:
  # First event, before additional players have a chance to enter
  game_started:
    base_score_widget:
      slide: base_slide
    base_level_widget:
      slide: base_slide
    base_ball_widget:
      slide: base_slide
```

This is all we need to have a nice single-player slide that shows the score, the ball, and a custom game-specific player variable.

3. Add a Second Player

With the addition of a second player we'll shrink the main score down a little, and show a small player_1 indicator in the upper-left and a player_2 indicator in the upper-right. Because the current player's score is already shown in the middle, in this example we'll show another player variable (in this case, the player's name) where their score normally is.

The _multi_player_ball_started_ event happens when a player's turn starts and there are multiple players in the game, so this will show our multiplayer slide at the beginning of every ball.

The _multiplayer_game_ event happens as soon as a second player is added, so we use it to swap the slides if player_2 joins after player_1 has already started their turn.

Because the "game" mode exists between player turns, its slides can interfere with other slide behavior (e.g. bonus slides and start/end of turn slides). We'll keep it clean and manually clear the score slide at the end of each ball.

```
slides:
  base_slide:
    - type: text
      text: "Single Player Game"
  multiplayer_slide:
    - type: text
      text: "Multiplayer Player Game"
slide_player:
  single_player_ball_started: base_slide
  multi_player_ball_started: multiplayer_slide
  # If a second player joins during player1's turn, swap base_slide for multiplayer_slide
  multiplayer_game:
    base_slide:
      action: remove
    multiplayer_slide:
      action: play
  ball_will_end:
    base_slide:
      action: remove
    multiplayer_slide:
      action: remove
```

With only two players, we can keep the "level" and "ball" widgets in the bottom left and right corners. We want to add the player_1 and player_2 widgets in the upper corners, and swap out the big "base_score_widget" for the slightly smaller "mp_score_widget" one:

```
widget_player:
  { ... game_started: ... }

  # Player 2 has entered the game
  player_added{num==2}:
    mp_score_widget:
      slide: multiplayer_slide
    base_level_widget:
      slide: multiplayer_slide
    base_ball_widget:
      slide: multiplayer_slide
    # Start with player1 name and player2 score
    mp_name_widget_player_1:
      slide: multiplayer_slide
    mp_score_widget_player_2:
      slide: multiplayer_slide

widgets:
  { ... base_widgets ... }

  mp_score_widget:
    - type: text
```

(continues on next page)

(continued from previous page)

```
      text: (score)
      style: score_style_multiplayer

  # Player 1 score, for when they're not playing
  mp_score_widget_player_1:
    - type: text
      text: (player1|score)
      style: mp_player_1
  # Player 2 score, for when they're not playing
  mp_score_widget_player_2:
    - type: text
      text: (player2|score)
      style: mp_player_2

  # Player 1 custom variable, for when they are playing
  mp_name_widget_player_1:
    - type: text
      text: (machine|last_career_player_1)
      style: mp_player_1
  # Player 2 custom variable, for when they are playing
  mp_name_widget_player_2:
    - type: text
      text: (machine|last_career_player_2)
      style: mp_player_2
```

4. Adding a third and fourth player

Since the multiplayer slide is already being shown, adding player_3 and player_4 is as easy as swapping out the "level" and "ball" widgets for player scores.


```
widget_player:
  { ... single and player_2 widget_player ... }

  # Player 3 has entered the game
  player_added{num==3}:
    mp_score_widget_player_3:
      slide: multiplayer_slide
    # Remove the "Level" widget and place the player3 score
    base_level_widget:
      action: remove
      slide: multiplayer_slide
```

(continues on next page)

(continued from previous page)

```
  # Player 4 has entered the game
  player_added{num==4}:
    mp_score_widget_player_4:
      slide: multiplayer_slide
    # Remove the "Ball" widget and place the player4 score
    base_ball_widget:
      action: remove
      slide: multiplayer_slide

widgets:
  { ... single and player_2 widgets .. }

  mp_score_widget_player_3:
    - type: text
      text: (player3|score)
      style: mp_player_3
  mp_score_widget_player_4:
    - type: text
      text: (player4|score)
      style: mp_player_4
  mp_name_widget_player_3:
    - type: text
      text: (machine|last_career_player_3)
      style: mp_player_3
  mp_name_widget_player_4:
    - type: text
      text: (machine|last_career_player_4)
      style: mp_player_4
```

5. Turn-by-turn display changes

It’s nice showing a custom player variable for the current player during their turn, but we want to swap out that widget for their score after their turn ends.

Because we don’t know how many players there will be, it’s not safe to use the next player’s turn to reset the previous player’s widget. Instead, we set each player’s custom variable widget at the start of their turn and reset their score widget at the end of their turn.

```
widget_player:
  { ... base and multi-player widgets ...}

  # Player Turns: Swap scores -> names when turn starts, revert when turn ends
  player_turn_started{number==1}:
    mp_score_widget_player_1:
      action: remove
    mp_name_widget_player_1:
      slide: multiplayer_slide
  player_turn_ended{number==1}:
    mp_score_widget_player_1:
      slide: multiplayer_slide
    mp_name_widget_player_1:
```

(continues on next page)

(continued from previous page)

```
      action: remove
  player_turn_started{number==2}:
    mp_score_widget_player_2:
      action: remove
    mp_name_widget_player_2:
      slide: multiplayer_slide
  player_turn_ended{number==2}:
    mp_score_widget_player_2:
      slide: multiplayer_slide
    mp_name_widget_player_2:
      action: remove
  player_turn_started{number==3}:
    mp_score_widget_player_3:
      action: remove
    mp_name_widget_player_3:
      slide: multiplayer_slide
  player_turn_ended{number==3}:
    mp_score_widget_player_3:
      slide: multiplayer_slide
    mp_name_widget_player_3:
      action: remove
  player_turn_started{number==4}:
    mp_score_widget_player_4:
      action: remove
    mp_name_widget_player_4:
      slide: multiplayer_slide
  player_turn_ended{number==4}:
    mp_score_widget_player_4:
      slide: multiplayer_slide
    mp_name_widget_player_4:
      action: remove
```

How to do "Picture in Picture" display

MPF uses a *window* to define the area on a screen that can be used to display graphics. This window and be further subdivided into *displays* that define areas of the window onto which slides and their widgets can be projected.

Here is an example of setting up a window and four displays in a config file.

```
window:
  width: 1080
  height: 1300
  title: CupheadWindow
  resizable: true
  fullscreen: false
  borderless: true
  exit_on_escape: true
  top: 0
  left: 0

displays:
```

(continues on next page)

(continued from previous page)

```
  insert:
    width: 100
    height: 100
  upper:
    width: 1040
    height: 280
  middle:
    width: 1040
    height: 580
  lower:
    width: 1040
    height: 320
```

A layout slide can then be made that sets the locations of each of these displays on a window. The x and y locations are relative to the lower left corner of the window. The order in which you define each widget determines which widget has priority over the other widgets. In this example the "insert"display is defined before the "lower" display so the "insert" display will be drawn on top of the "lower" display. This gives you a "picture-in-picutre" where the "insert" will appear to be projected on top of the "lower" display.

```
slides:
  layout:
    background_color: blue
    widgets:
      - type: display
        width: 69
        height: 65
        x: 60
        y: 200
        anchor_x: left
        anchor_y: top
        source_display: insert
      - type: display
        width: 1040
        height: 280
        x: 20
        y: 1270
        anchor_x: left
        anchor_y: top
        source_display: upper
      - type: display
        width: 1040
        height: 580
        x: 20
        y: 940
        anchor_x: left
        anchor_y: top
        source_display: middle
      - type: display
        width: 1040
        height: 320
        x: 20
        y: 340
```

(continues on next page)

(continued from previous page)

```
        anchor_x: left
        anchor_y: top
        source_display: lower
```

How to configure a "split screen" display

This is an example walk through of how to create a screen that is split into several smaller screens that can be independently controlled. Let's create a layout similar to Wizard Of Oz with four quadrants and a score display area at the bottom.

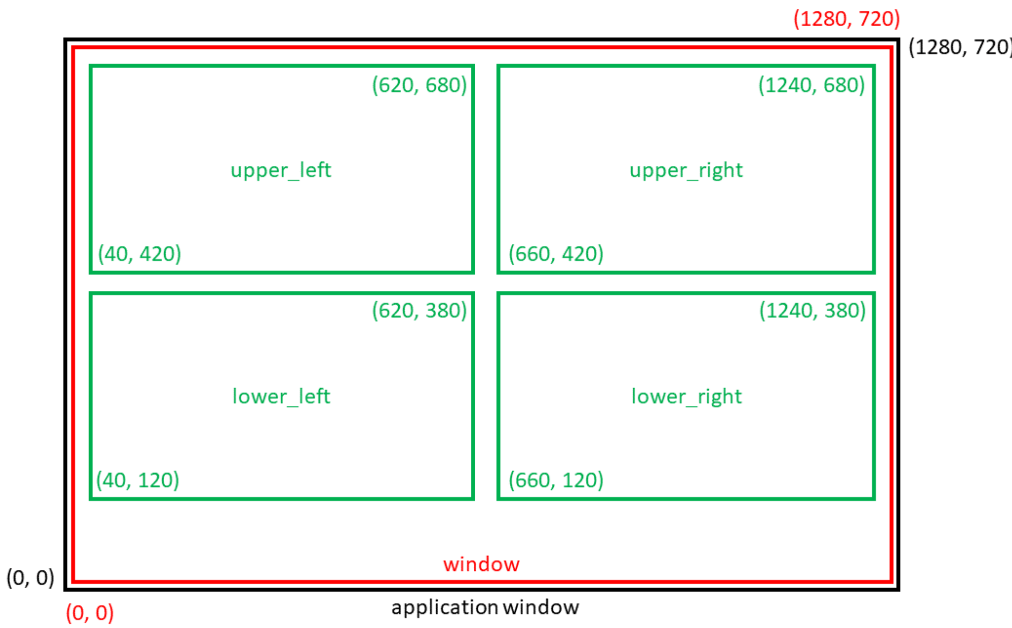
1. Create the displays

Let's start by defining the displays we will need for the layout. Remember that displays are an internal representation of a blank canvas that holds graphical content. They are logical and are targets for showing slides. We will need to define and configure five displays for this layout: one that covers the entire window area, and four smaller ones that will each be used for one of the four smaller quadrant displays.

```
#config_version=5
window:
  width: 1280
  height: 720
  resizable: false
  fullscreen: false
  borderless: true
  exit_on_escape: true
displays:
  window:
    width: 1280
    height: 720
  upper_left:
    width: 580
    height: 260
  upper_right:
    width: 580
    height: 260
  lower_left:
    width: 580
    height: 260
  lower_right:
    width: 580
    height: 260
```

2. Create the layout slide

The next step is to decide on the details of the layout. The following diagram shows the layout we will be creating, along with the lower left and upper right coordinates of each display widget based on a 1280 x 720 pixel main window.



To accomplish this in MPF, we will need to create a slide that will be shown in the main window display that will contain display widgets for each of the four smaller displays. The `source_display:` attribute of each display widget will be set to the corresponding display name that we want to display in each quadrant.

Note: While the display widgets will automatically scale the display contents to fit in the widget boundaries, it is recommended you use the same size display widget as the source display for the best visual results.

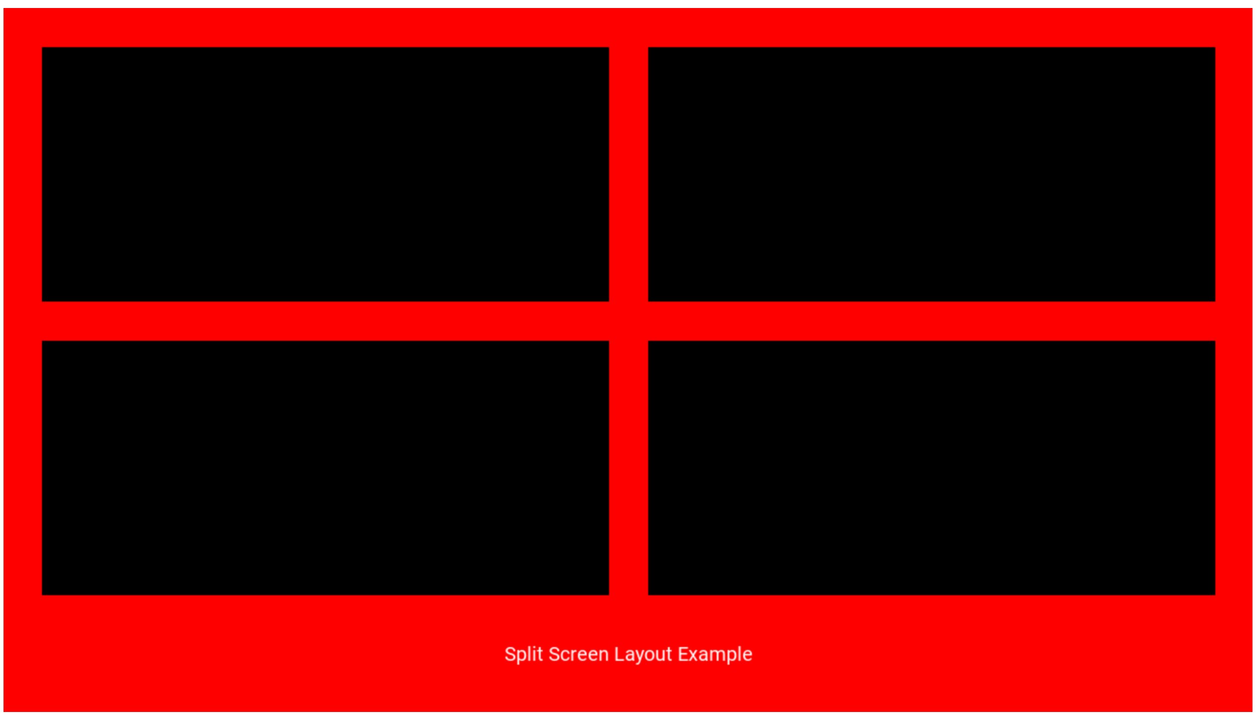
```
slides:
  layout_4_mini:
    background_color: red
    widgets:
      - type: display
        width: 580
        height: 260
        x: 40
        y: 420
        anchor_x: left
        anchor_y: bottom
        source_display: upper_left
      - type: display
        width: 580
        height: 260
        x: 660
        y: 420
        anchor_x: left
        anchor_y: bottom
        source_display: upper_right
```

(continues on next page)

(continued from previous page)

```
      - type: display
        width: 580
        height: 260
        x: 40
        y: 120
        anchor_x: left
        anchor_y: bottom
        source_display: lower_left
      - type: display
        width: 580
        height: 260
        x: 660
        y: 120
        anchor_x: left
        anchor_y: bottom
        source_display: lower_right
      - type: text
        text: Split Screen Layout Example
        y: 60
slide_player:
  mc_ready:
    layout_4_mini:
      target: window
```

The above config will display the `layout_4_mini` slide we just created as soon as the media controller is ready. Here is the result of the above config:



3. Create additional slides and show them on one of the smaller displays

Now that we have the desired layout, we can create additional content to show in any one of the smaller displays. It is just as simple as creating slides and setting their target value to the name of the desired display when showing them. Here is our example from the previous steps that has now been extended to show a simple slide in each of the four quadrants:

```
#config_version=5
window:
  width: 1280
  height: 720
  resizable: false
  fullscreen: false
  borderless: true
  exit_on_escape: true
displays:
  window:
    width: 1280
    height: 720
  upper_left:
    width: 580
    height: 260
  upper_right:
    width: 580
    height: 260
  lower_left:
    width: 580
    height: 260
  lower_right:
    width: 580
    height: 260
slides:
  layout_4_mini:
    background_color: red
    widgets:
      - type: display
        width: 580
        height: 260
        x: 40
        y: 420
        anchor_x: left
        anchor_y: bottom
        source_display: upper_left
      - type: display
        width: 580
        height: 260
        x: 660
        y: 420
        anchor_x: left
        anchor_y: bottom
        source_display: upper_right
      - type: display
        width: 580
        height: 260
        x: 40
```

(continues on next page)

(continued from previous page)

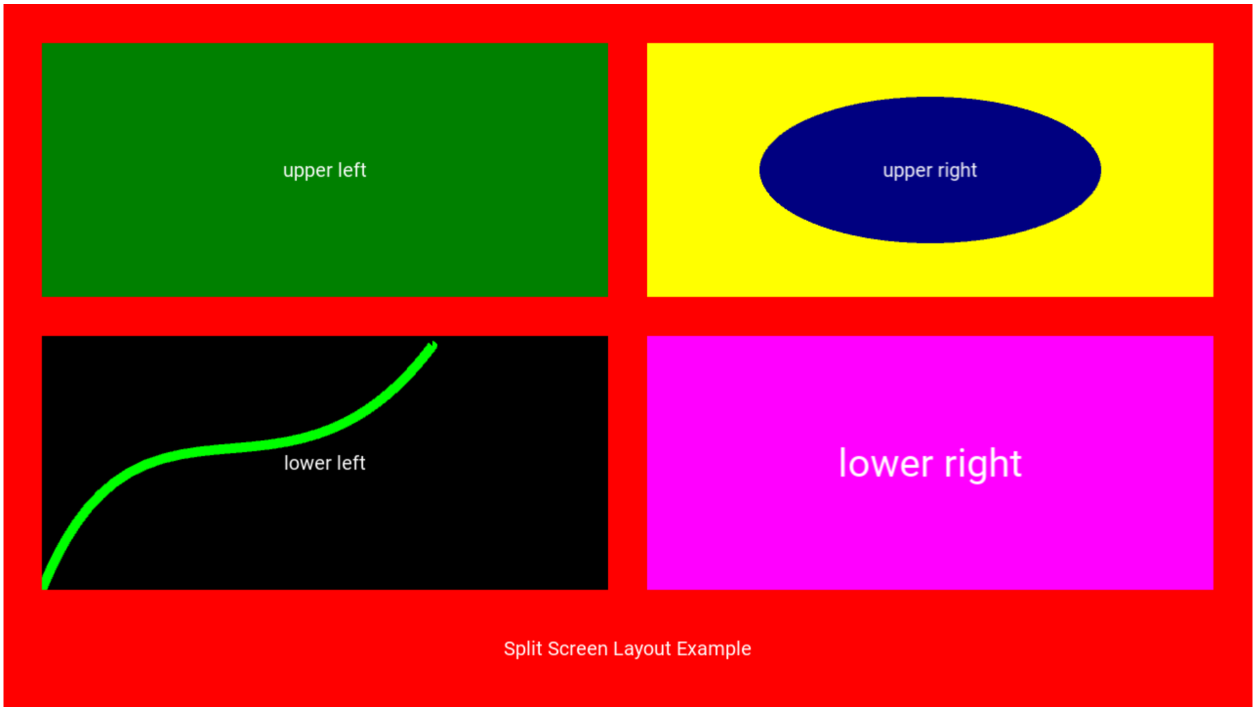
```
        y: 120
        anchor_x: left
        anchor_y: bottom
        source_display: lower_left
      - type: display
        width: 580
        height: 260
        x: 660
        y: 120
        anchor_x: left
        anchor_y: bottom
        source_display: lower_right
      - type: text
        text: Split Screen Layout Example
        y: 60
  slide_1:
    background_color: green
    widgets:
      - type: text
        text: upper left
  slide_2:
    background_color: yellow
    widgets:
      - type: text
        text: upper right
      - type: ellipse
        color: navy
        height: 150
        width: 350
  slide_3:
    widgets:
      - type: text
        text: lower left
      - type: bezier
        points: 0, 0, 100, 250, 250, 50, 400, 250
        color: lime
        thickness: 5
  slide_4:
    background_color: magenta
    widgets:
      - type: text
        text: lower right
        font_size: 40
slide_player:
  mc_ready:
    layout_4_mini:
      target: window
    slide_1:
      target: upper_left
    slide_2:
      target: upper_right
    slide_3:
      target: lower_left
    slide_4:
```

(continues on next page)

(continued from previous page)

```
    target: lower_right
```

The above config results in the following output:



4. Conclusion

You should now have a good working example on how to create a split screen layout and how easy it is to target your slides to a specific display. You could easily extend this example to display the current scores in the bottom section of the layout or put nice frames or other graphics around the displays. Remember, if you target the `window` display with a different slide your `layout_4_mini` slide will be replaced and your nice 4 quadrant layout will no longer be visible. This allows you to have an infinite number of possible layout slides and change them according to the context of your game.

Display Targets

Help us to write it

Slides Events

These events can be useful within players. For example, if you want to play 3 slides as a mode begins then the *mode_(name)_started* event can trigger the slide_1 - but what triggers slide_2 and slide_3?

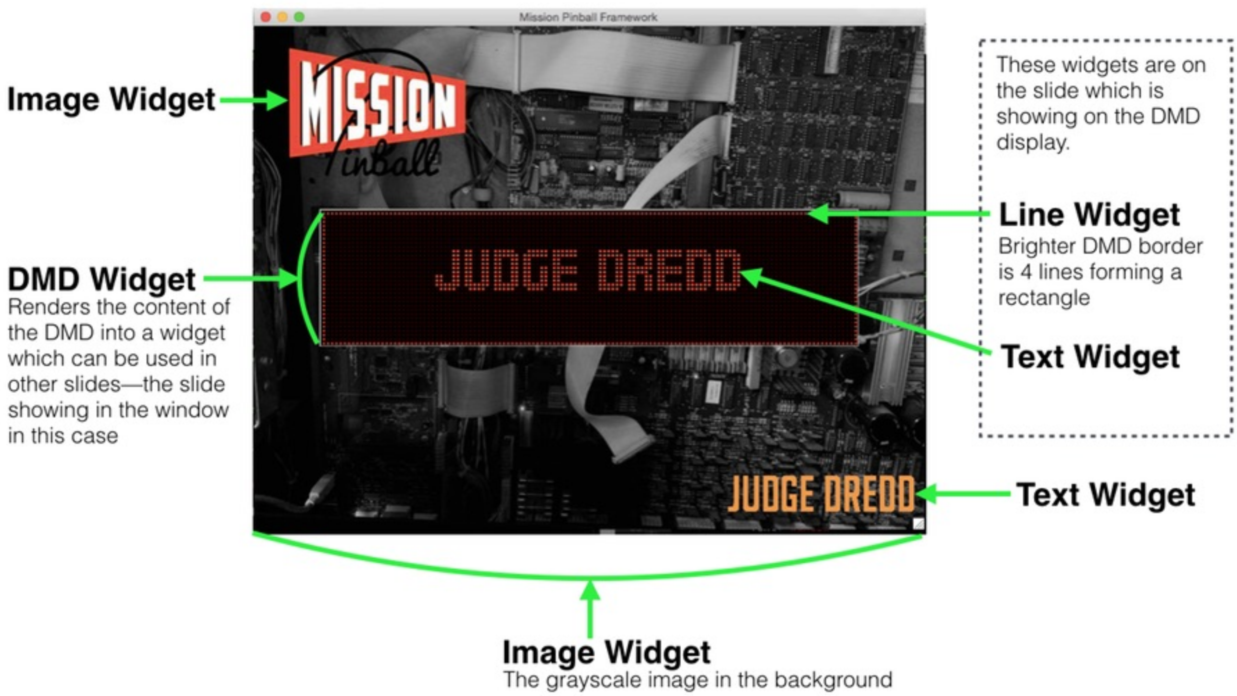
The slide_player: can be used to sequence the playing of additional slides using the slide_slide_1_removed event to trigger the next slide to be played.

Related Events

- *slide_(name)_created*
- *slide_(name)_removed*
- *slide_(name)_active*

Widgets

If a slide is a blank canvas, then "widgets" are the things you put on that blank canvas, like text, images, shapes, videos, etc. Here's an example of a slide (on the window display) showing how it's made up of different types of widgets.



Widgets have properties like size and position, and some widgets include additional properties depending on what type of widget they are. (Text widget have font properties, video widgets have properties controlling video playback, etc.)

You can control the stacking order of widgets on a slide (also called the "layer" or "z-order"), to specify which widget should be on top of another if they're overlapping.

You can specify all the widgets that are on a slide when you define that slide, and/or you can add widgets later to existing slides or remove certain widgets from slides while keeping others there.

You can even create a library of reusable "named" widgets which you can use again and again on many slides.

You can specify widget "styles" which are default properties that are inherited by all widgets based on that style. (So, for example, you could specify a set of styles for text widgets called "title", "default" and "small" that control the font name, font size, color, and spacing for widgets using that style.

Individual widget properties can also be animated, meaning you can change the size, position, opacity, etc. of a widget over time. You can animate multiple properties of a widget at the same time or in a sequence (or both), and you can specify which MPF events trigger animation sequences to start and stop.

In this section of the documentation, we'll look at all the different types of widgets (and their properties and settings), then look at how you position and animate them, how to use widget styles, and how you can create the reusable widgets.

You can test slides and widgets interactively using *Interactive MC (iMC)*.

Types of Widgets

Most popular

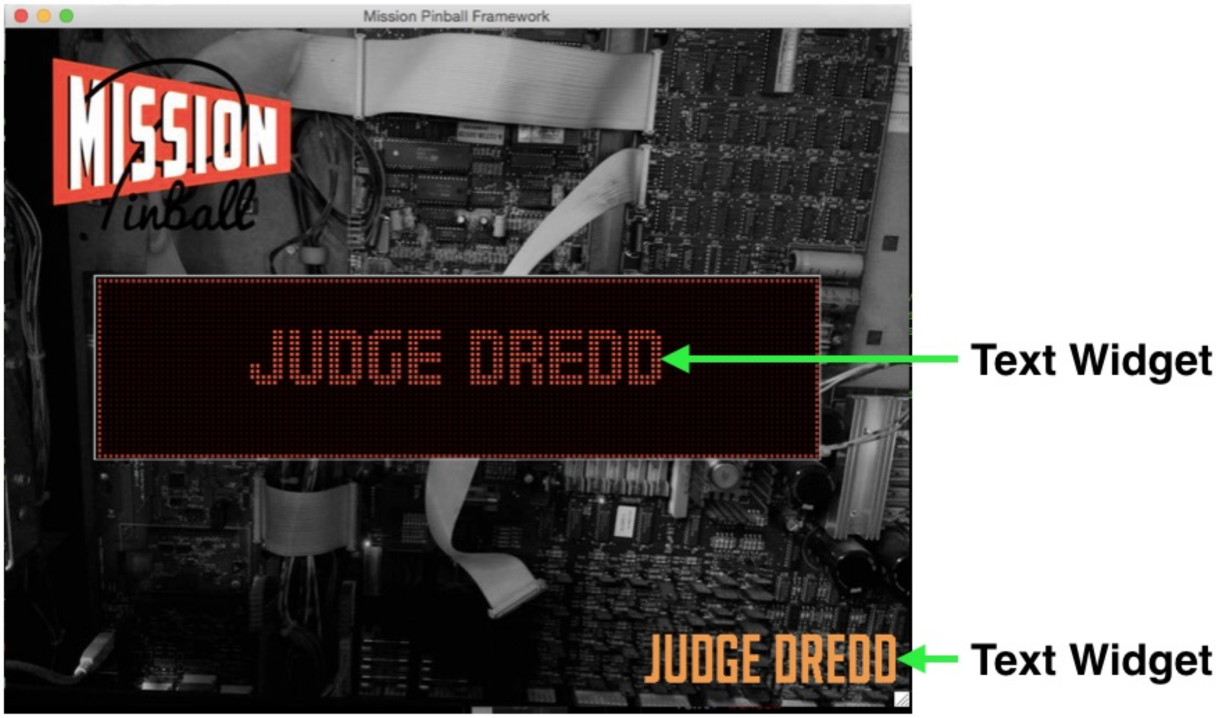
- Text
- Image
- Video

But also these:

- Bezier Curve
- Display
- Ellipse
- Line
- Points
- Quad
- Rectangle
- Segment Display Emulator
- Text Input
- Triangle
- Common Settings (all widgets)

Text Widget

The text widget is used to show text on a *slide*.



In addition to being able to specify static text, text widgets also include powerful functionality:

- You can configure dynamic text that is automatically updated (in real time) based on the value of a player variable or a machine variable.
- You can configure a placeholder "text string" that uses a lookup value to get its actual text. This is useful for things like multi-language support, or to be able to have different text strings based on a configuration file (family-friendly versus R-rated text, etc.)
- You can configure fonts and font styles to be automatically applied to text, and you can override them on a widget-by-widget basis.

You can also use *bitmap fonts* to customize fonts for your machine.

Settings

Here are a list of the settings you can use for text widgets:

```
type: text
text:
font_size:
font_name:
bold:
italic:
casing:
number_grouping:
min_digits:
halign:
valign:
```

Note: Text widgets also have "common" widget settings for position, opacity, animations, color, style, etc. Those are not listed here, but are instead covered in *common widget settings* page.

Also remember that all widget settings can be controlled via *widget styles*, rather than you having to set every setting on every widget.

The following text widget settings may be *animated*: `x:`, `y:`, `font_size:`, `color:`, `opacity:`, `rotation:`, and `scale:`.

type: text

Tells MPF that this is a text widget. This setting is required when using text widgets.

text:

This value is required. If you don't want text, use ""

Your text can contain placeholders as described in *dynamic text*.

Newline characters (\n) are supported in text values to create multiple lines with line breaks, however you must surround the text with quotes or the backslash will be treated as a printing character and will appear in the output. For example:

```
text: "Multiple\nlines"
```

will create multiple text lines with a line break, while the following will not:

```
text: Multiple\nlines
```

font_name:

The name of the font you want to use. This is the name only, without the file extension. For example:

Correct:

```
font_name: arial
```

Wrong:

```
font_name: arial.ttf
```

There's a lot that goes into fonts, so we have a whole section on *fonts* which you should read.

Usually fonts are controlled via *widget styles*. Also, if you're using a DMD or color DMD (or other pixel-style display), we have some *built in DMD fonts* that you can use which are pre-configured for DMDs.

bitmap_font:

A true/false value indicating whether the *font_name:* setting contains the name of a *bitmap_font* asset. When set to *True*, *font_name:* must refer to an existing bitmap_font asset name and *font_size:* will be ignored. When set to *False*, *font_name:* should refer to a font name.

font_size:

The size of the font (in points). Default is 15.

See the *full documentation on fonts* for details.

bold:

Boolean (True/False or Yes/No) which controls whether this font is bold. Note that this setting attempts to over-draw the font a few times to make it look bold, so the results are often not that great. You're better off finding an actual bold version of your font and using that font instead.

The default setting is `False`.

italic:

Boolean (True/False or Yes/No) which controls whether this font is italicized. Note that this setting simply skews the font when it's drawn, so the results are often not that great. You're better off finding an actual italicized version of your font and using it instead.

The default setting is `False`.

casing:

A string value that changes the casing of the text on the widget. Available values are:

- "lower": all characters will be lower case
- "upper": ALL CHARACTERS WILL BE UPPER CASE
- "title": All First Characters Are Capitalized
- "capitalize": Only the first character is capitalized

The default setting is `None` and the characters are displayed as-is.

number_grouping:

Boolean (True/False or Yes/No) which controls whether you want the separator between digits. In other words, it converts `1234567` into `1,234,567`.)

Note that this setting will search through the text string for digits and then insert the commas. In other words, if your text is "YOU SCORED 12345 POINTS", then it will convert it into "YOU SCORED 12,345 POINTS" even though the text is a mix-and-match of letters and numbers.

The default setting is `False`. (Note that prior to MPF 0.30, the default setting was `True`.)

Note: Currently this setting only inserts a comma. We need to add a setting to allow other characters (like a period which is common in Europe). If this is you, post a message to the forum and we'll bump up the priority on our to-do list.

min_digits:

Configures the minimum number of digits for the text to be displayed. This setting adds zeros to the left for digits that are shorter than the setting.

This is typically used in score displays, since pinball machines usually show a score as `00` instead of `0` when the player starts the game and has no points.

So for most machines, you'd add `min_digits: 2` to your text widgets which show the player's score.

The default setting is `0`.

halign:

Specifies the horizontal alignment of the text within the bounding box. Note that this setting *is not used* to align a widget on the screen. (See the *How to position widgets on slides* documentation for details on that.)

This setting is almost never used in MPF because the bounding box of a text widget is automatically created and sized based on the actual text and font chosen.

The default setting is `center`.

valign:

Specifies the vertical alignment of the text within the bounding box. Note that this setting *is not used* to align a widget on the screen. (See the *How to position widgets on slides* documentation for details on that.)

This setting is almost never used in MPF because the bounding box of a text widget is automatically created and sized based on the actual text and font chosen.

The default setting is `middle`.

anchor_y: baseline

Text widgets have an additional `baseline` option in addition to the other baseline options detailed in the *common widget settings* documentation.

Examples

The example config files section of the documentation contains *examples of text widgets*.

Dynamically Updating Text

Related Config File Sections
text widgets
segment displays
segment display players

Your text can contain placeholders which will be replaced and updated when the text is shown. Use `(param)` to replace the parameters of the event which triggers the text (usually you do not want to use this). Player vars from the current player can be accessed using `(player|var)` (e.g. `(player|score)` or `(player|ball)`). Furthermore, you can target a specific player using `(playerX|var)` where X is the player number starting at 1 (e.g. `(player1|score)`). To display machine variables use `(machine|var)` (e.g. `(machine|credit_string)`).

Text Substitution Strings

Image Widget

The image widget is used to display an image on a *slide*. It's also used to display animated images, which can either be animated GIFs or a folder or zip file of sequentially-numbered images (of any type).

Image types that support alpha channels (like PNGs) are supported.

Settings

Note: Image widgets also have "common" widget settings for position, opacity, animations, color, style, etc. Those are not listed here, but are instead covered in *common widget settings* page.

Also remember that all widget settings can be controlled via *widget styles*, rather than you having to set every setting on every widget.

The following image widget settings may be *animated*: `x:`, `y:`, `color:`, `rotation:`, `scale:`, `fps:`, `current_frame:`, and `opacity:`.

type: image

Single value, type: `string`.

Tells MPF that this is an image widget

image:

Single value, type: string name of a *image*.

The name of the image asset this widget will show. Details on image assets are *here*.

fps:

Single value, type: `integer`. Default: `10`.

For animated images, sets how fast it plays (frames per second).

loops:

Single value, type: `integer`. Default: `-1`.

The number of times an animated image will loop. Set to `-1` for unlimited. Note this is now consistent in 0.50 with other areas of MPF. In earlier versions of MPF this setting used `0` to specify unlimited loops.

auto_play:

Single value, type: `boolean` (Yes/No or True/False). Default: `True`

If the image is an animated image, configures whether it plays automatically when it's loaded.

This is good for looping images, but if you have an image you want to play at a specific point, you probably want to set this to no and play it from specific events via the widget player.

start_frame:

Single value, type: `integer`. Default: `0`.

Which start frame to use for animated images.

persist_frame:

Single value, type: `boolean` (`true`/`false`). Default: `false`

When true, the animated image widget will remember the frame the it was on when it was last used, and restore that frame when the widget is next used.

By default, an animated image will reset itself each time it is added to a slide.

Video widget

The video widget is used to display a video on a *slide*. This can either be full-screen videos or smaller videos that appear on a portion of the display.

Note that in MPF, videos are regular widgets, so they can go on top of other widgets, or other widgets can go on top of them, they can be moved and animated, etc.

Settings

```
type: video
video:
height:
width:
volume:
auto_play:
end_behavior:
control_events:
```

Note: Video widgets also have "common" widget settings for position, opacity, animations, color, style, etc. Those are not listed here, but are instead covered in *common widget settings* page.

Also remember that all widget settings can be controlled via *widget styles*, rather than you having to set every setting on every widget.

The following video widget settings may be *animated*: `x:` and `y:`.

type: video

Tells MPF that this is an image widget

video:

The name of the video asset this widget will show. Details on video assets are *here*.

height:

Allows you to specify the size (along with `width:` of the video on the screen). Set to `0` (or leave this setting out) to play the video at whatever size the asset is configured for (or, if a size is not specified there, at the native video size).

Note that the `height:` and `width:` settings cannot stretch or skew the video. So if you enter values that result in an aspect ratio for the video widget that does not match the video itself, then the video will be sized as large as it can within the bounds of the size of the widget.

width:

Lets you specify the width of the video. Set to `0` (or leave the setting out) to use the setting from the video asset and/or the native video width.

See the `height:` setting above for details.

volume:

Volume for this video on a scale from `0` to `1`. Default is `1.0`. Note that you can the volume during playback via the `control_events:` below.

Note: Currently the video volume and playback is not integrated with the rest of MPF's sound system in terms of tracks, ducking, etc. This is on our roadmap.

auto_play:

Boolean (True/False or Yes/No) which controls whether this video should start playing automatically. Default is `True`.

end_behavior:

Sets what happens when the video ends. Options include:

loop The video loops and starts playing again

pause The video stops and stays at the end (so it continues showing the final frame)

stop The video stops and the position is reset back to the beginning. This is the default.

control_events:

Control the playback of this video with MPF events. Options include:

play Starts playing the video from its current position.

pause Pauses the video at its current position.

stop Stops the video and resets the position back to the beginning.

seek Moves the video to a certain position based on a percentage. `0` is the beginning of the video, `1` is the end, `0.5` is 50% through, etc. (This is similar to `position:`, except it's based on percent instead of position.

This setting does not change the play/stop state.

position Moves the video to a certain position based on the time, (in seconds). In other words `value: 4.2` here would move the video to the 4.2 second mark. (This is similar to `seek:` except it's based on seconds instead of percent.)

volume Sets the volume of the video on a scale from `0` to `1`.

This setting does not change the play/stop state.

To use control events, add a `control_events:` section to the video widget, then create a list (with dashes) of `event:`, `action:` and (optionally) `value:` settings. Then when the event is posted, the action will be applied to the video.

Consider the example below:

```
slides:
  my_slide:
    - type: video
      video: my_video
      control_events:
        - event: play_my_vid
```

(continues on next page)

(continued from previous page)

```
        action: play
      - event: wizard_caught
        action: stop
      - event: some_event
        action: pause
      - event: what_event
        action: seek
        value: .5
      - event: move_it
        action: position
        value: 4.2
      - event: mute_me
        action: volume
        value: 0
```

In the example above, when the event *play_my_vid* is posted, the video will start playing. When the event *wizard_caught* is posted, the video will stop. *some_event* will pause the video, *what_event* will reset the video to the 50% position, *move_it* will set the video to the 4.2 second position, and *mute_me* will set the volume to zero.

Note that you can have as many different entries as you want here, even using different events for the same actions, etc.

Bezier Curve Widget

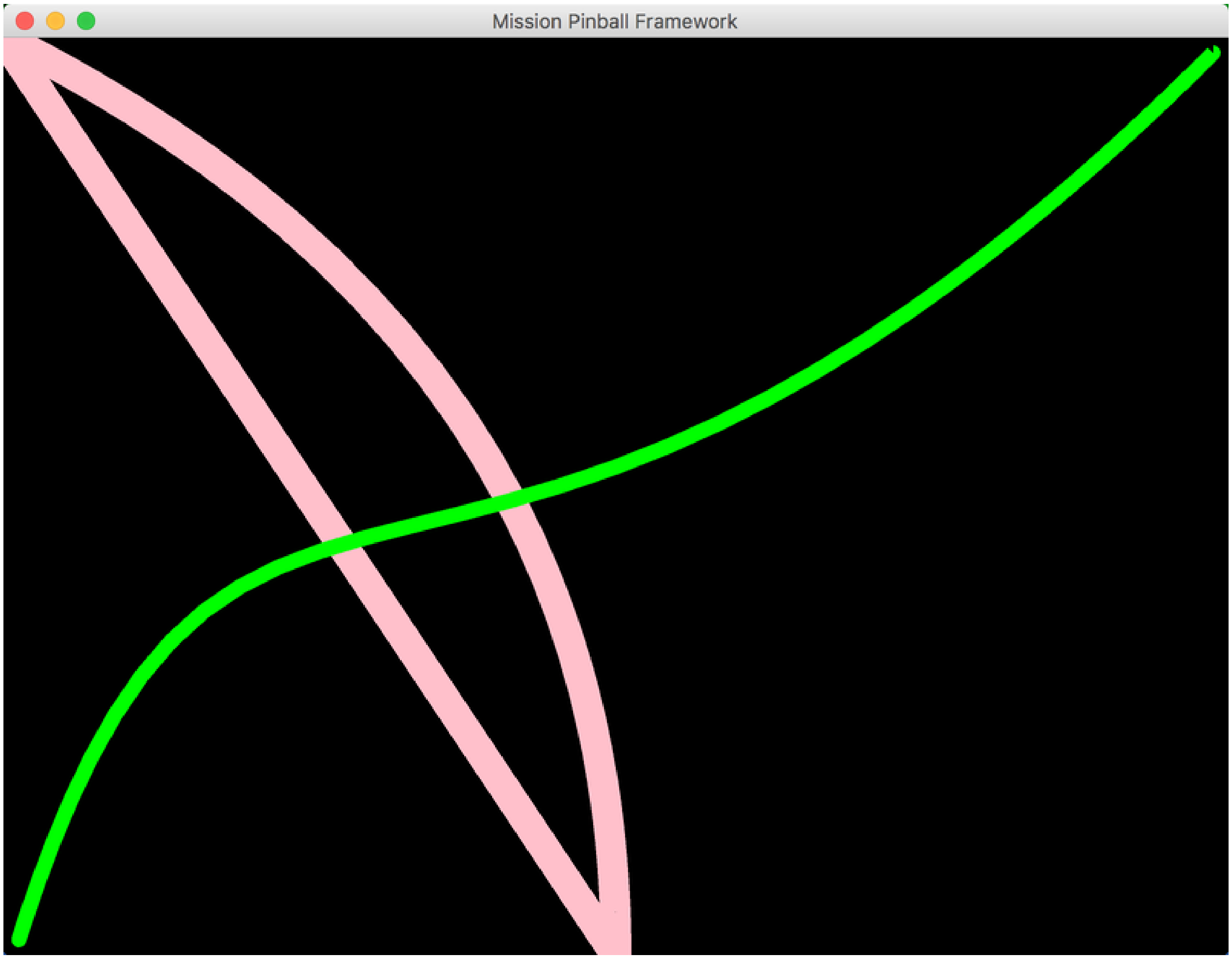
The bezier widget is used to draw a curved line on a *slide*. (Note that if you want to draw a straight line, you can use the *Line Widget*.)

Here's an example:

TODO This example just shows a blank slide in MPF 0.50?

```
#config_version=5
slide_player:
  mc_ready:
    bezier_example:
      - type: bezier
        points: 10, 10, 150, 450, 300, 100, 790, 590
        color: lime
        thickness: 5
        cap: square
      - type: bezier
        points: 0, 600, 400, 400, 400, 0
        color: pink
        close: true
        joint: miter
        thickness: 10
```

Which results in the following:



Settings

```
type: bezier
points:
thickness:
cap:
joint:
cap_precision:
joint_precision:
close:
precision:
```

Note: Bezier widgets also have "common" widget settings for position, opacity, animations, color, style, etc. Those are not listed here, but are instead covered in *common widget settings* page.

Also remember that all widget settings can be controlled via *widget styles*, rather than you having to set every setting on every widget.

The following bezier widget settings may be *animated*: `color:`, `thickness:`, `opacity:`, `points:`, `rotation:`, and `scale:`.

type: bezier

Tells MPF that this is a bezier curve widget. This setting is required when using bezier curve widgets.

points:

A list of points which make up the bezier curve, expressed in x/y pairs (so the number of items here has to be even).

The first pair is the starting point. The last pair is the ending point. Each pair in between is a point the curve will pass through.

For example:

```
points: 10, 10, 200, 50, 300, 200
```

This would draw a bezier curve starting at (10,10) and ending at (300,200), with a center point at (200, 50).

thickness:

The thickness of the line. You'll probably have to play with different settings to get it right. The default is `1.0`, so `2.0` is twice as thick as the default, `0.5` is half as thick, etc.

cap:

Determine the cap of the line, defaults to 'round'. Can be one of 'square' or 'round'

joint:

Determine the join of the line, defaults to 'round'. Can be one of 'none', 'round', 'bevel', 'miter'.

cap_precision:

Integer, defaults to 10.

Number of segments for drawing the "round" joint, defaults to 10. The joint_precision must be at least 1.

joint_precision:

Integer, defaults to 10.

Number of segments for drawing the "round" joint, defaults to 10. The joint_precision must be at least 1.

close:

Boolean (True/False), default is `False`.

If `True`, the line will be closed.

precision:

Integer, defaults to 180.

The number of individual segments that will be drawn between each pair of points.

Examples

The example config files section of the documentation contains *examples of bezier widgets*.

Display Widget

Display Widget Effects

Effects are used to apply a variety of fancy graphical effects to the contents of a display widget. The most commonly used effects are `dmd` and `color_dmd` which create the look of hardware DMDs (in version of MPF prior to 0.50, these were previously their own widget types). Multiple effects may be combined in a chain, however, effects can be CPU/GPU intensive!

Required settings

The following sections are required for the `effects` setting of the display widget:

type:

Single value, type: one of the options listed below.

The `type:` setting controls which effect will be loaded to process the display widget output. Here is the list of available effect types (the settings for each type are found below):

- `anti_aliasing` applies a very basic anti-aliasing
- `color_channel_mix` swaps color channels
- `color_dmd` creates an RGB DMD look
- `colorize` applies a color tint
- `dmd` creates a monochrome DMD look
- `dot_filter` creates a dot filter to look like individual round dots/pixels (similar to a DMD)
- `flip_vertical` vertically flips the contents
- `gain` applies a gain (brightness) adjustment
- `gamma` applies a gamma correction

- `glow` applies a pulsing glow effect
- `horizontal_blur` Gaussian blurs horizontally
- `invert_colors` inverts the colors
- `linear_gradient` applies a linear gradient tint
- `monochrome` converts the image to monochrome/grayscale
- `pixelate` pixelates the image
- `reduce` reduces the number of bits per color channel (reducing the number of resulting colors)
- `scanlines` displays flickering scanlines (like an old CRT)
- `vertical_blur` Gaussian blurs vertically

Settings for *anti-aliasing* effect:

The `anti-aliasing` effect does not have any settings.

Settings for *color_channel_mix* effect:

order:

List, type: `int`. Default: `[1, 2, 0]`

The new sorted order of the rgb channels. The list must contain an arrangement of the list `[0, 1, 2]`.

Settings for *color_dmd* effect:

dot_filter:

Single value, type: `boolean` (Yes/No or True/False). Default: `True`

Enables the “dot” look. Setting this to `False` means that the color DMD will not have dots.

dots_x:

Single value, type: `int`. Default: `128`

The number of DMD dots in the x direction.

dots_y:

Single value, type: `int`. Default: `32`

The number of DMD dots in the y direction.

blur:

Single value, type: `float`. Default: `0.1`

This is the radius of the "glow" of the pixels (when using `dot_filter: True`). This is expressed as a decimal relative to the size of the pixels. The default is `0.1` which means there's a 10% glow radius.

dot_size:

Single value, type: `float`. Default: `0.7`

The size of the individual "dots", expressed as a decimal relative to what their full size would be. A value of `1.0` will mean that each pixel will fill 100% of the space (e.g. no space in between), and it won't really look like separate pixels.

background_color:

Single value, type: `kivycolor`. Default: `191919ff`

The background color of the display (the color of the pixels when they're "off"). Note: this is a color with alpha channel value.

gain:

Single value, type: `float`. Default: `1.0`

A numeric multiplier that will be applied to every color channel of every pixel in this color DMD widget (brightness).

For example, if you set `gain: 1.2`, then a pixel on this color DMD's source display that has a color of (100, 100, 100) will be drawn with the color (120, 120, 120). (Each element multiplied by 1.2). Note that values above 255 will be capped at 255.

The default is 1.0 which means that the original colors are unchanged. You can play with this to act as a "poor man's" brightness control, but values too far above or below 1.0 will probably look weird.

shades:

Single value, type: `int`. Default: `0`

This is the number of shades each color channel will be reduced to. The default is `0` which disables it and uses the full 256 shades per color channel, meaning the color DMD widget will use have 256 shades each of red, green, and blue. (In other words, the default is standard 24-bit color for a total of 16.7m colors.)

Note that this setting can produce weird results depending on your source content. If you want an old school look, you might have better luck creating your videos and graphics with fewer colors and then not setting the shades option here.

Also note if you want to use full color (no shade reduction), it's better to set this to `0` and not `256` since 0 will disable this processing which will be less overhead.

Settings for *colorize* effect:

tint_color:

Single value, type: `kivycolor`. Default: `ff66ff00`

The color to tint the pixels in the display.

Settings for *dmd* effect:

dot_filter:

Single value, type: `boolean` (Yes/No or True/False). Default: `True`

Enables the "dot" look. Setting this to `False` means that the DMD will not have dots.

dots_x:

Single value, type: `int`. Default: `128`

The number of DMD dots in the x direction.

dots_y:

Single value, type: `int`. Default: `32`

The number of DMD dots in the y direction.

blur:

Single value, type: `float`. Default: `0.1`

This is the radius of the "glow" of the pixels (when using `dot_filter: True`). This is expressed as a decimal relative to the size of the pixels. The default is `0.1` which means there's a 10% glow radius.

dot_size:

Single value, type: `float`. Default: `0.7`

The size of the individual "dots", expressed as a decimal relative to what their full size would be. A value of `1.0` will mean that each pixel will fill 100% of the space (e.g. no space in between), and it won't really look like separate pixels.

background_color:

Single value, type: `kivycolor`. Default: `191919ff`

The background color of the display (the color of the pixels when they're "off"). Note: this is a color with alpha channel value.

gain:

Single value, type: `float`. Default: `1.0`

A numeric multiplier that will be applied to every color channel of every pixel in this color DMD widget (brightness).

For example, if you set `gain: 1.2`, then a pixel on this color DMD's source display that has a color of (100, 100, 100) will be drawn with the color (120, 120, 120). (Each element multiplied by 1.2). Note that values above 255 will be capped at 255.

The default is 1.0 which means that the original colors are unchanged. You can play with this to act as a "poor man's" brightness control, but values too far above or below 1.0 will probably look weird.

shades:

Single value, type: `int`. Default: `0`

This is the number of shades each color channel will be reduced to. The default is `0` which disables it and uses the full 256 shades per color channel, meaning the color DMD widget will use have 256 shades each of red, green, and blue. (In other words, the default is standard 24-bit color for a total of 16.7m colors.)

Note that this setting can produce weird results depending on your source content. If you want an old school look, you might have better luck creating your videos and graphics with fewer colors and then not setting the shades option here.

Also note if you want to use full color (no shade reduction), it's better to set this to `0` and not `256` since 0 will disable this processing which will be less overhead.

luminosity:

List, type: `float`. Default `[.299, .587, .114]`

This defines the luminosity factor for each color channel. The value for each channel must be between 0.0 and 1.0.

dot_color:

Single value, type: `kivycolor`. Default: `ff5500`

The color of the dots in the DMD. Defaults to classic DMD orange.

Settings for *dot_filter* effect:

dots_x:

Single value, type: `int`. Default: `128`

The number of dots in the x direction.

dots_y:

Single value, type: `int`. Default: `32`

The number of dots in the y direction.

blur:

Single value, type: `float`. Default: `0.1`

This is the radius of the "glow" of the pixels. This is expressed as a decimal relative to the size of the pixels. The default is `0.1` which means there's a 10% glow radius.

dot_size:

Single value, type: `float`. Default: `0.7`

The size of the individual "dots", expressed as a decimal relative to what their full size would be. A value of `1.0` will mean that each pixel will fill 100% of the space (e.g. no space in between), and it won't really look like separate pixels.

background_color:

Single value, type: `kivycolor`. Default: `191919ff`

The background color of the display (the color of the pixels when they're "off"). Note: this is a color with alpha channel value.

Settings for *flip_vertical* effect:

The `flip_vertical` effect does not have any settings.

Settings for *gain* effect:

gain:

Single value, type: `float`. Default: `1.0`

A numeric multiplier that will be applied to every color channel of every pixel in the display widget (brightness).

For example, if you set `gain: 1.2`, then a pixel on this display that has a color of (100, 100, 100) will be drawn with the color (120, 120, 120). (Each element multiplied by 1.2). Note that values above 255 will be capped at 255.

The default is 1.0 which means that the original colors are unchanged. You can play with this to act as a "poor man's" brightness control, but values too far above or below 1.0 will probably look weird.

Settings for *gamma* effect:

gamma:

Single value, type: `float`. Default: `1.0`

Sets the gamma factor of the effect.

Settings for *glow* effect:

blur_size:

Single value, type: `float`. Default: `0.5`

The blur width in pixels

intensity:

Single value, type: `float`. Default: `0.5`

The base intensity of the glow effect

glow_amplitude:

Single value, type: `float`. Default: `1.0`

The amplitude of the pulsing glow. Set to 0 if you want to disable the pulse.

glow_speed:

Single value, type: `float`. Default: `1.0`

The frequency of the glow effect in Hz.

Settings for *horizontal_blur* effect:

size:

Single value, type: `float`. Default: `4.0`

The blur width in pixels.

Settings for *invert_colors* effect:

The `invert_colors` effect does not have any settings.

Settings for *linear_gradient* effect:

color_stops:

Dictionary, type: `float:kivycolor`. Default: `None`

This defines the colors along the gradient. Each key is a floating point number in the range `[0.0, 1.0]` (inclusive) representing the position along the gradient while the value is the RGBA color at that position. Key values must be written in double quotes, for example: `"0.1": ff00ff00`

angle:

Single value, type: `float`. Default: `0.0`

The angle of the gradient in degrees. A value of `0.0` produces a horizontal gradient with the first color stop on the right while a value of `90.0` produces a vertical gradient with the first stop on the top and so on.

Settings for *monochrome* effect:

luminosity:

List, type: `float`. Default `[.299, .587, .114]`

This defines the luminosity factor for each color channel. The value for each channel must be between 0.0 and 1.0.

Settings for *pixelate* effect:

pixel_size:

Single value, type: `int`. Default: `10`

Sets the size of a new 'pixel' in the effect, in terms of number of 'real' pixels.

Settings for *reduce* effect:

shades:

Single value, type: `int`. Default: `16`

This is the number of shades each color channel will be reduced to. Note that this setting can produce weird results depending on your source content. If you want an old school look, you might have better luck creating your videos and graphics with fewer colors and then not setting the shades option here.

Settings for *scanlines* effect:

The `scanlines` effect does not have any settings.

Settings for *vertical_blur* effect:

size:

Single value, type: `float`. Default: `4.0`

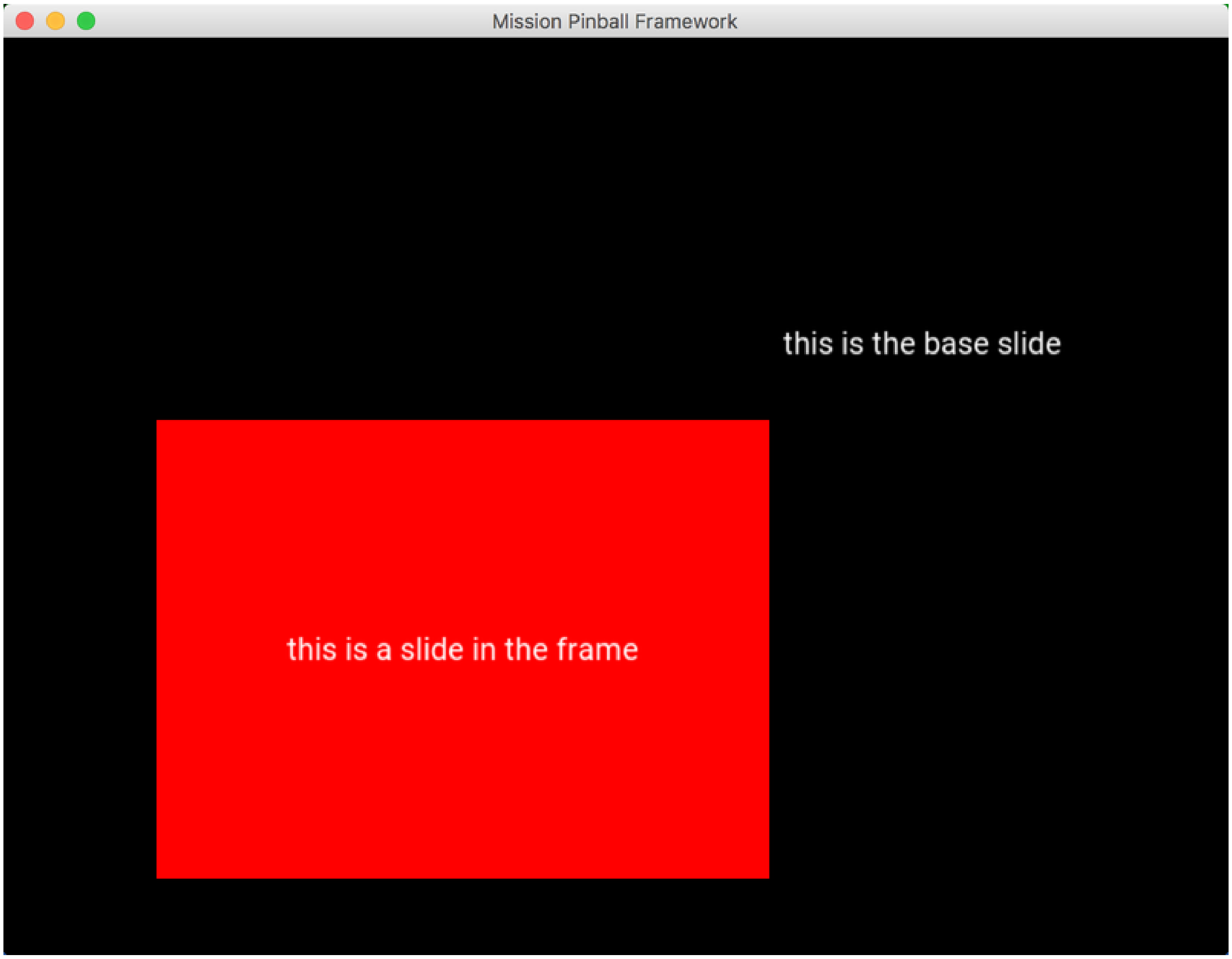
The blur width in pixels.

The display widget is used to show the contents of a display on a slide *slide* or another display (think of this like a picture-in-picture kind of thing). To attempt to clear up any confusion, there are two types of displays: a *display* which is basically an in-memory target for for slides and widgets, and a display widget (this help topic) which enables the graphical display of the previously mentioned type of display within its boundaries (it is the actual visual output of the logical display).

Here's an example:

```
#config_version=5
displays:
  window:
    height: 600
    width: 800
  my_frame:
    width: 400
    height: 300
    default: true

slides:
  base_slide:
    widgets:
      - type: display
        source_display: my_frame
        width: 400
        height: 300
        x: 300
        y: 200
      - type: text
        text: this is the base slide
        x: 600
        y: 400
  frame_slide:
    widgets:
      - type: text
        text: this is a slide in the frame
    background_color: red
slide_player:
  mc_ready.1:
    base_slide:
      target: window
  mc_ready.2: frame_slide
```

And the result:



Settings

```
type: display
source_display:
width:
height:
effects:
```

Note: Display widgets also have "common" widget settings for position, opacity, animations, color, style, etc. Those are not listed here, but are instead covered in *common widget settings* page.

Also remember that all widget settings can be controlled via *widget styles*, rather than you having to set every setting on every widget.

The following display widget settings may be *animated*: `x:` `y:`, and `pos:`.

source_display:

The name of the logical display to show on the screen within the widget boundaries. This name is available as a `target:` name is other areas of your configs when you want to target a slide the specified display.

More information on display targets is *here*.

width:

The width of the frame in pixels.

height:

The height of the frame in pixels.

effects:

A list of effects to apply to the display contents. These effects perform image processing to the source image and can be used to get an old school "DMD look" or "color DMD look" to your display as well as other special effects. For more information on effects, please review the *effects* documentation.

An example of a display widget with a dmd effect:

```
#config_version=5
slides:
  base_slide:
    - type: display
      source_display: dmd
      width: 640
      height: 160
      effects:
        - type: dmd
          dot_color: ff5500
```

Ellipse Widget

The ellipse widget is used to draw a solid ellipse (including circles) on a *slide*.

It can also be used to draw "wedges" (pie slices) or ellipses with sections missing (like Pac Man).

Note that ellipses are always solid. If you want an elliptical outline, use the *Bezier Curve Widget*.

Here's an example:

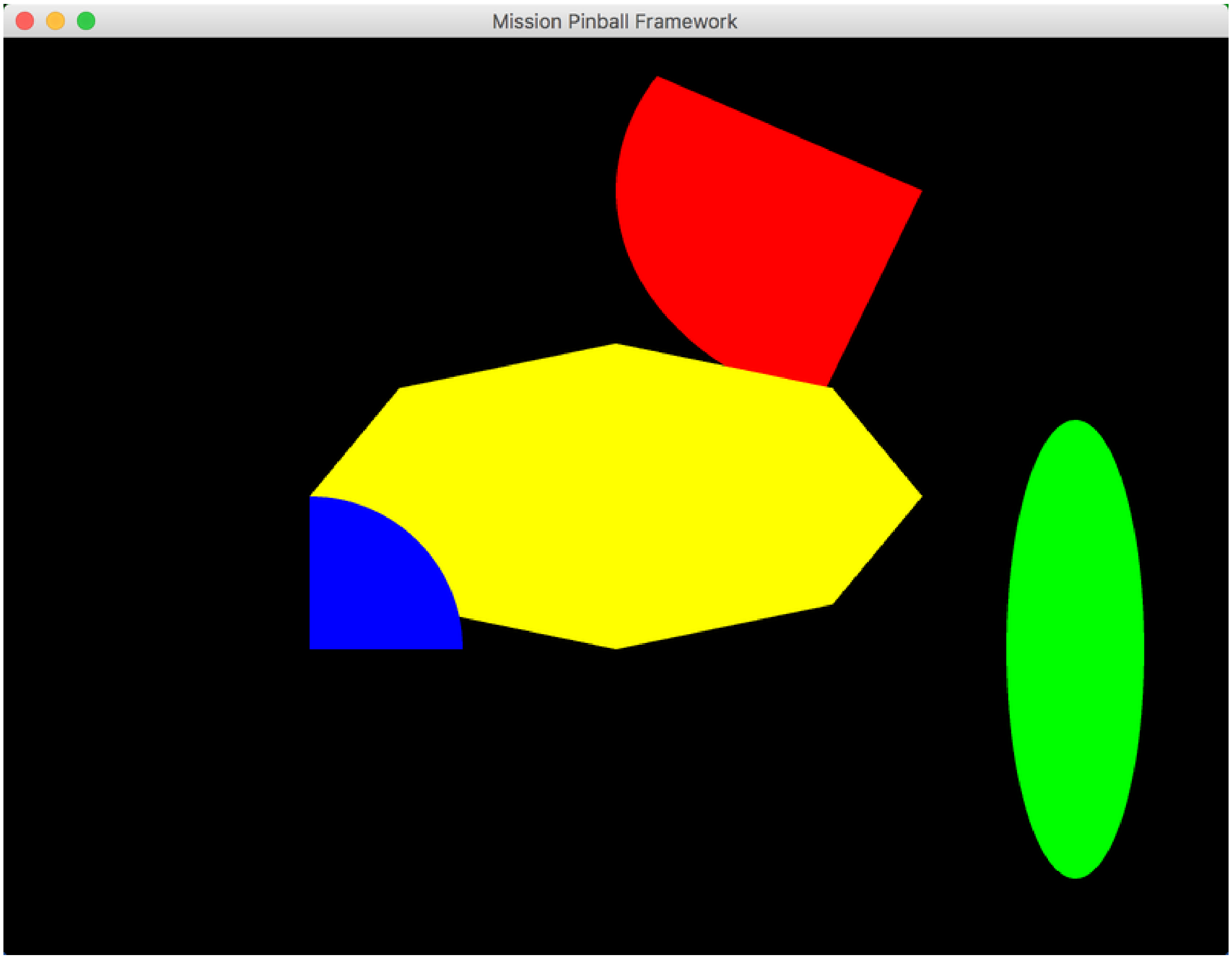
```
#config_version=5
slide_player:
  mc_ready:
    ellipse_example:
      - type: ellipse
        x: 200
```

(continues on next page)

(continued from previous page)

```
        y: 200
        width: 200
        height: 200
        color: blue
        angle_start: 0
        angle_end: 90
      - type: ellipse
        x: 400
        y: 300
        width: 400
        height: 200
        color: yellow
        segments: 8
      - type: ellipse
        x: 600
        y: 500
        width: 400
        height: 300
        color: red
        angle_start: 200
        angle_end: 300
      - type: ellipse
        x: 700
        y: 200
        width: 90
        height: 300
        color: lime
```

And the result:



Settings

```
width:
height:
segments:
angle_start:
angle_end:
```

Note: Ellipse widgets also have "common" widget settings for position, opacity, animations, color, style, etc. Those are not listed here, but are instead covered in *common widget settings* page.

Also remember that all widget settings can be controlled via *widget styles*, rather than you having to set every setting on every widget.

The following ellipse widget settings may be *animated*: `x:`, `y:`, `width:`, `position:`, `height:`, `size:`, `color:`, `angle_start:`, `angle_end:`, `opacity:`, `rotation:`, and `scale:`.

type: ellipse

Tells MPF that this is an ellipse widget. This setting is required when using ellipse widgets.

width:

The width (in pixels) of this ellipse. This setting is required.

The `width:` and `height:` settings set the bounding box that the ellipse will be drawn in. If you want a circle, set the width and height to be the same.

height:

The height (in pixels) of this ellipse. This setting is required.

segments:

The number of segments that will make up the ellipse. More segments will create a smoother edge, but depending on the size of your display and the size of the ellipse, you might not see much of a difference.

The default is `180`.

angle_start:

The angle, between 0-360, where the ellipse will start. The default is `0`.

angle_end:

The angle, between 0-360, where the ellipse will start. The default is `360`.

Note that a start angle of 0 and an end angle of 360 will create a complete solid ellipse.

Line Widget

The line widget is used to draw a straight line on a *slide*. (Note that if you want to draw a curved line, you can use the *Bezier Curve Widget*.)

Here's an example:

```
#config_version=5
slide_player:
  mc_ready:
    line_example:
      - type: line
        points: 0, 300, 800, 300
      - type: line
        points: 0, 100, 800, 100
```

(continues on next page)

(continued from previous page)

```
      - type: line
        points: 400, 95, 400, 0
        color: red
        thickness: 5
        cap: square
      - type: line
        points: 100, 500, 150, 550, 200, 450
        color: lime
        thickness: 2
      - type: line
        points: 500, 150, 600, 350, 650, 200
        color: blue
        close: true
        thickness: 3
```

And the results:

Settings

```
type: line
points:
thickness:
cap:
joint:
cap_precision:
joint_precision:
close:
```

Note: Line widgets also have "common" widget settings for position, opacity, animations, color, style, etc. Those are not listed here, but are instead covered in *common widget settings* page.

Also remember that all widget settings can be controlled via *widget styles*, rather than you having to set every setting on every widget.

The following line widget settings may be *animated*: `color:`, `thickness:`, `opacity:`, `points:`, `rotation:`, and `scale:`.

type: line

Tells MPF that this is a line widget. This setting is required when using line curve widgets.

points:

A list of point pairs which make up the line, expressed in x/y pairs (so the number of items here has to be even).

For example:

```
points: 10, 10, 200, 50, 300, 200
```

This would draw a line starting at (10,10) and going to (200, 50), and then from there, going to (300,200). If you just want a single straight line, then you would enter 4 values here: the x/y of the start and the x/y of the end.

thickness:

The thickness of the line. You'll probably have to play with different settings to get it right. The default is `1.0`, so `2.0` is twice as thick as the default, `0.5` is half as thick, etc.

cap:

Determine the cap of the line, defaults to 'round'. Can be one of 'none', 'square' or 'round'.

joint:

Determine the join of the line, defaults to 'round'. Can be one of 'none', 'round', 'bevel', 'miter'.

cap_precision:

Integer, defaults to 10.

Number of segments for drawing the "round" joint, defaults to 10. The joint_precision must be at least 1.

joint_precision:

Integer, defaults to 10.

Number of segments for drawing the "round" joint, defaults to 10. The joint_precision must be at least 1.

close:

Boolean (True/False), default is `False`.

If `True`, the line will be closed.

Examples

The example config files section of the documentation contains *examples of line widgets*.

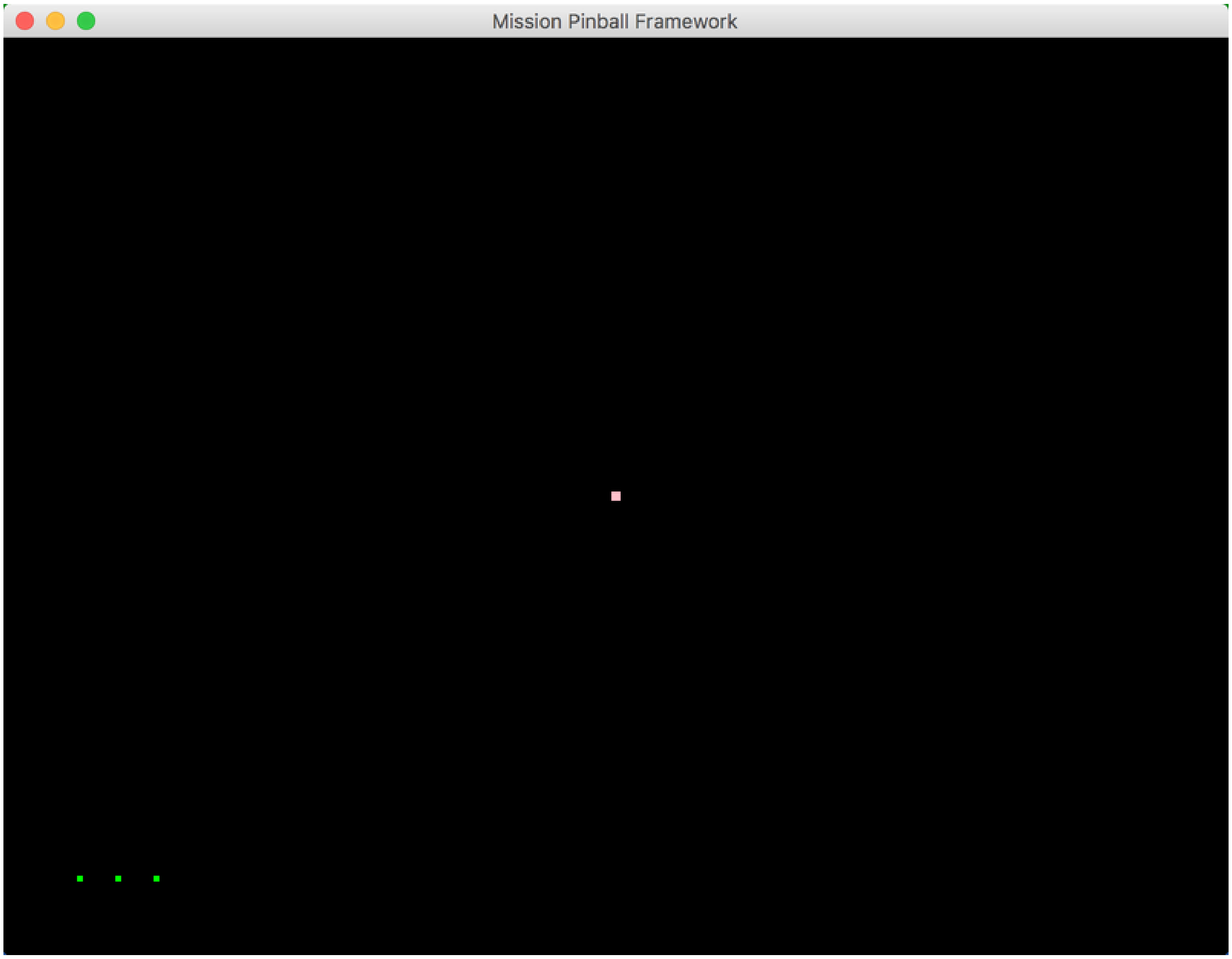
Points Widget

The points widget is used to draw points (individual square points) on a slide.

Here's an example:

```
#config_version=5
slide_player:
  mc_ready:
    points_example:
      - type: points
        points: 50, 50, 75, 50, 100, 50
        pointsize: 2
        color: lime
      - type: points
        points: 400, 300
        pointsize: 3
        color: pink
```

Which results in the following:



Settings

```
type: points
points:
pointsize:
```

Note: Points widgets also have "common" widget settings for position, opacity, animations, color, style, etc. Those are not listed here, but are instead covered in *common widget settings* page.

Also remember that all widget settings can be controlled via *widget styles*, rather than you having to set every setting on every widget.

The following points widget settings may be *animated*: `color:`, `points:`, `pointsize:`, `opacity:`, `rotation:`, and `scale:`.

type: points

points:

A list of the x,y coordinates of pairs of points.

pointsize:

Floating-point number, default is `1.0`.

The distance from the center of the point to the edge, so a value of 1.0 makes a point that's two pixels wide. (This is kind of like the radius, though points are square so it's not technically the radius. Probably there's some fancy math name for it.)

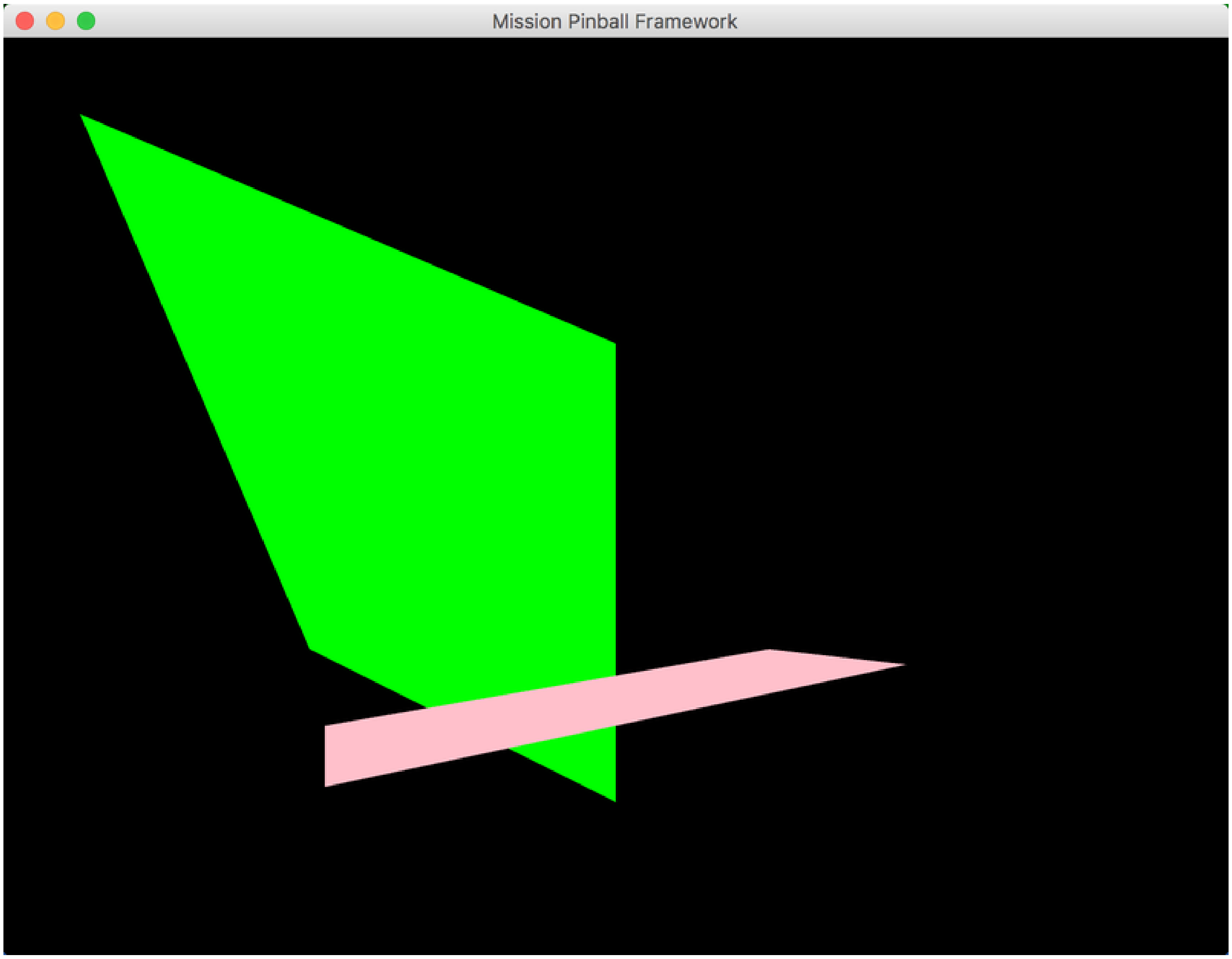
Quad Widget

The quad widget is used to draw solid polygons on a slide.

Here's an example:

```
#config_version=5
slide_player:
  mc_ready:
    bezier_example:
      - type: quad
        points: 210, 110, 210, 150, 500, 200, 590, 190
        color: pink
      - type: quad
        points: 50, 550, 400, 400, 400, 100, 200, 200
        color: lime
```

Which results in the following:



Settings

```
type: quad
points:
```

Note: Quad widgets also have "common" widget settings for position, opacity, animations, color, style, etc. Those are not listed here, but are instead covered in *common widget settings* page.

Also remember that all widget settings can be controlled via *widget styles*, rather than you having to set every setting on every widget.

The following quad widget settings may be *animated*: `color:`, `points:`, `opacity:`, `rotation:`, and `scale:`.

type: quad

Tells MPF this is a quad widget.

points:

A list of 8 values representing x,y coordinate pairs for the four corners of the quad.

A list of the x,y coordinates of the corners. Note that to have a normal four-cornered shape, the corners need to be in order. You can start with any one and go clockwise or counter-clockwise, but if you enter the corners in a mixed order like 1, 3, 2, 4 then it's possible your quad will fold over itself and look weird.

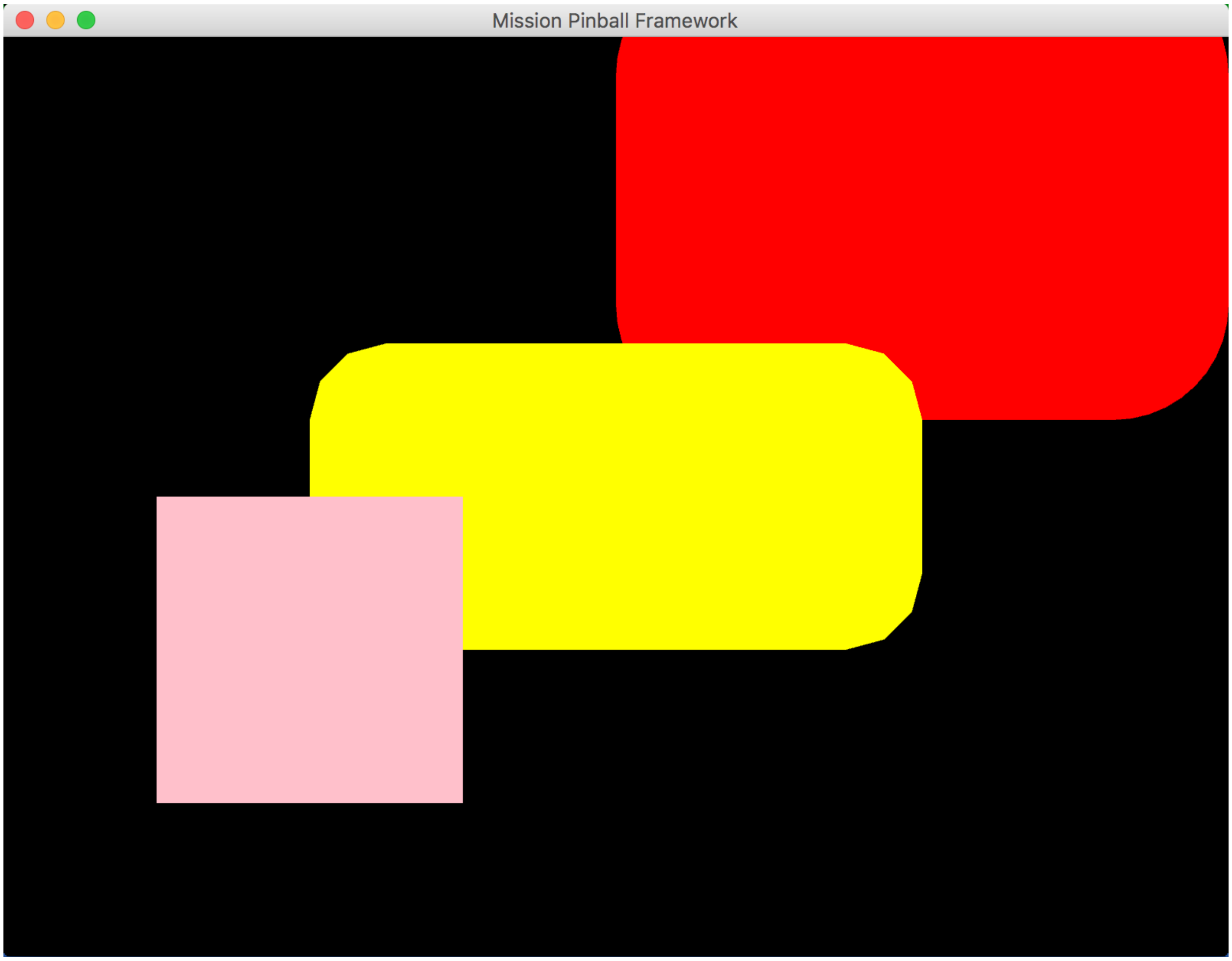
Rectangle Widget

The rectangle widget is used to draw a rectangle (or rounded rectangle) on a slide. Remember that a square is just a rectangle whose height and width are the same.

Here's an example:

```
#config_version=5
slide_player:
  mc_ready:
    rectangle_example:
      - type: rectangle
        x: 200
        y: 200
        width: 200
        height: 200
        color: pink
      - type: rectangle
        x: 400
        y: 300
        width: 400
        height: 200
        corner_radius: 50
        corner_segments: 3
        color: yellow
      - type: rectangle
        x: 600
        y: 500
        width: 400
        height: 300
        corner_radius: 75
        color: red
```

Which results in the following:



Settings

```
type: rectangle
width:
height:
corner_radius:
corner_segments:
```

Note: Rectangle widgets also have "common" widget settings for position, opacity, animations, color, style, etc. Those are not listed here, but are instead covered in *common widget settings* page.

Also remember that all widget settings can be controlled via *widget styles*, rather than you having to set every setting on every widget.

The following rectangle widget settings may be *animated*: `x:`, `y:`, `width:`, `height:`, `color:`, `corner_radius:`, `opacity:`, `rotation:`, and `scale:`.

width:

The width of the rectangle, in pixels.

height:

The height of the rectangle, in pixels.

corner_radius:

Number value of the radius of the corners (in pixels). Default is `0` which means sharp square corners.

corner_segments:

For rectangles with rounded corners (where `corner_radius:` is greater than 1), how many individual segments should make up the corner. The more segments, the smoother the corner is.

Default is `10`.

Segment Display Emulator widget

The segment display emulator widget is used to emulate hardware segment displays on a *slide*.

Here's an example:

```
#config_version=5
slide_player:
  mc_ready:
    display_slide:
      - type: segment_display_emulator
        name: display1
        character_count: 7
        character_slant_angle: 0
        character_spacing: 20
        segment_width: 0.11
        segment_interval: 0.04
        segment_off_color: 4b4c4a30
        segment_on_color: fe961bff
        side_bevel_enabled: true
        dot_enabled: true
        comma_enabled: true
        text: "HELLO"
        width: 600
        height: 150
        y: 100
```

The example above results in the following:

Settings

```
type: segment_display_emulator
name:
text:
flash_mode:
flash_frequency:
flash_mask:
display_type:
character_count:
character_spacing:
character_slant_angle:
padding:
background_color:
segment_off_color:
segment_on_color:
segment_width:
segment_interval:
bevel_width:
side_bevel_enabled:
dot_enabled:
comma_enabled:
character_map:
width:
height:
rotation:
scale:
```

Note: Segment Display Emulator widgets also have "common" widget settings for position, opacity, animations, color, style, etc. Those are not listed here, but are instead covered in *common widget settings* page.

Also remember that all widget settings can be controlled via *widget styles*, rather than you having to set every setting on every widget.

The following segment display emulator widget settings may be *animated*: `x:`, `y:`, `width:`, `height:`, `segment_on_color:`, `opacity:`, `rotation:`, and `scale:`.

type: segment_display_emulator

Tells MPF that this is a segment display emulator widget.

name:

The segment display name. This value is used to uniquely identify the segment display emulator widget when updating it using the *Segment Display player* in MPF. The value here must match the name assigned in the *segment_displays:* device section of your config. This value is only required if you wish to control the segment display emulator widget with the *Segment Display player*.

width:

The width of the segment display emulator widget (in pixels). This value is required.

height:

The height of the segment display emulator widget (in pixels). This value is required.

text:

The text characters to display in the widget. This value is required. If you don't want an initial text value, use "".

flash_mode:

The current display flash mode. Options include:

`off` The segment display does not flash (flashing is off). This is the default.

`all` All characters in the display will flash.

`match` Only the last two characters in the display will flash.

`mask` The `flash_mask` parameter determines which characters in the display will flash.

flash_frequency:

The number of times per second the display should flash. The default is `1.0`.

flash_mask:

Contains the flash mask string to use when flashing in mask mode. Each character of the flash mask string represents a character in the display. Character positions with an `F` character will be flashed while any other character will not flash. The default is `None` (no characters will flash). As an example, `FFFFFFFF________` will flash the first 8 character positions of a 16 character display which the last 8 characters will not flash. Note the `_` character could be replaced with any other character (other than `F`). You can use whatever character you wish for the non-flashing character positions.

display_type:

The type of display (7 segment, 14 segment). Options include:

7seg The segment display emulates a 7-segment display.

14seg The segment display emulates a 14-segment display. This is the default value.

character_count:

The number of character positions in the widget. The size of each character is determined by the widget size and the width is divided by the character count to get the character width.

character_spacing:

The space between each character/element (in pixels). The default value is `10`.

character_slant_angle:

The angle at which the characters are slanted (degrees from vertical). The default value is `0`.

padding:

The padding (empty space) around the display (in pixels). The default value is `20`.

background_color:

The background color of the display widget, in rgba format. The default value is `000000ff` (black).

segment_off_color:

The color of a segment that is off, in rgba format. The default value is `4b4c4aff` (gray).

segment_on_color:

The color of a segment that is on (active) for each character in the display in rgba format. If a single color is supplied, all characters in the display will be set to that color. This parameter can be animated and also controlled using the *Segment Display player*.See *Specifying Colors in Config Files* for more information on specifying colors in config files.

segment_width:

Width of each segment (as a decimal percentage of character width). The default value is `0.16` (16%).

segment_interval:

Spacing between segments (as a decimal percentage of character width). The default value is `0.05` (5%).

bevel_width:

Size of segment bevels (as a decimal percentage of character width). The default value is `0.06` (6%).

side_bevel_enabled:

Determines if the sides of each character should be beveled (`true` or `false`). The default value is `true`.

dot_enabled:

Determines if an integrated dot/period should be displayed in each character (`true` or `false`). The default value is `false`. When this is enabled, dot/period characters in the current `text` parameter value will be combined with the character immediately prior to the dot/period character and the dot segment will be on for that character (the dot will not use it's own character position in the display).

comma_enabled:

Determines if an integrated comma should be displayed in each character (`true` or `false`). The default value is `false`. When this is enabled, comma characters in the current `text` parameter value will be combined with the character immediately prior to the comma character and the comma segment will be on for that character (the comma will not use it's own character position in the display).

character_map:

The `character_map` parameter allows custom character segment mappings (which segments are on/off for each text character sent to the display). This advanced feature is useful for creating your own special characters or simply overriding the default mappings for any individual character. For more information on segment display character mappings, see *David Madison's Segmented LED Display - ASCII Library page <https://github.com/dmadison/LED-Segment-ASCII>*. This parameter is a dictionary with integer keys and values (key is the ascii character ordinal number, value is the segment bit mapping as an integer).

How to setup and use the virtual segment display emulator

This guide explains the basic steps to setup the virtual segment display emulator for your machine. Support for the visual component of the virtual segment display emulator is part of the MPF media controller and only available if you're using MPF-MC for your media controller.

1. Add your main display to your MPF config

Add the segment display to your list of displays in your machine-wide config file:

```
displays:
  window:
    width: 600
    height: 200
```

The example above contains a single display named "window" and has a size of 600x200. This will be the display that shows up on the computer screen.

2. Add your window configuration

The `window:` section of the machine-wide config holds the settings for the on-screen display window. If you don't have this section, add it now.

You can make the width and height anything you want. In this case we're just configuring it to be 600x200 with a window title of "Mission Pinball Framework".

```
window:
  width: 600
  height: 200
  title: Mission Pinball Framework
```

3. Configure a window slide to show the on screen segment display

Now we have a working on-screen window, but if you run `mpf both` now, your on screen window will be blank because we haven't built any slides to show up.

So in this step, we're going to build a slide for the on-screen window that will be shown when MPF starts. We'll add some widgets to that slide to make it look like a segment display.

First, create a `slides:` section in your machine config (if you don't have one already), and then create an entry for the slide that we want to show. In this case, we've decided to name that slide "window_slide_1". (Of course you can call this slide whatever you want.)

```
slides:
  window_slide_1:
```

Next we have to add some widgets to that slide. (Refer to the *documentation on widgets* if you're not familiar with widgets yet.)

The first widget will be a *segment display emulator widget* with a *glow effect* which is a widget which renders a emulation of a segment display:

```
slides:
  window_slide_1:
    widgets:
      - type: segment_display_emulator
        name: display1
        character_count: 7
        character_slant_angle: 0
```

(continues on next page)

(continued from previous page)

```
        character_spacing: 20
        segment_width: 0.11
        segment_interval: 0.04
        segment_off_color: 4b4c4a30
        segment_on_color: fe961bff
        side_bevel_enabled: true
        dot_enabled: true
        comma_enabled: true
        text: HELLO
        width: 600
        height: 150
        y: 100
```

As you can see there are a lot of configuration options to modify the rendering of the segment display segments/characters. This leads to a lot of very different looks for the resulting characters. One important item to note is the name parameter of the segment display emulator must match the name of the hardware segment display in MPF that we wish to connect to.

4. Configure the slide to show when MPF starts

Now we have a nice slide with the virtual segment display on it, but if you run MPF, you still won't see it because we didn't tell MPF to show that slide in the window. So that's what we're doing here:

```
slide_player:
  init_done:
    window_slide_1:
      target: window
```

If you don't have a slide_player: entry in your machine-wide config, go ahead and add it now. Then create an entry for the *init_done* event. This is the event that the media controller posts when it's ready to be used, so it's a good event for our use case.

Then under that event, create an entry to show the slide you just created in the previous step.

5. Configure your virtual segment display "hardware"

At this point you have a simple display configured, and you have default content showing up (the text "HELLO"). The final step is to add the configuration for your virtual segment display "hardware" so that MPF can control your segment display emulator as if it were a hardware display.

MPF contains a virtual hardware platform to allow it to run without physical hardware connected (*Using MPF without physical hardware*). This virtual platform contains code to allow it to communicate with segment display emulator widgets as if it were a real hardware display (in fact, you can develop your game using the virtual segment display and easily migrate it to actual hardware later with few configuration changes).

The first step is to create a *segment_displays:* entry in your machine wide config and add an entry for each segment display emulator widget (in this example we created a single widget so we will only need one entry).

```
segment_displays:
  display1:
    number: 1
```

A couple of things to note in the above configuration. `display1` is the name we are assigning to the segment display. This parameter value must match the one we assigned to the‘‘name‘‘ parameter of the segment_display_emulator widget when it was created on the slide previously (we used a value of `display1`). Be sure these values match or the communications between MPF and MPF-MC will not update the segment display widget properly.

Repeat this process for each segment display emulator widget you configure.

Now we need to let MPF know to send changes to the segment displays to the virtual segment display emulator in MPF-MC. This is accomplished using the *virtual_segment_display_connector:* plugin.

```
virtual_segment_display_connector:
  segment_displays: display1
```

The `segment_displays` parameter contains a list of all the segment display names you want to use in the connector to communicate with the segment display emulator widgets in MPF-MC.

6. Update your virtual segment display using the segment_display_player

Now that the virtual segment display is configured in the hardware section, it is time to configure the mechanism to update the text in the display. To do this, we use the *Segment Display player* (see also *segment_display_player:*).

```
segment_display_player:
  update_segment_display_hello:
    display1:
      text: "HELLO"
  update_segment_display_red:
    display1:
      action: set_color
      color: "FF0000"
  update_segment_display_score:
    display1:
      text: "{players[0].score:d}"
```

The segment display player establishes segment display updates that are triggered by events. In the above example, the `update_segment_display_hello` event sets the segment display text for display1 to `HELLO`. The `update_segment_display_red` event sets the segment display color to red for display1. Finally, the `update_segment_display_score` event sets the text to the score for player 1 (this will update automatically as the score changes using *Text Templates*).

Your virtual segment display should now be fully functional and ready for you to customize further for your specific project.

Text Input Widget

The text input widget is a special widget which lets the player use the flipper buttons to cycle through letters and numbers and to select them. This is used in the high score name entry and the service mode.

Currently the text input widget flashes a cursor over the selected letter, and the player hitting the flipper buttons changes the letter in place. In the future, we'll add an option to show all the letters on the screen in a long list as well.

Settings

Here are the list of settings you can use for text_input widgets:

```
type: text_input
key:
char_list:
max_chars:
initial_char:
keep_selected_char:
dynamic_x:
dynamic_x_pad:
shift_left_event:
shift_right_event:
select_event:
abort_event:
force_complete_event:
font_size:
font_name:
bold:
italic:
halign:
valign:
```

Note: Text widgets also have "common" widget settings for position, opacity, animations, color, style, etc. Those are not listed here, but are instead covered in *common widget settings* page.

Also remember that all widget settings can be controlled via *widget styles*, rather than you having to set every setting on every widget.

type: text_input

Tells MPF that this is a text_input widget. This setting is required when using text_input widgets.

key:

single

char_list:

String value, default is `ABCDEFGHIJKLMNOPQRSTUVWXYZ\_- \`.

A list of all the characters that are available to be chosen by the player as they're entering their name or initials. The order they are here is the order they show up as the uses scrolls left or right. If you want to add, remove, or change any of the defaults, just add a new `char_list:` setting to this text_input widget and completely replace the default list with your own list.

Note that "back" and "end" characters will automatically be added to the end of this list.

max_chars:

Integer value, default is `3`.

How many characters can be entered into this text input field.

initial_char:

Single character value. Default is `A`.

The character from your `char_list:` that you want to be the initial character selected before the player starts entering their name.

keep_selected_char:

Boolean (True/False or Yes/No), default is `True`.

When a player hits the start button to select a character and then the cursor moves to the next position, should the selected character stay with the character they just selected, or should it go back to the `initial_char:`?

dynamic_x:

Boolean (True/False or Yes/No), default is `True`.

If True, then the x position of this text widget will be updated as characters are selected and entered. If False, then the widget's x position will not change, and additional characters will be added to the right edge.

In other words, if you plan to center this widget, set this to True. If you plan on left justifying it, set it to False.

dynamic_x_pad:

Integer value. Default is `0`.

If you're using the `dynamic_x:` setting above, this is the number of additional pixels that will be added to the total width of the widget to calculate the dynamic x position.

block_events:

A list of events that, when posted, will prevent the text input from shifting or selecting input values. Useful for when a flipper cancel is used to select and the subsequent flipper inactive events shouldn't change the input.

Used in conjucting with `release_events` setting below.

release_events:

A list of events that, when posted, will unblock the text input from shifting or selecting input values.

Used in conjuction with `block_events` setting above.

shift_left_event:

The event that, when posted, will shift the selected character from the *char_list* to the left. Default is `sw_left_flipper`.

shift_right_event:

The event that, when posted, will shift the selected character from the *char_list* to the right. Default is `sw_right_flipper`.

select_event:

The event that, when posted, will select (or "enter") the currently highlighted character and move the cursor to the next position. Default is `sw_start` (which is the event that's posted when a switch tagged with *start* is hit).

abort_event:

The event that, when posted, will abort (or cancel) the character entry process. Default is `sw_esc` (which is the event that's posted when a switch tagged with *esc* is hit).

force_complete_event:

The event that, when posted, will mark the text entry process as complete, even if the player hasn't entered all their characters yet. Default is `None`.

font_size:

Same as the `font_size:` setting for the *Text Widget*. See that documentation for usage.

font_name:

Same as the `font_name:` setting for the *Text Widget*. See that documentation for usage.

bold:

Same as the `bold:` setting for the *Text Widget*. See that documentation for usage.

italic:

Same as the `italic:` setting for the *Text Widget*. See that documentation for usage.

halign:

Same as the `halign:` setting for the *Text Widget*. See that documentation for usage.

valign:

Same as the `valign:` setting for the *Text Widget*. See that documentation for usage.

anchor_y: baseline

Text input widgets have an additional `baseline` option in addition to the other baseline options detailed in the *common widget settings* documentation.

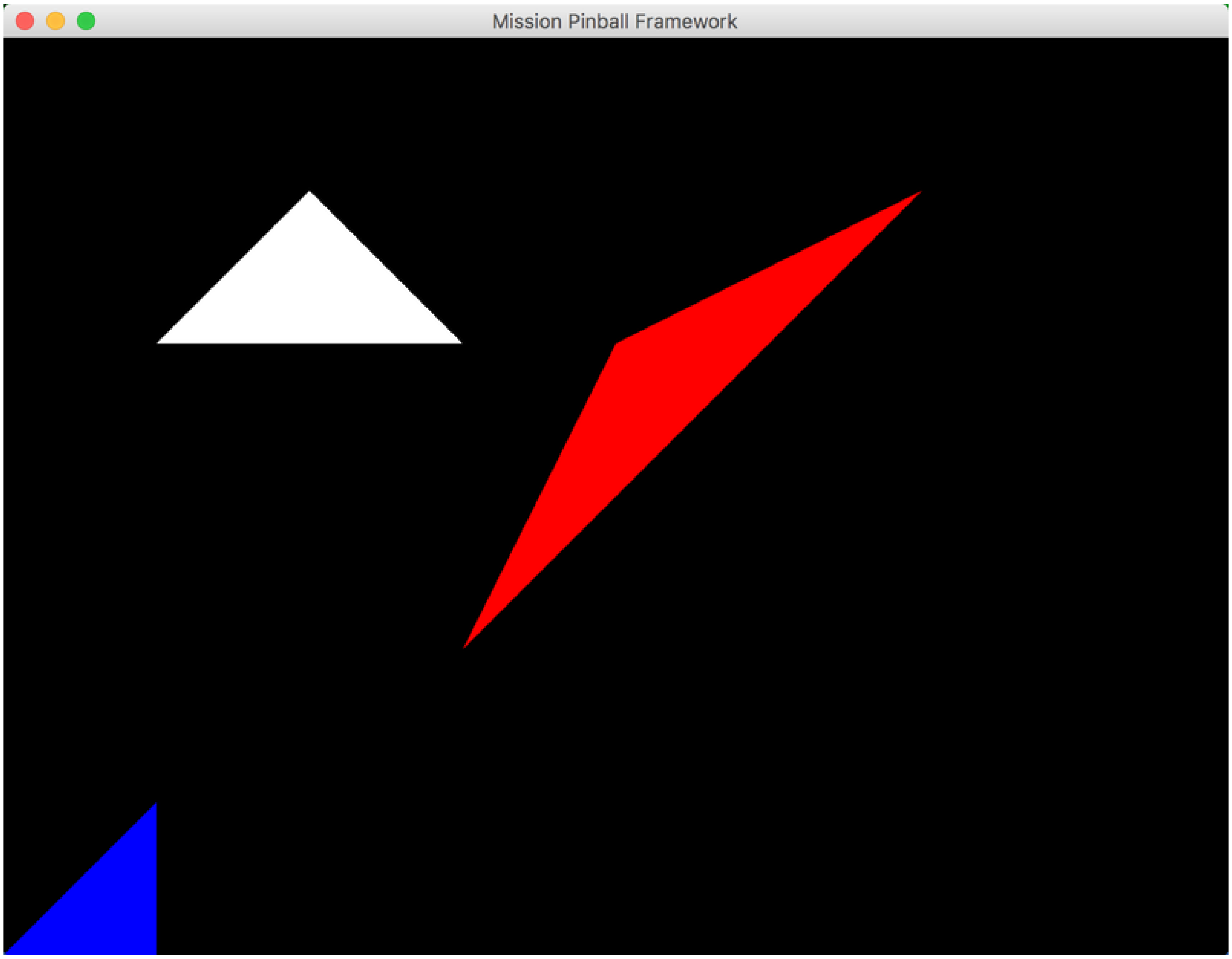
Triangle widget

The triangle widget is used to draw triangles on a *slide*.

Here's an example:

```
#config_version=5
slide_player:
  mc_ready:
    triangle_example:
      - type: triangle
        color: blue
        points: 0, 0, 100, 0, 100, 100
      - type: triangle
        points: 400, 400, 300, 200, 600, 500
        color: red
      - type: triangle
        points: 200, 500, 100, 400, 300, 400
```

The example above results in the following:



Settings

```
type: triangle
points:
```

Note: Triangle widgets also have "common" widget settings for position, opacity, animations, color, style, etc. Those are not listed here, but are instead covered in *common widget settings* page.

Also remember that all widget settings can be controlled via *widget styles*, rather than you having to set every setting on every widget.

The following triangle widget settings may be *animated*: `color:`, `points:`, `opacity:`, `rotation:`, and `scale:`.

type: triangle

Tells MPF that this is a triangle widget.

points:

A list of six numbers which are the the x,y coordinates for each of the three corners. For example, `points: 400, 300, 200, 300, 400, 200` would be a triangle with one corner at (400, 300), another corner at (200, 300), and the final corner at (400, 200).

Camera Widget

The camera widget is used to show live video from an attached camera a *slide*.

Here's an example:

```
#config_version=5
mpf-mc:
  widgets:
    camera: mpfmc.widgets.camera
slide_player:
  mc_ready:
    camera_example:
      - type: camera
        width: 800
        height: 600
```

Settings

```
type: camera
width:
height:
camera_index:
```

TODO

Common Settings that Apply to All Widget Types

The following settings are "common" settings that apply to all *types of widgets*:

```
type:
x:
y:
anchor_x:
anchor_y:
opacity:
z:
rotation:
animations:
reset_animations_events:
color:
style:
adjust_top:
adjust_bottom:
```

(continues on next page)

(continued from previous page)

```
adjust_left:
adjust_right:
expire:
key:
events_when_added:
events_when_removed:
```

type:

Specifies the type of widget, such as `type: text` or `type: image`. This setting is required (since MPF needs to know what kind of widget it is).

x:

The horizontal position of the widget on the slide. This setting can be entered in several ways:

- Absolute position: a number like `0`, `200`, or `-50`
- Relative position entered as a percent: `20%` or `-12%`
- A positional keyword: `left`, `center`, or `right`
- A combination of positional keyword and a value: `left+10%`, `right-5`

The default value is `center`.

See the *widget positioning* documentation for full details on how to position a widget on a slide.

y:

The vertical position of the widget on a slide. This setting can be entered in several ways:

- Absolute position: `0`, `200`
- Relative position entered as a percent: `20%`
- A positional keyword: `top`, `middle`, or `bottom`
- A combination of positional keyword and a value: `bottom+10%`, `top-5`

The default value is `middle`.

See the *widget positioning* documentation for full details on how to position a widget on a slide.

anchor_x:

The horizontal "anchor" point of the widget which specifies what point on the widget is used for the horizontal positioning. Valid options are `left`, `center` (or `middle`), and `right`.

The default value is `center`.

See the *widget positioning* documentation for full details on how to position a widget on a slide.

anchor_y:

The vertical "anchor" point of the widget which specifies what point on the widget is used for the vertical positioning. Valid options are `top`, `middle` (or `center`), and `bottom`.

The default value is `middle`.

See the *widget positioning* documentation for full details on how to position a widget on a slide.

opacity:

A value from 0 to 1 which controls the opacity (or transparency) of the widget. You can use decimal values between 0 and 1 for partial transparency.

- Completely transparent (e.g. invisible): `0`
- Completely opaque (e.g. normal): `1`
- 50% transparent: `0.5`

The default value is `1`.

Note that some widget types allow you to set values greater than 1, which will have the effect of making the "glow" of the widget brighter. This isn't a great effect, but it could be useful in some cases.

> **Caution:** Note that opacity values are 0 to 1, not 0 to 100. If you set `opacity: 100` then that's really like 10,000% opacity and your widget will probably look really weird.

z:

Specifies the "layer" or "z-order" of the widget. Higher z values mean that if parts of two widgets overlap on the slide, the one with the higher value will be drawn on top of the one with the lower value. (e.g. `z: 100` will be drawn on top of `z: 99`.)

The default drawing order of widgets is controlled by the order the widgets are listed in the slide, widget group, or widget_player config entry. So usually you don't need to manually set the z value, instead just put them in the order you want in your config.

However, being able to manually set the z value is nice if you want to add a widget to an existing slide and have it appear above and below certain widgets.

The default z value is `0`.

If you do want to add a widget with a particular z order to an existing slide, you'll probably have to set those existing widgets to a z value other than 0.

rotation:

Specifies the rotation of the widget. Values are entered in degrees. For example, 90 = one quarter rotation counter clockwise. May be used to rotate images and GIFs.

animations:

Contains a list of events and the animated widget properties and steps for each of those events. See the *widget animation documentation* for details.

reset_animations_events:

A list of events which are used to reset the widget to its original settings and stop all running animations. See the *widget animation documentation* for details.

Note that this seems like a grammatical error, since it's "animations events", but it's correct in this case because this setting is for a list of events that resets the widget animations (since animations themselves are a list of separate animations).

color:

Sets the color (and opacity) of the widget. This is pretty straightforward for most widget types (like text and the various shape widgets). If you set this for an image or video widget, it will have the effect of "tinting" the widget with the color you specified.

You can enter this as a hex color string or a color name. See the *color instructions* for details.

If you're entering hex strings, you can enter either 6 or 8 characters. The first six characters are RGB values (`00-ff` each), and the final is the opacity (`00-ff`). If you don't enter an opacity, `ff` (fully opaque) is used.

The default value is `ffffffff` which is white at 100% opacity.

style:

The name of the style (or styles) you want to apply to this widget. Note that styles must be previously defined somewhere in your config in order to use them. Also you can override any setting from the style by also manually including it in the widget config. See the *style documentation* for details.

New in MPF 0.51: Multiple style names can be provided for this setting, and the corresponding styles will be applied to the widget sequentially. As a result, individual style names cannot have spaces in them.

The default value is `None` which means no style is used.

adjust_top:

Redefines the top point of the widget when used in positioning to compensate for widgets that have visual top points that don't align with their technical top points.

The default value is `None`.

See the *widget positioning* documentation for full details on how widget positioning offset adjustments work.

adjust_bottom:

Redefines the bottom point of the widget when used in positioning to compensate for widgets that have visual bottom points that don't align with their technical bottom points.

The default value is `None`.

See the *widget positioning* documentation for full details on how widget positioning offset adjustments work.

adjust_left:

Redefines the left point of the widget when used in positioning to compensate for widgets that have visual left points that don't align with their technical left points.

The default value is `None`.

See the *widget positioning* documentation for full details on how widget positioning offset adjustments work.

adjust_right:

Redefines the right point of the widget when used in positioning to compensate for widgets that have visual right points that don't align with their technical right points.

The default value is `None`.

See the *widget positioning* documentation for full details on how widget positioning offset adjustments work.

expire:

Sets a time (such as `expire: 2s`) for this widget to be removed from the slide once it's added to it. This is useful with the widget_player when you want to add a widget to an existing slide and then remove it again.

The default value is `None`.

key:

Specifies a "key" name which is assigned to the widget which you can later use to target this widget if you want to do something to do (change a property, remove it, etc.) You don't need to specify keys for every widget—only for the ones that you want to target later.

See the *documentation on widget keys* for details.

events_when_added:

List of one (or more) values, each is a type: `string`. Default: `None`

A list of one or more names of events that MPF will post when this widget is added to a slide. Enter the list in the MPF config list format. These events are posted exactly as they're entered.

events_when_removed:

List of one (or more) values, each is a type: `string`. Default: `None`

A list of one or more names of events that MPF will post when this widget is removed from a slide (or when the slide it is on is removed). Enter the list in the MPF config list format. These events are posted exactly as they're entered.

Adding widgets to a slide

Now that you know what widgets are, it's time to look at how you can actually use them.

Option 1. Define widgets when you define a slide

The easiest way to create and use widgets is to include them in the slide configuration when the slide itself is created.

You can do this when you define a slide in the `slides:` section of your config, or when you show a slide in the `slide_player:` section of your config. See the *How to create slides* guide for details.

Option 2. Use the "widget player"

If you want to add a widget (or a groups of widgets) to an existing slide, you can use the *widget_player:*. You can define your widgets there, or you can use widgets that you've *pre-defined*.

How to position widgets on slides

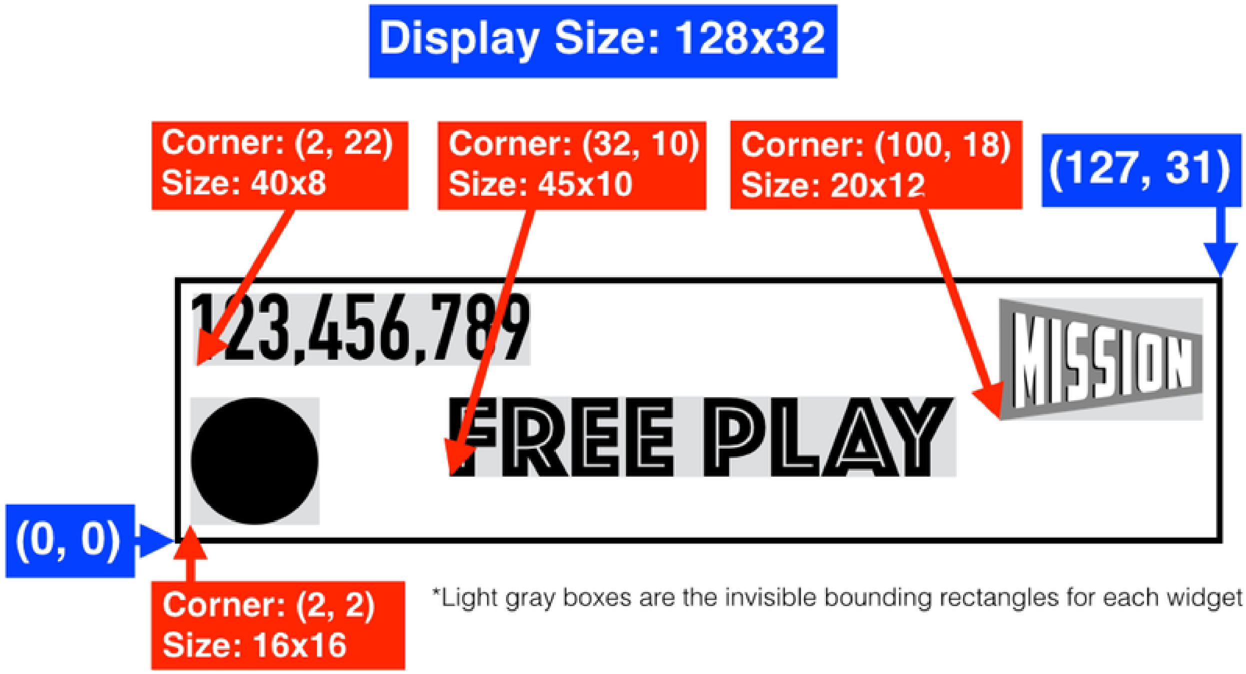
Probably the most important thing to know about putting widgets on slides is how to position them.

1. Understanding MPF display coordinates

At the most basic level, every display slide has a resolution (always conveyed in the order width, then height), and widgets have a position on slide (horizontal, then vertical).

- The dimensions of the slide are always described *width* (x), then *height* (y). (So a 128x32 display is 128 pixels wide and 32 pixels tall.)
- The "zero" position is the lower-left corner. (Just like an x-y cartesian coordinate graph from school.)
- Since the `(0, 0)` position is the actual location of the lower-left corner pixel, the upper-right pixel is actually one less than the width and height of your slide. (e.g. a display that's 128 pixels wide has x positions 0 through 127.
- A widget's position is always described *horizontal* (x), then *vertical* (y). So a widget at position `(10, 20)` is 10 pixels in from the left edge and 20 pixels up from the bottom.

Here's a simple example that illustrates this:



By the way, in MPF, the actual "pixel size" of the display as MPF sees it is separate from actual pixels of the physical display. So you could have a display in MPF that's 400x300 pixels, but you show that full size on an LCD that's 1920x1200 pixels. MPF will automatically scale the logical display to fit in the size of the window you configure on the physical display. This is known as "resolution independence", and is nice if you ever have to replace your LCD in the future and the new one you buy doesn't have the same resolution as your old one.

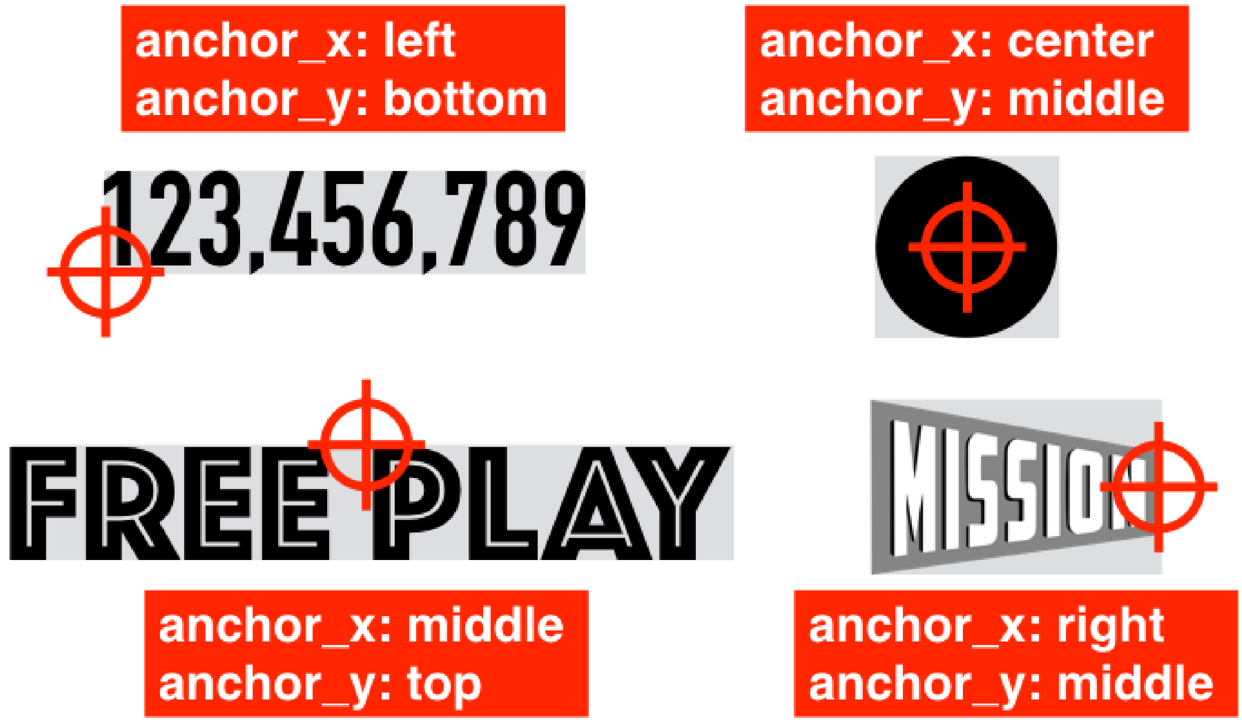
2. Understanding widget "anchors"

In the diagram from the first step, the "position" of each widget is set based on its lower-left corner. In real life, if you had to position every widget based on its lower-left corner all the time, you'd go crazy! For example, to "center" a widget, you'd have to calculate what the x and y offsets were and then do some math, and then if you animated the widget's size you'd have to recalculate it. . . it would be a mess!

Fortunately MPF does all this math for you.

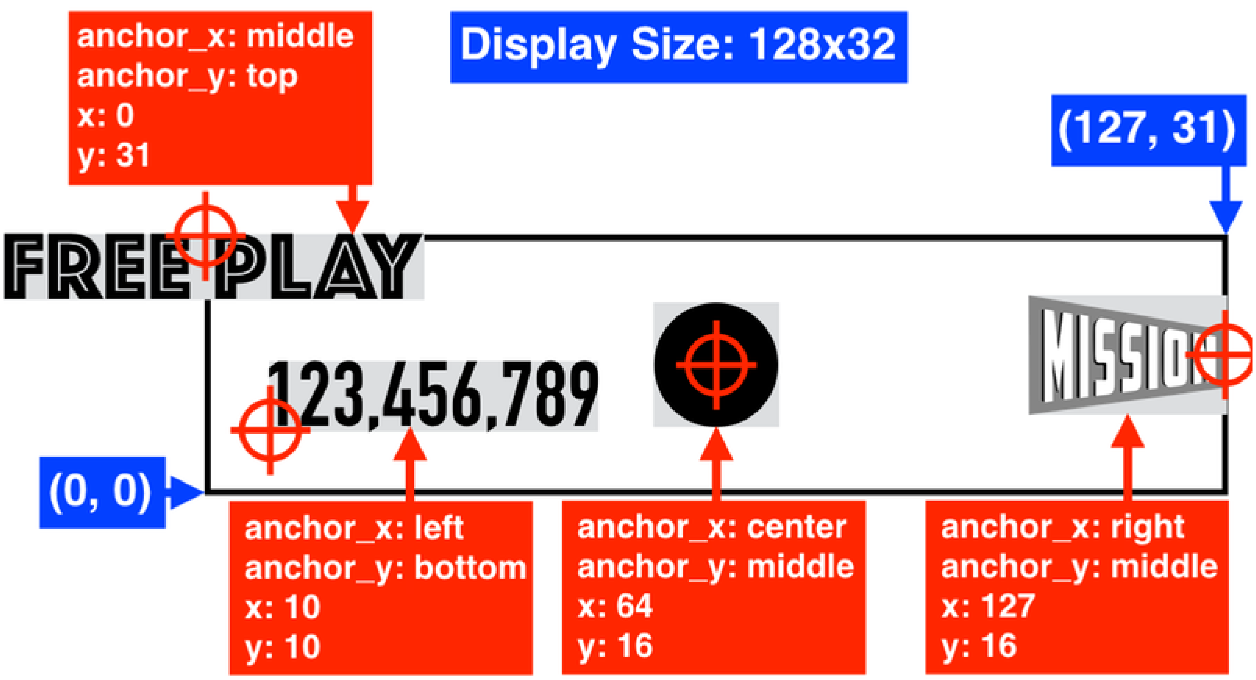
When you configure a widget in MPF, you can config its "anchor" point (both `anchor_x` for the horizontal anchor and `anchor_y` for the vertical anchor.)

A widget's anchor setting tells MPF what point on the widget is used to position it on the slide. Here are some examples which show how various anchor settings are applied to different widgets. The red bulls-eye target represents the point that's used by MPF to position that widget with each type of anchor settings.



3. Combing anchors and widget positioning

Now that you know how the coordinates and anchors work, let's look at some examples that combine these two concepts:



In the diagram above, you can see how the bulls-eye anchor target is the actual point of the widget that is positioned with each widget's `x:` and `y:` settings.

You'll also notice that widgets can be fully or partially be positioned outside the boundaries of a slide. (This is useful if you want to animate a widget "entering" the slide from off screen–you'd position the widget so it's outside the bounds of the visible window and then animate it moving on.) Also note that positioning can be negative. Negative x values are off the left edge of the slide, and negative y values are off the bottom.

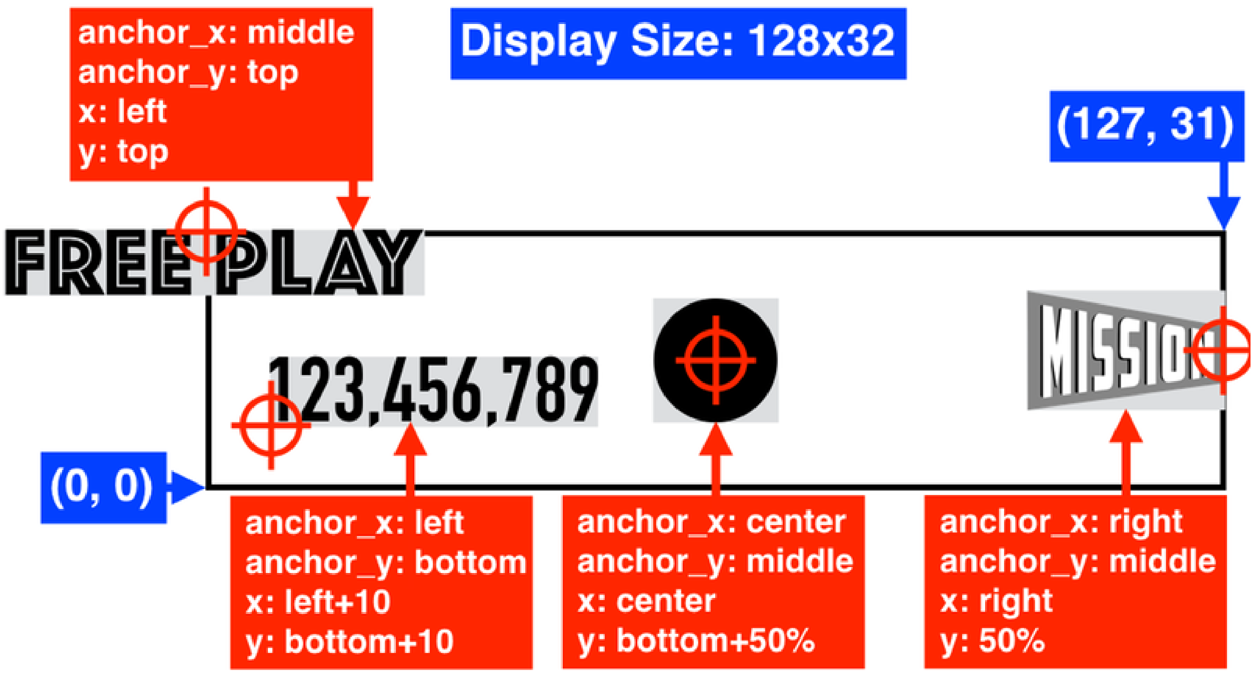
As you look at this example, you can probably start to see that different anchors make sense for different types of positioning. For example, if you have several widgets that you'd like to left-align, then it makes sense to set their anchors to `anchor_x: left` and positioning them based on their left edge.

By default, MPF uses the center of the widget for the anchor. This is what you get if you do not include an `anchor_x:` or `anchor_y:` setting. (Also the terms `middle` and `center` are interchangeable in all widget anchor and positioning settings.)

4. Relative positioning

Even though anchors are powerful, it can still be kind of confusing to position widgets based solely on `x:` and `y:` pixel values. After all, you constantly have to think about how big your display is and do lots of math to get your values set.

Fortunately MPF can use relative positions for a widget's `x:` and `y:` values, as show here:



There are a lot of different options in this diagram, so let's go through them one-by-one.

First, for `x:` values, you can use:

- `x: left` - Positions the anchor of the widget at the left edge of the slide
- `x: center` - Positions the anchor of the widget in the horizontal center
- `x: right` - Positions the anchor on the right edge

You can also use percentage values. The percentages are automatically calculated based on the width of the slide. So if you set `x:50%` and your slide is 800 pixels wide, the x value will be 400. (`x: 50%` is the same as `x: center`.)

For `y:` values, you can use:

- `y: top` - Positions the anchor of the widget at the top of the slide.
- `y: middle` - Positions the anchor of the widget in the vertical middle.
- `y: bottom` - Positions the anchor on the bottom edge.

Again, you can also use percentages.

What's really cool is you can *also* combine relative words with pixels and percentages. Some examples:

- `x: center+10` - Positions the x anchor of the widget 10 pixels to the right of the center position.
- `x: center-10` - Positions the anchor 10 pixels to the left of the center.
- `y: top-10%` - Positions the y anchor 10% below the top edge of the slide.

5. Try to use relative & percent positioning for everything

If you can manage to use relative (top/bottom/left/middle/etc.) and percentage values for everything, then your display system will be completely resolution independent!

Remember we said that the logical size of a display in MPF can be scaled up to any size physical display. So if you build your configs for a 1024x768 display, and then a few years down the line, you install a 1600x1200 monitor, you can make one simple config change to tell MPF to scale your 1024x768 up to the 1600x1200 display. That's fine, but you won't have a display that's as crisp as it can be because the graphics card will be scaling everything.

However, if you config all your widget positioning using only relative positions and percentages, then if you get a new display in the future, you can change the native logical resolution of your display in MPF and then make full use of the full resolution. It would be like everything instantly becoming high res!

6. Widget positioning offset adjustments

Another features of widget positioning in MPF is something known as an "offset adjustment". So far we saw how anchors can be positioned in the middle or an edge of the widget. The offset adjusts let you fine-tune the position of the anchor so it can be anywhere–including off the widget altogether!

Why would you want to do that? The main reason is that sometimes the technical edge of your widget is not exactly in the position that makes the most logical sense. A good example of this is text widgets. Many fonts have bounding boxes that are a few pixels bigger than the actual rendered text. For example, the text bounding box will allow for lower case letters that hang down below the baseline, but most pinball machines only use uppercase letters. This makes it hard to align the baseline of your font because there is random space under it:

Consider the following example where you want to align the bottom of the text with the bottom of the circle. The black areas represent the visible pixels, and the gray area is the actual widget bounding box. Even though this font is small (only 5 pixels tall, uses for small text on a DMD), it still has two blank rows of pixels below every letter. This means that if you set the `anchor_y: bottom` on both your text and the circle, they will not actually be aligned:

What's even worst is that this font only has 1 extra row on top, so if you want to center-align it with another widget you won't get the actual center of the visible text.

Fortunately MPF has a way to deal with this in the form of anchor adjustments. There are four adjustment values you can configure for a widget:

- `adjust_top`
- `adjust_right`
- `adjust_left`
- `adjust_bottom`

All of these settings are optional. (They all default to `0`.)

You might think it's weird that there are top, right, left, and bottom adjustments. Why not just have simple x and y adjustments? The reason is because having four is easiest when you're actually laying out your slides. For example, you might have a widget (like our text widget) with different amount of extra space on the top versus the bottom. So letting you specify an offset for the top and a separate offset for the bottom means that you can anchor and position that widget by either the top or the bottom and you don't have to mess with the adjusts each time. (It also means that center anchors will actually be in the visual center of the widget.) In other words, you set your adjustments once and never have to worry about them again.

For all the adjustments, positive values move the edge of the widget more towards the center (cutting off extra pixels), and negative values move it more away from the center (adding padding)

Going back to the example from before, if we add `adjust_bottom: 2`, that will move the adjustment point 2 pixels towards the middle, meaning our bottom alignment now actually aligns:

Negative values have the effect of adding padding to widgets, which can also be nice as you're aligning and distributing things.

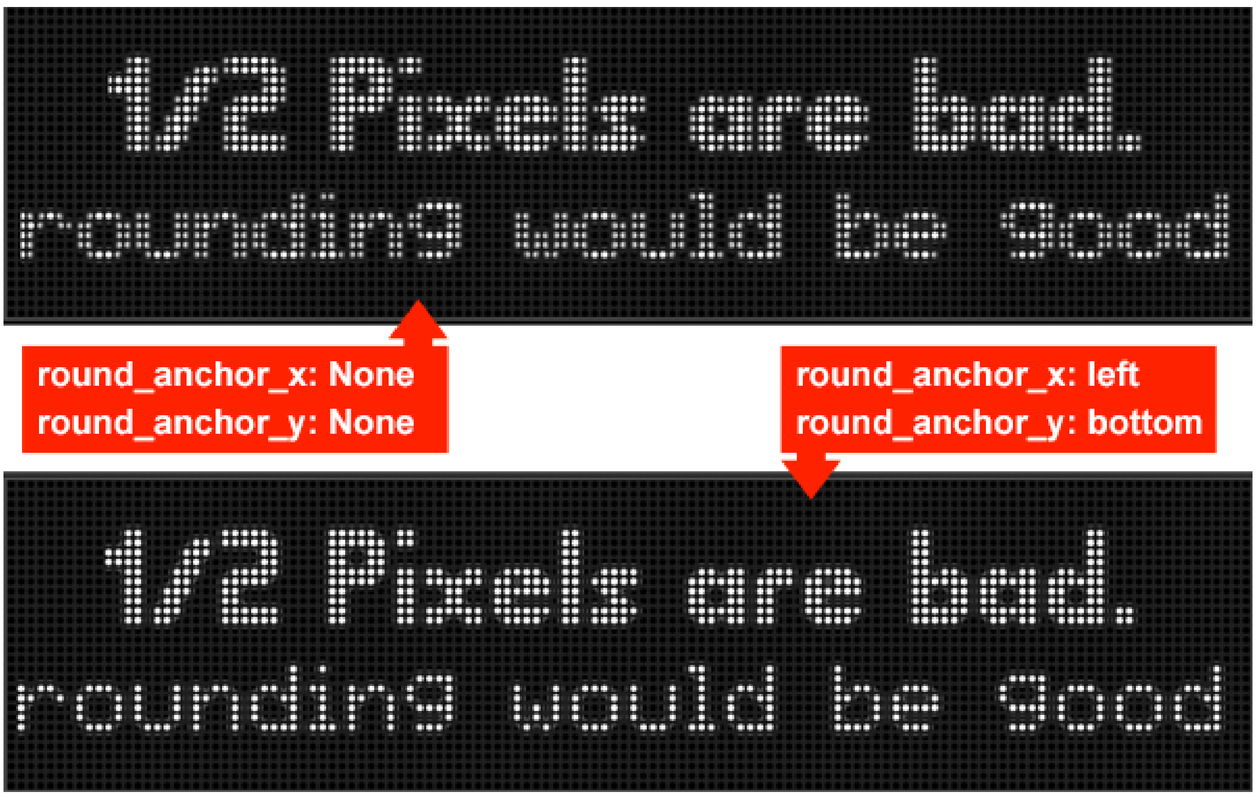
The only other thing to know about adjustments is that they only affect the positioning of the widget. Adjustments are not cropping, and they will not "cut off" or "trim" the widget.

7. Widget position rounding

Sometimes a center-anchored or percentage-based widget will end up at a position with a fractional pixel. High-resolution displays have no trouble smoothing out partial pixels, but low-resolution displays (like DMDs) may render the widget blurry.

You can prevent MPF-MC from positioning widgets on pixel fractions with the `round_anchor_x:` and `round_anchor_y:` setting, either locally on a widget or globally on the display. When present, this setting will force MPF-MC to round fractional anchor positions in the specified direction.

- `round_anchor_x: left` - Round the horizonal pixel position down
- `round_anchor_x: right` - Round the horizonal pixel position up
- `round_anchor_x: center` - Do not round the pixel position (default)
- `round_anchor_y: bottom` - Round the vertical pixel position down
- `round_anchor_y: top` - Round the vertical pixel position up
- `round_anchor_y: center` - Do not round the pixel position (default)



This setting is valid on `widgets` and `displays`. If you have a display and a widget both configured for rounding, the widget's setting will take priority.

8. Widget positioning can be done in styles

One of the powerful features of widgets in MPF is that you can configure widget styles, which are like buckets of settings that are applied and merged into widget settings. You can put any widget settings you want in a style (and then specify the style to be applied to a widget in the `style:` setting in a widget config, a slide config, a show, or a widget player).

Styles can be used in several different ways. For example, you can configure a style for text widgets which has the font name, font size, and adjustments so you can simply add `style: big` to a widget and everything will be there.

You can also put `x:` and `y:` settings in styles and use them to position and size the widgets on different parts of your display. For example, you might have an area of the screen that always shows some kind of status message, and even though that might be used throughout your game, you might always want the same font, alignment, size, and positioning no matter what's there. So you can define a style called `info_zone` and then any text widget that uses that style will always show up in the right place. (You can also use styles for z-order and animations, so you can use a style to define popups and other things that you'll use over and over.)

See the How To guide on widget styles for details.

9. Putting it all together

So now you've seen all the options for positioning and placement of widgets. But how do you actually use them? Simple. Everything discussed here are just regular widget settings. So you can use them in slides:

```
slides:
  slide1:
    widgets:
      - type: text
        text: MY WIDGET
        x: left+10%
        y: top-10%
        adjust_bottom: 2
```

You can use them in *named widgets*:

```
widgets:
  my_cool_widget:
    - type: text
      text: MY WIDGET
      x: left+10%
      y: top-10%
      adjust_bottom: 2
```

You can use them in the widget player:

```
widgets:
  my_widget:
    - type: text
      text: "MY WIDGET"
widget_player:
  some_event:
    my_widget:
      widget_settings:
        x: left+10%
        y: top-10%
        adjust_bottom: 2
```

And you can use them in shows:

```
# in your machine config
widgets:
  my_widget:
    - type: text
      text: "MY WIDGET"
##! show: test_show
# in your show
- duration: 1
  widgets:
    my_widget:
      widget_settings:
        x: right-15.4%
        y: top
```

How to animate display widgets

One of the features of MPF is that you can animate display widgets. Animating a widget means that you can change a widget's properties over time. You can pretty much change any numeric property, including size, position, opacity, etc.

When animating widgets, you specify multiple properties to change at the same time, or a sequence of changes one after the other (or both). You can also specify the duration of each step, the "easing" formula that affects the curve (acceleration/deceleration) of the change, and whether the animation is a one-time thing or a repeating loop.

You can also configure animations to start playing as soon as the widget is created, or tie steps (or series of steps) to MPF events, meaning a widget might be static, then the event "move_widget" is posted and it moves, then the event "remove_widget" is posted and it's animated away.

This How To guide will show you how to do all of that.

1. Understanding animations in MPF

MPF animations are properties of widgets. For example, here's a basic widget with no animations:

```
slides:
  slide_1:
    widgets:
      - type: text
        text: MY TEXT
        color: red
```

To add animations to a widget, you simply add an `animations:` setting to that widget, and then under there you add specific animation steps and settings. For example:

```
slides:
  slide_1:
    widgets:
      - type: text
        text: MY TEXT
        color: red
```

(continues on next page)

(continued from previous page)

```
        animations:
          show_slide:                   # animation trigger event
            - property: opacity         # name of the widget property we're animating
              value: 1                  # target value of that property for this step
              duration: .5s             # duration for this step (how long it takes to get there)
            - property: opacity         # second step in the animation (starts with a hyphen)
              value: 0
              duration: .5s
              repeat: true              # added to the final step, tells this animation to repeat (loop)
```

In the example above, an `animations:` setting has been added to the widget. Then under there, you add the name of the event you want to use to trigger this animation to start. In this case, we use a special event called `show_slide:` which means these animations are triggered when the slide is shown on a display.

Next, notice that under the event, there are two steps (each beginning with a hyphen and a space).

There are several settings you can specify in each step. (See the config file reference for animations for details)

In this example, there are three settings for the first step:

```
- property: opacity
  value: 1
  duration: .5s
```

The **property** setting is the name of the widget's property that you want to animate. This can be almost any numerical property of the widget, including `x:`, `y:`, `opacity`, etc. (Different widget types have different types of animatable properties. For example, on text widgets you can animate the `font_size:`, on various shape widgets you can animate the `height:`, `width:` and `rotation:`, etc.)

2. Relative animation values

Sometimes it is desirable to animate a value a relative amount from a widget's current value rather than specifying an absolute target value. This can be done using `relative: True`. With the `relative:` parameter set to `True`, the new target value will set by adding the `value:` parameter to the widget's current `property:` value when the animation starts. When `relative:` is set to `False`, the animation target uses the actual `value:` property value as its destination.

The following example animates a widget 50 pixels in the `x` direction over one second from its current locat
then -50 pixels in the `y` direction over another second followed by a 45 degree rotation over 500 ms:

```
- property: x
  value: 50
  relative: True
  duration: 1s
- property: y
  value: -50
  relative: True
  duration: 1s
- property: rotation
  value: 45
```

(continues on next page)

(continued from previous page)

```
relative: True
duration: 500ms
```

3. Animation trigger events

The animation trigger event (which is the `show_slide:` entry in the example from the previous step is the name of the MPF event you want to use to start the animation.

These are regular *MPF events* and can be anything—a shot being made, a switch hit, etc. (See the *event reference* for a full list of events.)

In most cases, however, you'll probably want to trigger an animation to start playing when the slide is created, so in addition to being able to use any MPF event, there are also a few special events (sometimes called "magic events") that have special meaning here:

add_to_slide:

This event is triggered when a widget is added to a slide. This is useful when you're using the *widget_player* to add to new widget to an existing slide, and you want an animation to be applied to that widget as soon as it's added.

remove_from_slide:

This event is triggered when a widget is is removed from a slide.

pre_show_slide:

This event is triggered when the slide this widget is part of is about to be shown. This doesn't necessarily get called when the slide is created or when the `slide_player:` event happens, because if the slide is not the highest priority slide, then the slide will be created but not shown. So this event happens right before the slide is shown.

If there's an entrance transition, this method is called BEFORE the transition starts. In other words, it means the animation will be playing as the slide transition is happening.

show_slide:

This event is triggered when the slide this widget is part of has been shown and is the current slide on the display. This doesn't necessarily get called when the slide is created or when the `slide_player:` event happens, because if the slide is not the highest priority slide, then the slide will be created but not shown. So this event happens right before the slide is shown.

If there's an entrance transition, this method is called AFTER the transition starts. In other words, it means the animation will NOT be playing as the slide transition is happening.

pre_slide_leave:

This event is triggered by the current slide that's being shown on a display is about to be replaced by another slide.

If there's an exit transition, this method is called BEFORE the transition starts. In other words, it means the animation will be playing as the slide transition is happening.

slide_leave:

This event is triggered by the current slide that's being shown on a display is has been replaced by another slide.

If there's an exit transition, this method is called AFTER the transition starts. In other words, it means the animation will be NOT playing as the slide transition is happening.

You might wonder what this is for, since what's the point of an animation if the slide is not showing? This is useful if you want to pause or reset an animation when the slide is not active. Then you can resume or restart the animation with the "pre_show_slide" or "show_slide" event when the slide is shown again.

slide_play:

This event is triggered when the slide this widget is part of is played as part of a `slide_player:` "play" command, either via a standalone slide player config or as a show step).

Other slide-related MPF events

In addition to the seven special-purpose animation trigger events listed above, there are three standard MPF events which are posted when slides are created, when they become active, and when they're removed. See the events reference for details on when these three events are posted.

- *slide_(slide_name)_created*
- *slide_(slide_name)_active*
- *slide_(slide_name)_removed*

4. Animating multiple properties at once

The example animation above includes two steps (one to set the opacity to 1 and the next to set it to 0). By default steps are sequential, meaning that one step completes before the next one starts. However you can add a `timing: with_previous` to an animation step which will make it so that step runs at the same time as the step before it. This means you can animate multiple properties at once.

For example, to make the text grow and shrink while also fading on and off:

```
slides:
  slide_1:
    widgets:
      - type: text
```

(continues on next page)

(continued from previous page)

```
        text: MY TEXT
        color: red
        font_size: 50
        animations:
          show_slide:
            - property: opacity
              value: 1
              duration: .5s
            - property: font_size
              value: 100
              timing: with_previous        # makes this step run at the same time as the previous one
              duration: .5s                # specify a duration for each step, even when with_previous
            - property: opacity
              value: 0
              duration: .5s
              repeat: true
            - property: font_size
              value: 50
              duration: .5s
```

Notice that the animation in the example above has 4 steps, but steps #2 and #4 have the setting `timing: with_previous`. You can chain together as many `with_previous` steps as you want. (The default setting for one step to run after the previous one is `timing: after_previous`, but since that's the default you don't need to explicitly add it.

Also note that all 4 steps above specify `duration: .5s`. However you can make each step a different amount of time. In fact you can even make multiple `with_previous` steps different durations (though the animation won't move on to the next `after_previous` step until all the simultaneous steps are complete).

By the way, the example above is a widget that's part of a slide, but remember you can add animations to widgets anywhere a widget is defined (in the slide properties, in a show step, as part of a *named widget*, as part of a `widget_settings:` override section in the `widget_player:`, etc.)

It is also possible to animate multiple properties in a single animation step by using a list in both the `property:` and `value:` parameters (there must be the same number of items in both lists). The following example moves a widget diagonally to the coordinate (10, 20) while rotating it 180 degrees over 5 seconds:

```
- property: x, y, rotation
  value: 10, 20, 180
  duration: 5s
```

5. Multi-step animations with different trigger events

So far all of the animation examples have been triggered on the `show_slide` event (which means they start animating as soon as the slide is shown).

You can create multiple event entries in the animation that cause different animations to take place when different events occur. You can mix and match these as much as you want, including mixing the "special" animation trigger events with regular MPF events.

```
slides:
  slide1:
    widgets:
      - type: text
        text: I'M GOING TO MOVE
        x: 50
        y: 50
    animations:
      move_up:
        property: y          # if there's just one animation step, we don't need the hyphen
        value: 100
      move_down:
        property: y
        value: 0
      move_right:
        property: x
        value: 100
      move_left:
        property: x
        value: 0
      move_home:
        - property: x
          value: 50
        - property: y
          value: 50
          timing: with_previous
```

In the above example, we have five different animation events configured. These are just regular MPF events which you can use from logic blocks, shots, switch events, etc. When the event `move_up` is posted, this widget will move to the top of the display (`x: 100`), when the `move_left` event is posted, it will move to the left of the screen, etc.

If `move_home` is posted, there are two steps in the animation which both run together to move the widget back to its initial position.

Again, you can use any combination of properties and any number of steps for each event.

You can also use a property from your event. For instance, you can move a widget based on a player variable:

```
slides:
  slide1:
    widgets:
      - type: text
        text: I'M GOING TO MOVE
        x: 50
        y: 50
    animations:
      player_loops:          # this is posted when the player variable "loops" changes
        property: y
        value: (value)       # this is the value of loops
```

6. Looping and repeating animations

So far, every animation sequence we've looked at will just run through once and then stop. However, you can add `repeat: true` (or `repeat: yes`) to the last step of an animation, and that will cause that animation to loop back to the beginning and keep repeating.

Of course you can mix-and-match repeating animations with one time animations. For example:

```
slides:
  slide1:
    widgets:
      - type: text
        text: BOO!
        y: -50
        font_size: 90
        animations:
          show_slide:
            property: y
            value: 50
            duration: 500ms
          pulse_boo:
            - property: font_size
              value: 100
              duration: 250ms
            - property: font_size
              value: 90
              duration: 250ms
              repeat: true
          bye_boo:
            - property: y
              value: 100
            - property: x
              value: 150
              timing: with_previous
```

In the example above, when the slide is shown (or when the widget is added if this config was in your `widgets:` section and you added it via a `widget_player:` entry), the widget will fly into the slide from the bottom (since the initial y value is -50, it will start off the screen). Then when the `pulse_boo` event is posted, the two-step animation which makes the font size bigger and smaller will starting playing and repeat forever. Finally when `bye_boo` is posted, the widget will fly off the screen to the upper right.

There is something special to consider when working with 360 degree rotations. Setting the property rotation to 360 results in a complete turn of a widget. However, attempting to repeat the 360 degree rotation appears to do nothing. This is because of how Kivi handles 0 and 360 degrees. When the rotation completes, the widget is at rotation 360 degrees. Repeating the step fails to cause the widget to move because it is already at destination. To create continuously rotating widgets, a two step process is required:

```
slides:
  slide1:
    widgets:
      - type: text
        text: I'M GOING TO ROTATE
        x: 50
        y: 50
```

(continues on next page)

(continued from previous page)

```
        animations:
          show_slide:
            - property: rotation
              value: 0
              relative: false
              duration: 0
            - property: rotation
              value: 360
              relative: false
              duration: 2s
              repeat: true
```

The first step creates a reset point setting the widget at 0 degrees. This assures that there is always a destination to rotate to which is provided by the following step.

7. Inserting a "pause"

Sometimes you might want to add a timed "pause" to an animation, where one step animates, then it pauses, then another step animates.

The easiest way to do that is just to add a step where the property value in the step is the same as whatever value that property is currently at. This is easy to do using a relative property value of 0 as shown in the following example. So you still have the step in the animation, it just isn't doing anything since the widget's property is already at the desired target value. For example:

```
slides:
  slide1:
    widgets:
      - type: image
        image: flying_toaster
        y: -50
    animations:
      show_slide:
        - property: y
          value: 50
          duration: 1s
        - property: y
          value: 0
          relative: true
          duration: 2s
        - property: y
          value: 200
```

The the example above, the `flying_toaster` image will move in from the bottom of the screen (to `y:50`) in 1 second, then pause for 2 seconds (since `y: 50` again), then move out of the top of the screen in 1 second.

8. Easing

You can also set "easing" values for each animation step which controls the formula that's used to interpolate the current value to the target value over time. The default is `linear` which just does a constant motion (no acceleration/deceleration) over time. Refer to the *Easing Instructions* for details on how this works and descriptions of all the options.

9. Creating reusable "named" animations

Much like *named widgets*, you can also create pre-defined animations that you can easily apply to any widget. You do this by adding those animations to the `animations:` section of your config, like this:

```
animations:
  fade_in:
    property: opacity
    value: 1
    duration: 1s
  fade_out:
    property: opacity
    value: 0
    duration: 1s
```

Now you can use these animations, by name, in any widget or widget_player config where you would ordinarily define your own animations.

For example, to configure a widget to fade in (assuming the widget was initially created with `opacity: 0`:

```
widgets:
  hello_widget:
    - type: text
      text: HELLO
      animations:
        show_slide: fade_in
```

Again remember this can be done anywhere you configure an animation. So if you later wanted to fade that text out when the event "timer_hurry_up_complete" is posted, you can do it like this:

```
widgets:
  hello_widget:
    - type: text
      text: HELLO
      animations:
        show_slide: fade_in
        timer_hurry_up_complete: fade_out
```

10. Chaining multiple named animations together

When working with named animations, you can chain together multiple named animations for a single event by specifying them as a list, like this:

```
widgets:
  hello_widget:
    - type: text
      text: HELLO
      animations:
        some_event: fade_in, fade_out, pulse
```

Any animation with `timing: with_previous` in the first step will run with the previous one, meaning you can create lots of little effects and sub-animations and then combine them in reusable ways throughout your config.

You can even use the same animation over and over in a sequence to repeat something a certain number of times. For example:

```
animations:
  pulse:
    - property: opacity
      value: 0
      duration: 100ms
    - property: opacity
      value: 1
      duration: 100ms
      timing: after_previous
widgets:
  widget1:
    - type: text
      text: HELLO
      animations:
        flash_3x: pulse, pulse, pulse
```

In the example above, when the MPF event "flash_3x" is posted, it will cause widget1 to pulse three times.

11. Animating a progress bar

MPF can also animate progress bars or similar things. In this example, we animate a progress bar based on the player variable `progress` by hooking the width of the bar to the event `player_progress` which is posted when the value changes:

```
slides:
  green_slide:
    widgets:
      - type: rectangle
        y: 50
        z: 20
        anchor_x: left
        anchor_y: center
        width: 0
        height: 16
        corner_radius: 3
        corner_segments: 3
        color: 00FF00
        animations:
          player_progress:
            - property: width
              value: (value)
```

Easing Instructions

MPF has the ability to use "easing" functions to adjust the acceleration and deceleration of motions associated with *slide transitions* and *widget animations*.

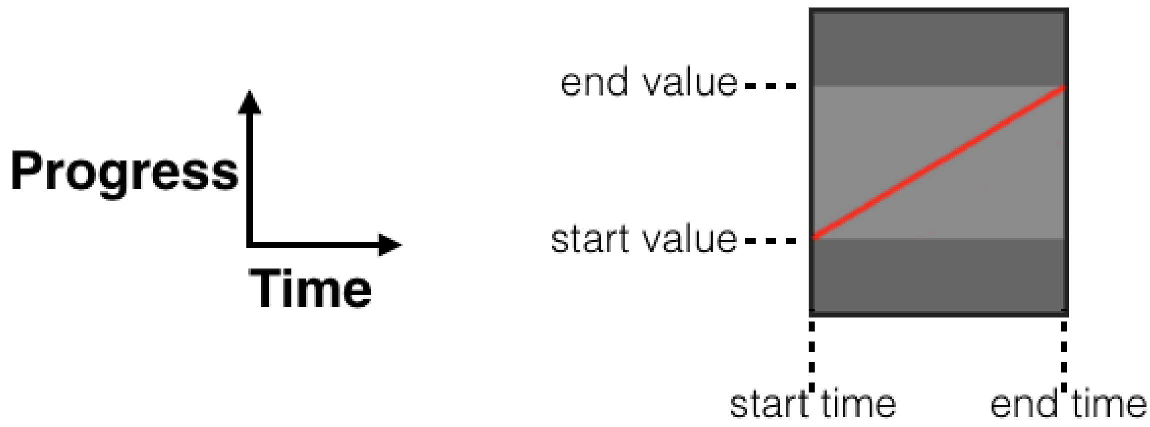
An easing function is a formula that calculates a progress value based on an input value.

Let's look at a simple (but not realistic) example of animating a widget that moves 10 pixels in 10 seconds. With no easing function applied, it would have moved 1 pixel after 1 second, 2 pixels after 2 seconds, etc.

At first you might think this seems fine, but to the viewer it will not look natural because it will instantly start moving at full speed, and then it will stop suddenly when it gets to the end. A more natural approach would be to have it accelerate slowly at the beginning and then to decelerate as it approaches the end.

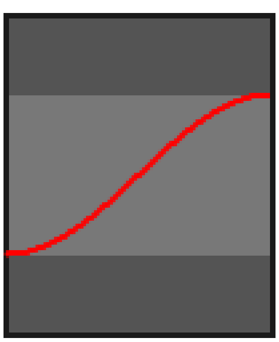
All animation and transition functions in MPF change a value over a certain amount of time. (Move 50 pixels in 2 seconds, change the opacity from 100% to 50% in 500ms, etc.)

We can illustrate this with a graph, where time is the X axis, and the value is the Y axis, like this:



The image above shows the default formula with no easing applied. (This is technically the "linear" easing function.) The value of the function is directly related to the time, and the speed of change is the same at the beginning and end.

But what if we wanted our animation to start slow and accelerate, then slow down again towards the end? For that, we could use a formula like this:



Notice that at the beginning (in the lower left corner), as you move right, the red line doesn't change too much. Then towards the middle, the red line changes more as the transition speeds up, and then at the end (towards the upper right), the line changes more slowly.

Here's an animated GIF which shows five different easing functions applied to animate text moving left and right.

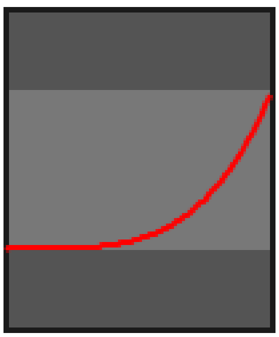
Don't worry about the function names. We'll cover those in a bit.

Note: If you're viewing the PDF version of these docs, you won't see the GIFs since they're animated. You can view the docs online to see them.

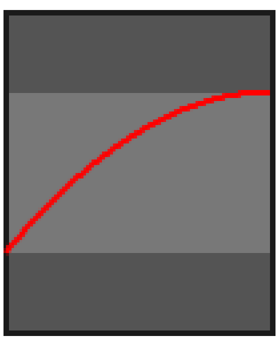
Note that the move to the left and the move to the right are two separate animations, meaning the a single movement left or right is showing the same easing function used in both directions.

If you're curious about the MPF config used to create this animated GIF, we've posted it here.

You can also imagine how an easing formula would look if you wanted something to start slow, but then speed up without slowing down again. (This might be useful if you want a widget to move off screen since it will have a gentle start and then it will shoot off and get faster and faster.) That function might look like this:



Conversely, if you have a widget coming in from off screen, you might want it to start out fast and then slow down as it approaches its final location. For that you could use what's essentially the opposite of the previous formula, like this:



The important thing to remember with these easing formulas is that the red line does NOT represent the path the moving objects take, rather, it represents how the progress of the change happens over time.

Where can you apply easing?

In MPF, these easing functions are used in two places:

- For widget animations, to affect how the progress of an animated property progresses over time.
- For some (not all) slide transitions, to affect the progress of the transition over time.

Remember when you're animating a widget, you can animate ANY numerical property. So this can include the x/y position on the display, but it can also include the size, scale, and/or the opacity (transparency).

Here's an animated GIF showing the same five easing functions applied to each text widget's opacity property (cycling them between 1 and 0):

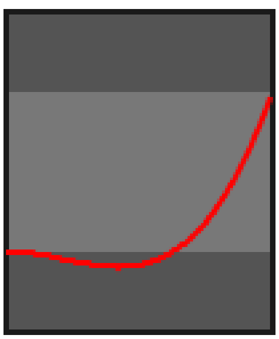
Refer to the *slide transition* and *widget animation* documentation for details on how to actually apply these easing functions. It's pretty straightforward—essentially you just add `easing: <function_name>` to the animation or transition property, like `easing: in_out_circ`.

Now lets look at the different types of easing functions MPF supports:

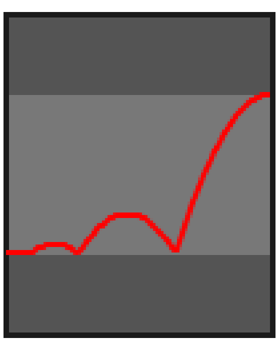
Easing "start" functions

The following functions apply an easing formula at the beginning of the time and then accelerate to the end:

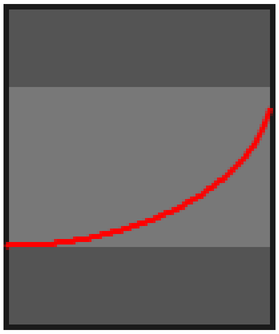
`easing: in_back`



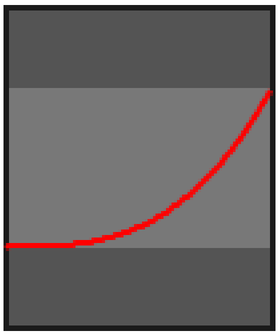
`easing: in_bounce`



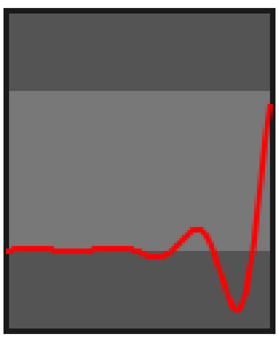
`easing: in_circ`



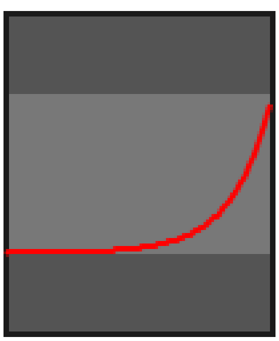
easing: in_cubic



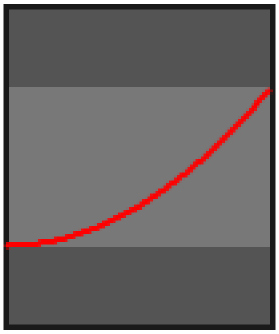
easing: in_elastic



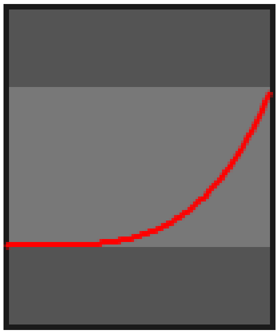
easing: in_expo



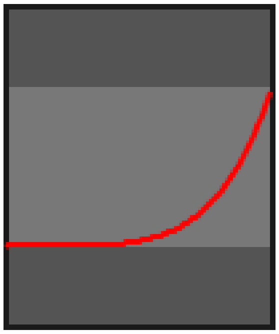
easing: in_quad



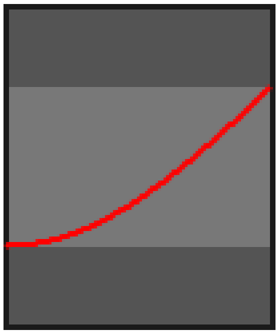
easing: in_quart



easing: in_quint



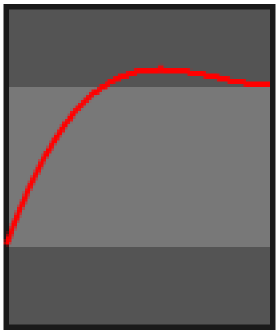
easing: in_sine



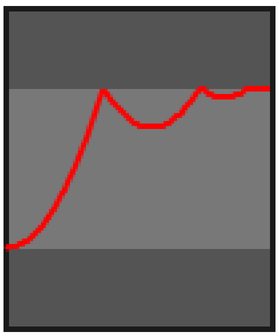
Easing "end" functions

The following functions apply an easing formula at the end of the time, meaning they start fast and then slow down towards the end:

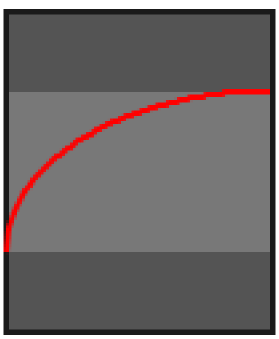
easing: out_back



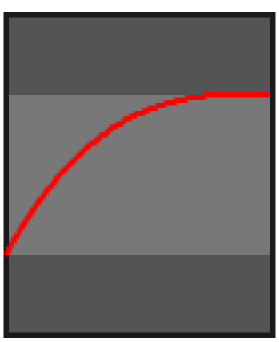
easing: out_bounce



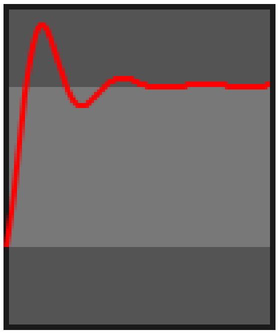
easing: out_circ



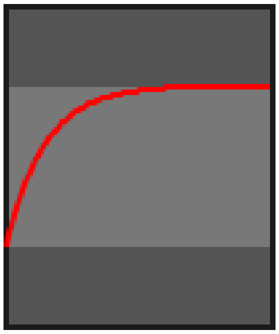
easing: out_cubic



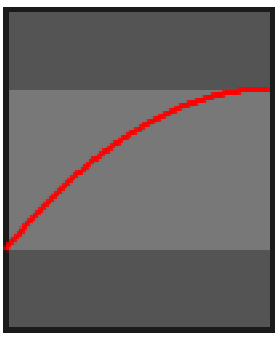
easing: out_elastic



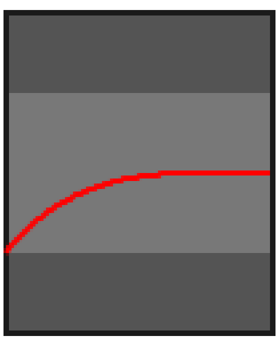
easing: out_expo



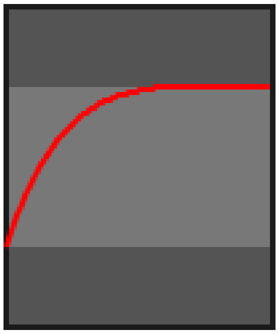
easing: out_quad



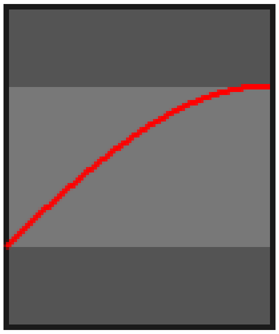
easing: out_quart



easing: out_quint



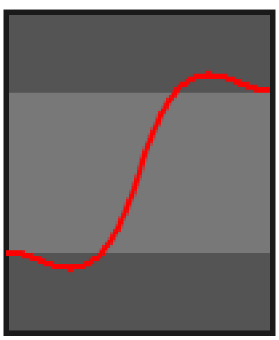
easing: out_sine



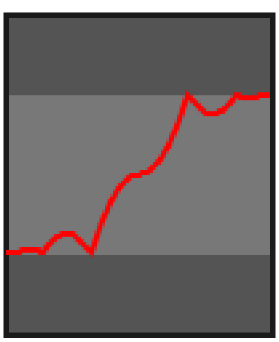
Easing both "start" and "end" functions

The following functions apply the easing to both the beginning and the end of the time, meaning they start slow, accelerate in the middle, and then slow down again at the end.

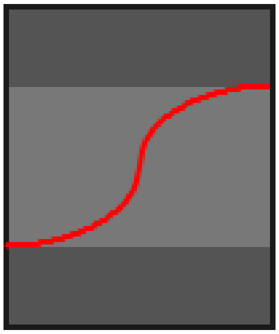
easing: in_out_back



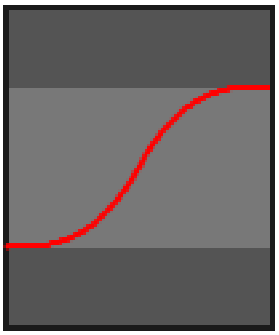
easing: in_out_bounce



easing: in_out_circ

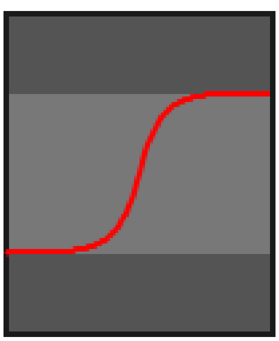


easing: in_out_cubic

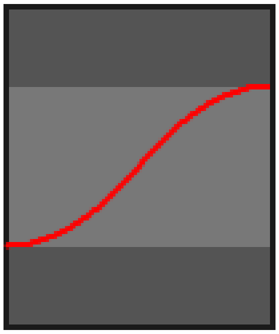


easing: in_out_elastic

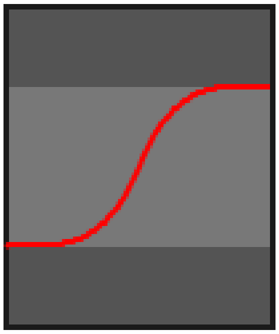
easing: in_out_expo



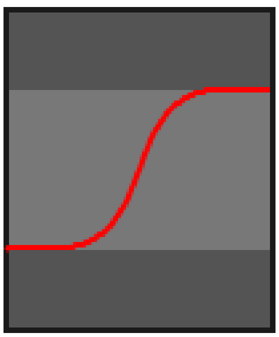
easing: in_out_quad



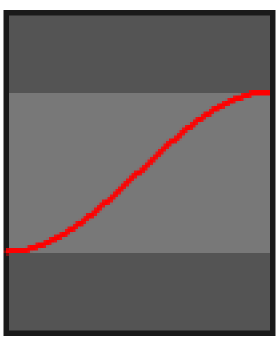
easing: in_out_quart



easing: in_out_quint



easing: in_out_sine



We'd like to give a shout out and thanks to the creators of the Kivy multimedia library (which is what the MPC MC uses) for creating the graphs we used in our easing documentation.

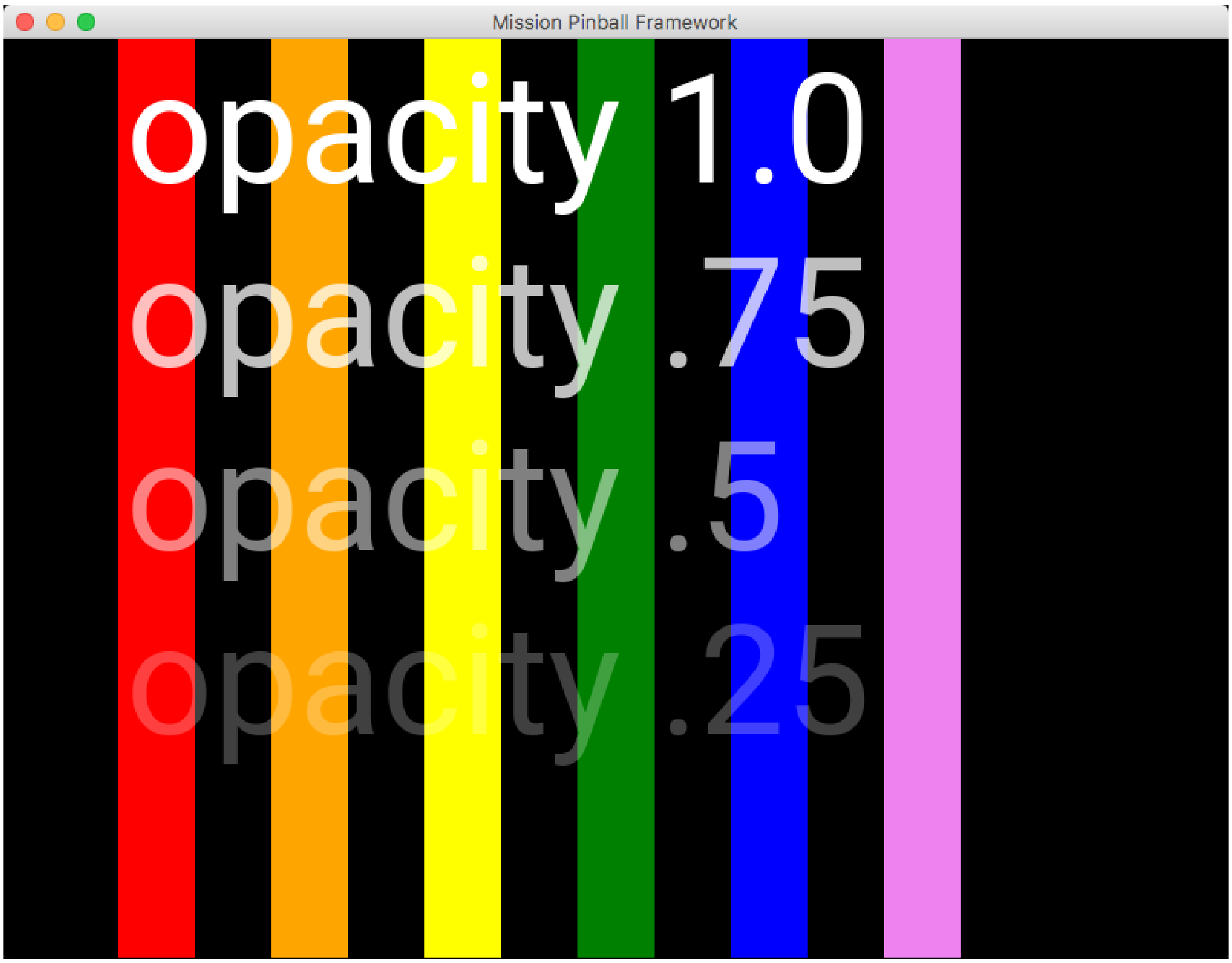
Widget Styles

See *widget_styles:*.

Widget Opacity & Transparency

All widgets in MPF can have "opacity" settings which control how transparent they are. 100% opacity is the default, where nothing would show through that widget. 0% opacity means that the widget is completely transparent and would not show up at all. 50% means it's about half-way in between, etc.

Here's an example. (This example is from the *MC Demo* which you can download and run to see it in action.)



Specifying opacity by opacity: setting

Every widget type has an optional setting called `opacity:` which you can use to set the opacity of that widget. This is a value from 0.0 to 1.0, with `0` meaning 0% opacity (completely transparent and not visible at all), `1.0` meaning 100% opacity (the default), `0.25` meaning 25%, etc.

Note that you can animate the opacity setting to cause a widget to blink or flash. This is easier than adding and removing the widget over and over, as with this method the widget stays put, it's just alternating between visible and invisible. See the *How to animate display widgets* guide for details.

You can apply opacity settings to all widget types, including images and videos. (The opacity setting will affect the opacity for every pixel in the image or video. If you just want an image with transparent parts, then you would use a PNG or GIF with alpha settings instead.)

Specifying opacity by color

For widget types that accept `color:` settings (text and the various shape widgets), you can specify a transparency level as part of the color by adding a fourth byte to the color hex value. (If your color value is only six characters, MPF automatically adds `ff` (fully opaque) to the end.

For example, regular red with 100% opacity would be:

```
color: ff0000
```

Or it would also be (this is the same as the prior example):

```
color: ff0000ff
```

If you wanted red with 50% opacity, you could enter:

```
color: ff000080
```

There's not really any difference between setting the opacity at the `color:` setting versus the `opacity:` setting. The opacity setting is nice because it's applicable to all widget types (including those without color settings), and it's animatable. But the color setting is nice because you can set the opacity and color at the same time. It really doesn't matter.

Working with Fonts

You can specify which font you want to use as a property of any of the widgets that contain text. You can use system-wide fonts that are installed on the computer running MPF as well as fonts that are in your machine's `/fonts` folder.

You specify fonts by name only (not including the extension), and MPF will first look in your machine's fonts folder, and if it doesn't find the font there, it will look in the MPF-MC's built-in fonts folder, and finally in your machine's system fonts location.

Note: The MPC MC contains a few pixel-based for use on DMDs. See *How to use DMD fonts* for details.

For consistency of appearance across computers, we highly recommend that you put the fonts you want to use in your machine's fonts folder.

Specifying which font a particular widget uses is done via that widget's `font_name:` setting, so see either the *Text Widget* or *Text Input Widget* reference for details.

Keep in mind that all widget properties, including fonts, can be configured as part of a widget style and easily applied to new widgets with a single line.

How to use DMD fonts

MPF includes three built-in fonts which are pre-configured as widget styles which look good on DMDs. These fonts are included in the MPF-MC package. They can be used with any widget that uses fonts, including the Text and Text Input widgets.

If you don't use one of these fonts on your DMD and just show some text, here's what the results look like:

```
slides:
  my_slide:
    - type: text
      text: MISSION
```

Sure, it works, but it doesn't look good because the default font is a regular font that's made for a high-res display.

Instead you can use these three styles. (Of course you can use your own fonts too, but sometimes it's hard to find ones that look good on a low-res DMD.)

style: big

big is 10 pixels tall.

```
slides:
  my_slide:
    - type: text
      style: big
      text: MISSION
```

style: med

medium is 7 pixels tall.

```
slides:
  my_slide:
    - type: text
      style: medium
      text: MISSION
```

style: small

small is 5 pixels tall.

Notice that this font has a color set and we're using it with a Color DMD. All three of these fonts (like any font) can be used on a mono or color DMD.

```
slides:
  my_slide:
    - type: text
      style: small
      text: MISSION
      color: 00ffcc
```

MPF also supports *Bitmap Fonts* if you want to create your own fonts for your machine.

How to Set Fonts and Sizes Globally?

You usually want to use three to five different font + size combinations in your machine. However, you often want to adjust them later on without touching all your slides and widgets. For that reason it makes sense to define a *widget_style* for all your fonts and sizes:

```
widget_styles:
  text_small:
    font_size: 15
    color: red
  text_default:
    font_size: 21
    color: blue
```

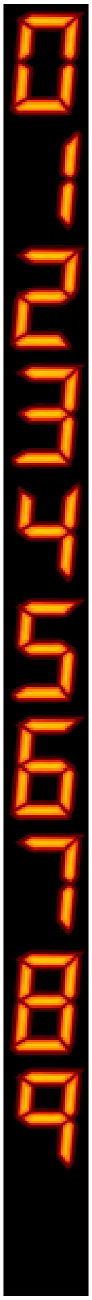
You can then use `style: text_default` in your text widgets. See *widget_style* for details.

Bitmap Fonts

You can create your own fonts for your machine using bitmap fonts. There are several programs or online tools to create bitmap font descriptors.

1. Create an Image For Your Font

An example might look like this:



2. Map Your Characters in a Descriptor File

This file might look like this:

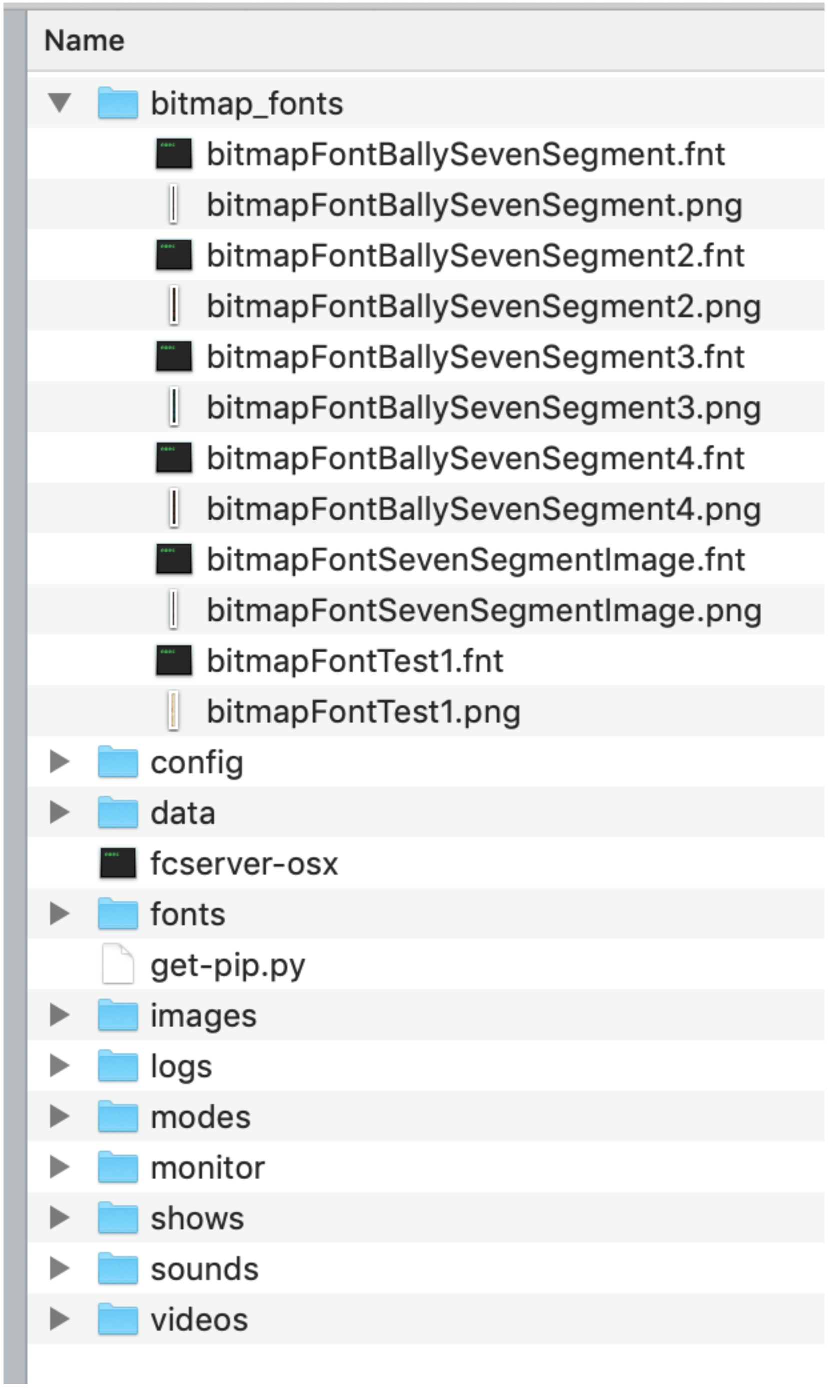
```
info face=font size= bold= italic= charset= unicode= stretchH= smooth= aa= padding=0,0,0,0 spacing=0,0␣
↪outline=0
common lineHeight=55 base=55 scaleW=40 scaleH=55 pages=1 packed=0
page id=0 file="bitmapFontBallySevenSegment4.png"
chars count=11
char id=48 x=0 y=0 width=40 height=55 xoffset=0 yoffset=0 xadvance=40 page=0 chnl=15
char id=49 x=0 y=55 width=40 height=55 xoffset=0 yoffset=0 xadvance=40 page=0 chnl=15
```

(continues on next page)

(continued from previous page)

```
char id=50 x=0 y=110 width=40 height=55 xoffset=0 yoffset=0 xadvance=40 page=0 chnl=15
char id=51 x=0 y=165 width=40 height=55 xoffset=0 yoffset=0 xadvance=40 page=0 chnl=15
char id=52 x=0 y=220 width=40 height=55 xoffset=0 yoffset=0 xadvance=40 page=0 chnl=15
char id=53 x=0 y=275 width=40 height=55 xoffset=0 yoffset=0 xadvance=40 page=0 chnl=15
char id=54 x=0 y=330 width=40 height=55 xoffset=0 yoffset=0 xadvance=40 page=0 chnl=15
char id=55 x=0 y=385 width=40 height=55 xoffset=0 yoffset=0 xadvance=40 page=0 chnl=15
char id=56 x=0 y=440 width=40 height=55 xoffset=0 yoffset=0 xadvance=40 page=0 chnl=15
char id=57 x=0 y=495 width=40 height=55 xoffset=0 yoffset=0 xadvance=40 page=0 chnl=15
char id=32 x=0 y=550 width=40 height=55 xoffset=0 yoffset=0 xadvance=40 page=0 chnl=15
```

3. Put PNG File and Descriptor Into the bitmap_fonts Folder



Some things to note:

- The file name of the image is defined in the .FNT file
- The ASCII code for each character is defined by a starting position (x, y for the upper left corner) and a width and height value.

4. Use the Font in Your Slide

You can use the font in your config:

```
slides:
  slideBaseBackglass:
    widgets:
      - type: text
        text: (player1|score)
        font_name: bitmapFontBallySevenSegment4
        bitmap_font: true
```

Alternatively, you can also use bitmap fonts in *widget styles*.

How to create reusable widgets

This guide explains how you can create reusable "named" widgets that you can use again and again on multiple display slides. This saves you from having to copy-and-paste the same widget (or sets of widgets) into multiple slide configurations, it makes it easy to update and fine-tune your widget config since you only have to change it in one place, and it lets you add individual widgets to the display that will show up regardless of what slide is currently showing.

1. Understanding widgets

Before we look at how to create reusable widgets, let's look at how regular widgets work in MPF.

You probably know that you can have a `slides:` section of your config (either machine-wide or mode-specific configs), and when you define a slide, you can specify what widgets are on that slide, like this:

```
slides:
  my_slide:
    widgets:
      - type: text
        text: HELLO!
      - type: text
        x: 0
        font_size: 5
        text: YAY PINBALL
      - type: image
        image: background1
```

In the example above, the slide called *my_slide* has three widgets–two text widgets and a background image. (Remember that the "z order" or "layer" of widgets is top-to-bottom, so the *HELLO!* widget is on top, then *YAY PINBALL* is next, and they're both on top of the *background1* image.

These three widgets are permanently attached to the slide called *my_slide*. There's no way to reuse them on any other slides.

2. Creating reusable widgets

But what if you had a widget you wanted to use on multiple slides? For example, maybe you have a widget with some animations that comes on the display when a certain shot is made, and you want that widget to appear on any slide (whichever slide happens to be showing at that time).

The way to do that is to create a "named" widget that's reusable. You do that in the `widgets:` section of your config. (This can be either a machine-wide or a mode config file.)

For example:

```
widgets:
  laughing_jackal:
    - type: image
      image: jackal
```

Now you have a widget defined called *laughing_jackal* that you can add to any slide. (Note that this example is simple, but any widget type with any widget settings can be defined here, including positioning, colors, animations, etc.

The only "catch" is that the list of widget names is global across MPF. So even though you can define widgets in both the machine-wide or the mode config files, named widgets are processed when MPF starts up, so don't use the same name twice since whichever one loads second will overwrite the first one.

3. Using your named widget

Now that you have a widget defined, how do you add it to a slide? That's done via the "widget" config player, which means you can add a `widget_player:` section to a config file to trigger it based on an event, or you can add it via the `widgets:` section of a show step. (All the examples in this guide will be based on the `widget_player:` section of a config file, but you can use them all in show steps too. Just use them in a `widgets:` section of a show step and do not include the event name.

There are several options you can use in the widget player, depending on how you and where you want to show your widget (which display, which slide, etc.)

"Express" config

If you just want to add your widget to whichever slide is current on the default display, you can use the "express" config, like this:

```
widget_player:
  some_event: laughing_jackal
  some_other_event: another_widget
```

With the config above, when the event *some_event* is posted, the widget called *laughing_jackal* will be added to the current slide on the default display. Notice that you can add multiple entries here for different widgets and different events.

This widget is added with whatever settings you defined for it in the `widgets:` section of your config. It's all pretty straightforward, though you might have to play with the `z:` setting (the layer) to get it to show up. (For example, if your current slide has a full size background, you'd want to configure your widget with a `z:` setting that's a higher priority so it shows up on top of the background image.)

Adding a widget to a specific slide (by slide)

If you want to build a slide and include a reusable widget, you can reference the widget's name in your slide config by declaring `widget:` instead of `type:`.

```
widgets:
  jackpot_value_widget:
    - type: text
      text: (jackpot_total)
      style: medium
slides:
  hero_hurryup:
    - type: text
      text: "Hurry Up!"
    - type: text
      text: "Jackpot:"
    - widget: jackpot_value_widget

slide_player:
  show_hero_slide: hero_hurryup
```

Adding a widget to a specific slide (by event)

If you want to add your widget to a particular slide (versus whatever slide happens to be showing at the moment), you can do so by specifying that slide name in the `widget_player:`. For example:

```
widget_player:
  some_event:               # event that will trigger this widget to show
    laughing_jackal:        # widget you want to show
      slide: my_slide
```

In the example above, when the event *some_event* is posted, the widget *laughing_jackal* will be added to the slide called *my_slide*. If *my_slide* is the current active slide on the display, you'll see the widget appear. If that slide is not being shown, the widget will still be added, and it will be there the next time that slide is shown.

Remember you can add as many events and widgets as you want to the `widget_player:` section of your config, and you can even mix-and-match formats, like this:

```
widget_player:
  some_event:
    laughing_jackal:
      slide: my_slide
  some_other_event: another_widget
```

Adding a widget to a specific display target

Rather than specifying a particular slide to add your widget to, you can target a display, and the widget will be added "on top" of whatever slide is currently being shown:

```
widget_player:
  some_event:
    laughing_jackal:
      target: display1
```

Remember in MPF, display targets are the names of a display (dmd, window, etc.).

More details about this are in the *Widget layers, z-order, & parent frames* guide.

Overriding named widget settings

When you create your named widget, it contains a bunch of settings that are used to add it to a slide. (That's sort of the whole point.)

However sometimes it's useful to be able to override or add additional settings at play time. You can do this in the `widget_settings:` section of the `widget_player:` in a config file or the `widgets:` section of a show step.

For example, if you use a widget for the tilt warning like in the previous example, you'd probably want that widget to be removed after a few seconds, which you could do like this:

```
widget_player:
  tilt_warning:              # event
    tilt_warning:            # widget name
      widget_settings:       # additional settings to be added / updated
        expire: 2s
```

(Technically speaking, if you were going to show a tilt warning widget, you'd probably also want to play a sound and maybe flash all the lights on the playfield, so in your real game you're probably actually create a show to do this and then play it via the `show_player:` section of your config and include the widget in the `widgets:` section of the show, but you get the idea.)

You can also set the expiration time of a widget when you define the widget in the `widgets:` section of the config. See the config file reference for details.

You can add/update any setting for the widget (color, text, position, animations, widget_styles, z (layer), etc.)

Removing widgets

You can also use the widget player to remove named widgets from a slide that had been previous added. To do this, just add an `action: remove` setting to the widget player, like this:

```
widget_player:
  show_jackal: laughing_jackal
  hide_jackal:
    laughing_jackal:
      action: remove
```

The config above will add the *laughing_jackal* to the current slide on the default display when the event *show_jackal* is posted, and then it will remove it when the event *hide_jackal* is posted.

Creating named groups of widgets

All of the examples in this guide showed using a single widget as named widget. But you can actually define multiple widgets in a named widget (essentially meaning that your named widget is really a named group of widgets. For example:

```
widgets:
  widget3:
    - type: text
      text: HI
      color: ff0000
      font_size: 100
    - type: text
      text: THERE
      color: 00ff66
      font_size: 100
    - type: text
      text: EVERYONE!
      color: ff00ff
      font_size: 100
```

You play, show, or hide this “widget” in the same way as every other example in this guide, except in this case, playing *widget3* will actually add all three widgets to the slide. (Again you can play with z-order / layering, and remember that each widget (even in a multi-widget group) can have its own z-order settings.

Putting it all together, these are the basics of using named widgets in MPF. The important takeaways are:

- Widget names are global, so don’t use the same name twice.
- Everything here can be done in either the `widget_player:` section of a config file or the `widgets:` section of a show step.
- All widget options are valid, including keys, animations, expiration, styles, positioning, z-ordering, colors, transparencies, padding, etc.
- When “playing” a widget, you can target a display or a slide.
- Once a widget is “played” and added to a slide, it becomes just another widget on that slide. The fact that it was put there by the widget player doesn’t matter.

Adding multiple named widgets in one event

You can also add multiple named widgets from a single event. This is nice if you want to add widgets to multiple displays or slides at the same time. For example:

```
widget_player:
  some_event:
    widget1:
      target: dmd
    widget2:
      target: lcd
```

Note that if you do this, the structure of YAML requires that you have at least one setting under each widget name, so you can just add a `target:` or `action: add` if you don't want to change or set anything else in the widget.

Dynamically choosing a widget based on variables

You can use a placeholder widget in a slide to dynamically choose any reusable widget for that slide, depending on an event parameter or player variable.

To create a placeholder widget in the slide, use the `widget:` setting with the standard *dynamic text* formatting.

For example, using the player variable "hero_class" to pick a text image (but could be an image widget as well):

```
widgets:
  hero_portrait_rogue:
    - type: text
      text: "Portrait Rogue"
  hero_portrait_bard:
    - type: text
      text: "Portrait Bard"
  hero_portrait_mage:
    - type: text
      text: "Portrait Mage"
slides:
  hero_slide:
    - type: text
      text: (player|name)
    - type: text
      text: Level (player|level)
    - widget: hero_portrait_(current_player.hero_class)
slide_player:
  show_hero_slide: hero_slide

##! mode: base
variable_player:
  set_var_rogue:
    hero_class:
      action: set
      string: "rogue"
```

You can also use the parameters of an event to determine the widget to include. In the following example from a game with different multiballs, the event *mball_lock_lit* might post with either "angel" or "demon" as the *mball_name* parameter.

```
slide_player:
  mball_lock_lit: mball_lock_slide
slides:
  mball_lock_slide:
    widgets:
      - type: text
        text: Lock is Lit
      - widget: lock_lit_(mball_name)
```

(continues on next page)

(continued from previous page)

```
widgets:
  lock_lit_angel:
    - type: text
      text: Angels Anarchy
    - type: image
      image: bg_locklit_angels
  lock_lit_demon:
    - type: text
      text: Demons Derby
    - type: image
      image: bg_locklit_demons
```

Expiring (auto removing) widgets

You can use the widget player to add widgets to slides which will be removed automatically after a pre-determined about of time. This is done via a widget's "expire" setting. There are several ways you can expire a widget:

Option 1: In the widget or slide definition

```
widgets:
  my_widget:
    type: text
    text: HELLO
    expire: 2s
```

In the example above, whenever you add that widget to a slide (via the widget_player or the widgets: section of a show), that widget will expire and disappear two seconds later.

Option 2: In the widget player

Instead of tying an expire time to a widget when you define the widget, you can specify the expiration when the widget is shown via the widget player.

Here's an example:

```
widgets:
  my_widget:
    type: text
    text: HELLO    # no expiration here
widget_player:
  show_widget_event:
    my_widget:
      widget_settings:
        expire: 2s
```

In the above example, the widget player dynamically adds the 2 second expiration time when the widget is shown after *some_event* is posted.

Option 3: Remove a widget on some event

Instead of automatically removing a widget after a pre-determined amount of time, remember you can use the widget player to remove a widget by name, which means you can use one event to show the widget and another event to remove it. For example:

```
widgets:
  my_widget:
    type: text
    text: HELLO    # no expiration here
widget_player:
  show_widget_event: my_widget
  remove_widget_event:
    my_widget:
      action: remove
```

In the example above, the event *some_event* will cause my_widget to be added to the current slide on the default display, and the event *some_other_event* will cause it to be removed.

Widget Keys

Widget keys are used to uniquely identify instances of widgets which you can later use to update or remove the widget.

Note that you can you can also identify widgets by name (which is almost always more straightforward). You only need to use a key if you want to put multiple instances of the same widget on the same slide, and then you need a way to identify a individual ones to update or remove them.

Adding the Same Widget Multiple Times

When adding the same widget to a slide or target simultaneously, keys are used to differentiate the widgets from one another. An important aspect to note is that only one instance of a specific widget can be modified with a given event for the widget_player. This means that if you want to add the same widget multiple times, you need to have unique events to call each widget. This can be done in one of two ways, which are shown below.

Using the Same Event With Different Priorities

This is an example using priorities of the events, which will affect the priority:

```
widget_player:
  some_event.1:
    widget_1:
      key: widget_1_1
      slide: slide_2
      widget_settings:
         # <list of settings below go here>
  some_event.2:
    widget_1:
      key: widget_1_2
      slide: slide_2
```

(continues on next page)

(continued from previous page)

```
      widget_settings:
         # <list of settings below go here>
```

It will add widget_1 to slide_2 two different times. In order to make this meaningful, you would want to add additional `widget settings:`, such as position, rotation, color, opacity, etc. This is important, otherwise it will add the widget with the same settings twice, which would overlap each other.

Using the Same Event With Different Conditional Logic

An additional method would be to have unique events that call the same widget multiple times. This could be done in one of two ways: completely unique events (example: event_1 and event_2) or by using conditional logic on the same event (example: event_1{param1} and event_1{param2}.

This is an example using unique conditional formatting for the same event:

```
widget_player:
  some_event{parameter_1 <10}:
    widget_1:
      key: widget_1_1
      slide: slide_2
      widget_settings:
         # <list of settings below go here>
  some_event{parameter_1 < 50}:
    widget_1:
      key: widget_1_2
      slide: slide_2
      widget_settings:
         # <list of settings below go here>
```

It will add widget_1 to slide_2 if the conditional criteria is met. If the criteria is met for both of the events, they will both be played at the same time. If they are both played at the same time, you would likely want to add additional widget settings, such as position, rotation, color, opacity, etc. This is important, otherwise it will add the widget with the same settings twice, which would overlap each other.

Remove or Update a Specific Widget Instance

To remove or update a specific instance of a widget from the page, you need to refer to the key of that widget. This is done by the following code, which has calls upon the generic widget and the key when an event is posted.

```
widget_player:
  some_event:
    widget_1:
      key: widget_1_1
      action: remove  #this could also be update
      widget_settings:
         # <list of settings below go here>
```

The above block of code would listen for some_event to occur, and then remove the instance of widget_1 with the key widget_1_1. You can also use the `action: update` and a set of `widget_settings:` to update the widget with the new properties.

Widget layers, z-order, & parent frames

When you have multiple widgets on a slide, you can control the layer (or z-order) of the widgets, controlling which widgets are on top of others in cases where two or more widgets overlap.

When adding a widget to an existing slide, you also have the option to add it to the "parent frame" (and not to the slide), meaning that if the slide changes, the widget will still be there.

Let's look at how all this works.

Overlapping widgets, layers, & z-order

Any time you have two widgets that overlap, MPF must decide which widget will be drawn "on top" of the other.

At the most basic level, any time you have more than one widget listed in a config (whether it's in a `widget_player:`, `slide_player:`, or a definition in a `slides:` or `widgets:` section), the widgets will be drawn in the order they are in the config.

For example, here's a slide that has widget3.1, then widget3.2, then widget3.3:

```
slides:
  3_widgets:
    - type: text
      text: widget3.1
      color: red
      font_size: 80
      y: 40%
    - type: text
      text: widget3.2
      color: orange
      font_size: 80
      y: 50%
    - type: text
      text: widget3.3
      color: violet
      font_size: 80
      y: 60%
```

The result is like this. Note that widget3.1 is on top of widget3.2, which is on top of widget3.3:

In this example, all three widgets are 100% opaque, but if any of them had opacity of less than 100%, then you would see the lower level widget through the higher one. See the *Widget Opacity & Transparency* guide for details.

You can also use the `z:` setting to manually set the relative order of how you want the widgets to overlap. Widgets with higher `z:` values will be drawn on top of those with lower values.

Here's the same example as before, but with `z:` values added:

```
slides:
  3_widgets:
    - type: text
      text: widget3.1
      color: red
      font_size: 80
      y: 40%
      z: 1
    - type: text
      text: widget3.2
      color: orange
      font_size: 80
      y: 50%
      z: 100
```

(continues on next page)

(continued from previous page)

```
  - type: text
    text: widget3.3
    color: violet
    font_size: 80
    y: 60%
    z: 2
```

And the results:

Note that *widget3.2* is on top since it's `z:` is 100, then *widget3.3* is next with `z: 2`, and finally *widget3.1* is on the bottom with `z: 1`

Notes about z-order:

- The default `z:` value is `0`, so anytime you have a widget without a `z:` setting, it's like you have `z: 0`.
- The order the widgets are listed in the config file is only used as a tie-breaker if multiple widgets have the same `z:` settings. (This is why the first example worked, since all three widgets had `z: 0`.)
- You can mix-and-match order and `z:` settings.
- The actual numeric `z:` settings don't matter. You can have 1, 2, 3 or 100, 200, 300, or 1, 20000,

1000000 or whatever you want.

- Setting `z:` values for widgets on a slide is only really used if you want to later use the widget player to add a widget to a slide in between certain existing widgets.
- In most slides, you will not mess with `z:` settings and instead use the order of the widgets in the config file to set the order they are on the slide.

Adding widgets to parent frames

When you use the `widget_player:`, it will add the widget to the current slide on the default display.

If you want to target a specific slide, you can add a `slide:` setting to your widget player with the name of the slide.

In both cases, the widget player will add the widget to a slide.

However, it's also possible to add a widget to the "frame" which holds the slides, meaning that the widget is shown "on top" of the slide rather than as part of the slide.

Why would you want to do that?

Sometimes it's useful to have a widget which "stays put" even as the underlying slides change.

One example is for tilt warnings. When the player gets a tilt warning, you might want to show the text "WARNING" for 2 seconds. However if you use the regular widget player to add this widget to the current slide, then if that slide is replaced by another slide during those 2 seconds, your tilt warning will disappear too.

Another example is the scores. Maybe you want those to show along the bottom on top of every slide? Or maybe something like the news crawl on the bottom of the *Dialed In* display?

To do this, you can add the setting `z: -1` under the `widget_settings:` section of your desired widget. Here's an example:

widget_player:

trigger_event:

my_widget:

widget_settings: z: -1

This will make the widget added to the frame, and not to a slide, meaning your widget will ride "on top" of every slide (and even on top of any slide transitions that take place).

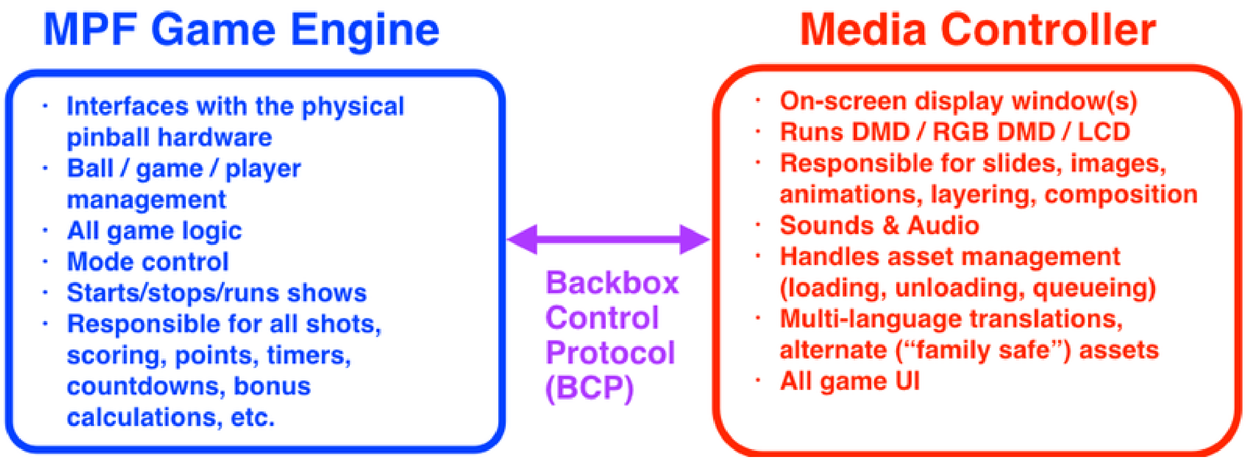
Widgets versus Slides: When to use each?

Help us to write it

Media Controllers

One of the most important things to understand about the architecture of MPF is that the core MPF game engine is completely separate from the process that controls graphics and audio. We call the thing that handles graphics and audio a "media controller." The game engine and media controller talk to each other via something called "BCP" which is a protocol we created for this purpose which stands for "Backbox Control Protocol". (More details on BCP are available at the MPF developer site.)

Here's a diagram that shows what each piece does:



Why are the MPF game engine and media controller two separate processes? Two reasons:

First, having two processes means that each one can run on a separate core in a multi-core host computer. This makes efficient use of hardware since the trend is to have multiple cores. If the game engine and media controller were combined, then your quad-core Raspberry Pi 3 would have all the MPF stuff running on one core while the other three cores were wasted doing nothing.

Second, having two processes means you can replace MPF's default media controller with something else if you want different features. For example, there is a group of people building an open source *Unity 3D-based media controller* which can be used for very advanced 3D display graphics.

The MPF Media Controller

The MPF media controller (which we call "MPF-MC") is the default media controller option that 99% of MPF users use. (If you haven't read about what a media controller is and how it fits into MPF, *do that first*).

Like MPF, the MPF-MC is also written in Python, using a Python-based multimedia framework called Kivy. Kivy is a wrapper for the native graphics & sound libraries on your computer, and it leverages the latest technologies including SDL2, Gstreamer, and OpenGL.

All of the tutorials and installation guides included in this documentation explain how to install and use the MPF-MC, so there's really nothing to know about it other than it's probably the one you're using.

The MPF Unity BCP Server

The MPF Unity BCP Server is a Unity 3D-based media controller for MPF. You can use it if you want to program your machine's graphics and sounds via Unity 3D. (If you haven't read about what a media controller is and how it fits into MPF, *do that first*).

MPF's Unity BCP Server is also free & open source, and hosted in the unity-bcp-server repo in the Mission Pinball GitHub account. See the readme in that repo for more details including instructions on how to use it.

How to run MPF and the MPF-MC on different computers

Since the BCP protocol uses a standard TCP socket connection, you can actually run MPF and the MPF-MC on different computers. (We're not sure what the use case for this is exactly, but it's definitely possible.)

To do it, just install MPF on one computer and MPF-MC on another.

Then on the machine running MPF, configure the `host:` setting as the name or IP address of the machine running the MPF-MC, and on the MPF-MC computer, set the `servers:` section to listen on the IP address you want to use. (See the *bcp section of the config file reference* for details.

Remember to set the firewall on the computer running MPF-MC to accept incoming connections on the port that BCP is listening on.

> **Warning:** The BCP protocol has no security, so it's fine if both the computers are inside your pinball machine or on your home network, but it's not designed to be run across a public network.

Multiple Simultaneous Media Controller Connections

You can create multiple BCP connections for multiple media controllers. However, this is not useful on its own. You also need to map slide/widget_players to the right MC. Let us know in the forum if you need this.

Listing 2: your_machine_folder/config/multiple_connections_config.yaml

```
#config_version=5

bcp:
    debug: True
    connections:
        local_display:
            host: localhost
            port: 5050
            type: mpf.core.bcp.bcp_socket_client.BCPClientSocket
            required: True
            exit_on_close: True
        another_display:
            host: localhost
            port: 9001
            type: mpf.core.bcp.bcp_socket_client.BCPClientSocket
            required: True
            exit_on_close: True
```

Creating your own Media Controller

It's possible to create your own media controller for your own specific needs. All you have to do is listen for incoming BCP connections and then parse the commands and from there you can do pretty much anything you want. Let us know in the forum if you want to do this.

CHAPTER 11

Sounds, Music & Audio

Note: Everything in this "Displays & Graphics" section is about default the MPF Media Controller

Since the release of MPF 0.30, audio and sound support has been provided by a custom audio library built on SDL2, SDL_Mixer, and GStreamer libraries. This custom library allows the MPF development team to create audio features optimized for pinball machines. The first release provides basic sound loading and playback capabilities along with some great new features like *ducking* and *sound pools*. (Sound support is part of the MPF media controller and only available if you're using MPF-MC for your media controller).

The basic concept with audio in MPF is that you collect all your audio files (16-bit .wav, .ogg, and .flac files are currently supported) and put them in the `/sounds` folder in your machine folder (you can organize them into sub-folders if you would like). Then in your config file you create entries for each sound which map a friendly name to the actual file on disk. You can also set a bunch of defaults for each sound, such as volume, start time, etc. Then when you want to play a sound in a game, you can refer to it by the friendly name from your configuration file. You can also add entries into your configuration file to set up sounds so they play based on certain MPF events. (For example, play the sound "laser" every time the event from a pop bumper being hit is posted.) You can also add sounds to your show files so they play in-sync with lighting and display effects.

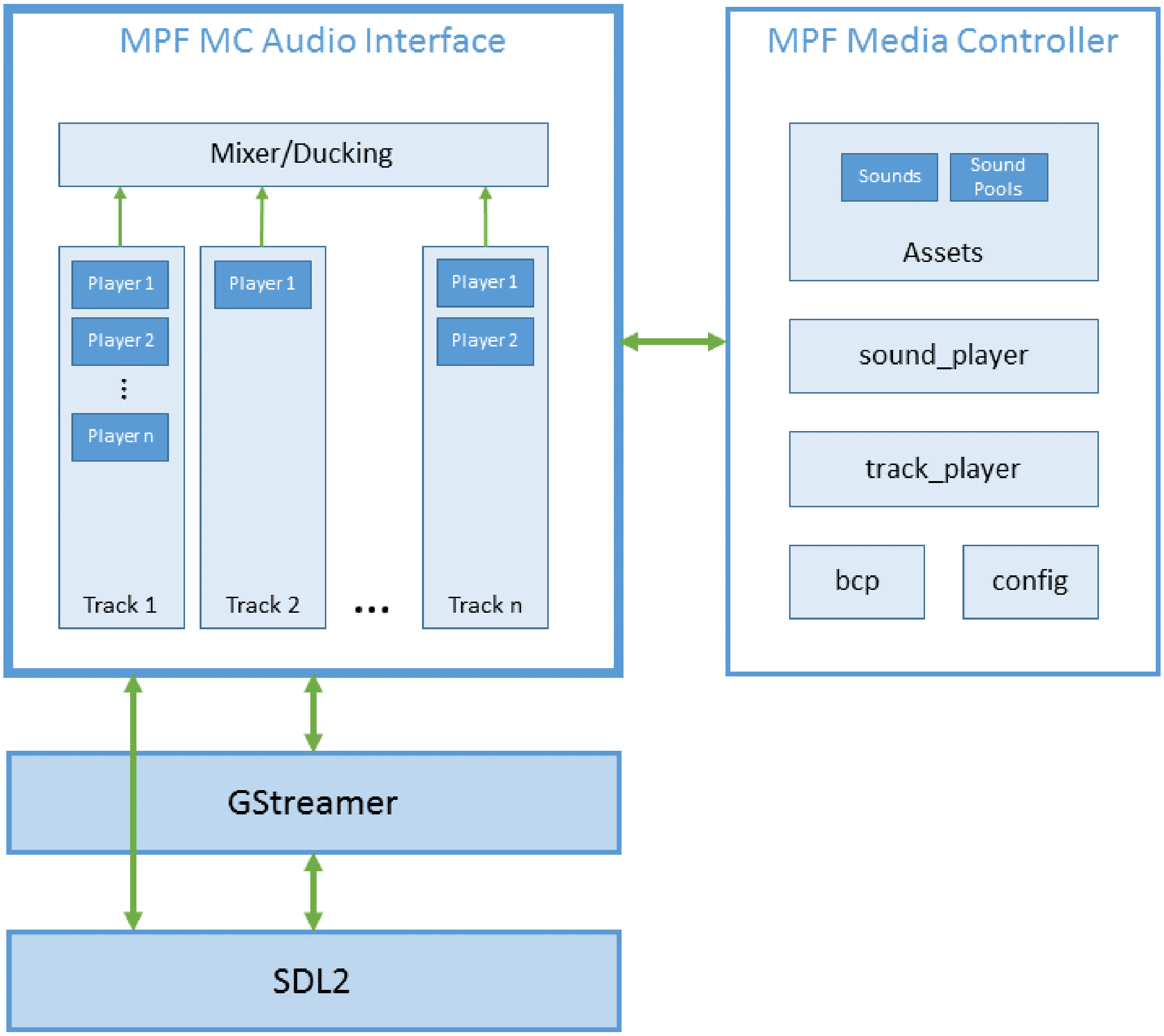
You can think of the audio system in MPF as a sound mixing board that you control via configuration settings and events. It is divided into *tracks* (similar to channels on a mixer), each of which has its own properties such as name, volume and the number of sounds that may be played simultaneously. New in MPF 0.50 are specialized track types optimized for specific audio tasks (such as music playback and creation). You can create up to 8 tracks in your sound system, although typically most machines will typically use 3 standard tracks ("voice", "sfx", and "music"). Sounds are played on specific tracks and then the tracks are mixed together to form the final mix. The sounds themselves are objects that include many properties that control how they will be played such as what track they play on, volume, looping, priority, how long to wait in the playback queue before being discarded, *ducking*, etc.

Sounds can be grouped together into a logical grouping called a sound pool. Sounds pools allow you to reference a group of sound variations as if it were a single sound. A sound pool name may be used anywhere a sound asset name may appear. Pools can be used for random differences in a sound (such as slight variations of a slingshot sound) or for an ordered sequence of sounds that will repeat. Another common use for sound pools is to play a random callout from a defined list when triggered.

You configure your sound system (including tracks) in the *sound_system:* section of your machine configuration file. You add settings for individual sound files in the *sounds:* section. You can configure sounds to automatically play on standard tracks when selected MPF events are posted in the *sound_player:* section. Sound pools are specified in the *sound_pools:* section. Sound loop tracks use *sound_loop_sets* and the *sound_loop_player* to play and loop sounds. Playlist tracks use *playlists* and the *playlist_player* for playing music. Tracks can be controlled when selected MPF events are posted in the *track_player:* section.

MPF Sound & Audio Technical Overview

The MPF MC Audio Interface is a custom audio Python extension library with features designed to support common pinball sound requirements. It is written on top of the SDL2, SDL_Mixer, and GStreamer libraries that are installed with Kivy which is required to run the MPF MC software (no additional installs necessary for the audio library).



The SDL2 library (https://www.libsdl.org/) is responsible for all low-level communications with the system audio hardware. The user selects the basic audio interface settings: sample rate, output channels, and buffer size (defaults are provided). These settings are used to initialize the SDL2 library which then negotiates with the system audio hardware to create a connection that is as close to the desired settings as possible. The SDL2 library is responsible for creating the main audio thread and calling the main audio callback function at a fast enough rate to provide audio buffers to the hardware without any gaps in playback. It also provides the thread synchronization and protection utilized in the audio library through its mutex-related functions. The audio library also uses the SDL2 audio format conversion functions to convert between various low-level audio formats to communicate with the system sound hardware.

SDL_Mixer (https://www.libsdl.org/projects/SDL_mixer/) is an add-on library for SDL2 that provides basic audio mixing, sound loading and playback, and sound streaming capabilities. The MPF MC audio interface does not use the mixing features of SDL_Mixer. Instead, it only utilizes the sound file loading functions of the library.

GStreamer (https://gstreamer.freedesktop.org/) is an open source, cross-platform pipeline-based multimedia framework that links together a wide variety of media-handling components (including simple audio playback, audio and video playback, recording, streaming and editing) to complete complex workflows. The MPF MC audio interface uses GStreamer for all its sound file loading functions and real-time audio streaming. All audio is fed into SDL2 for final output.

The audio interface is divided into *tracks*, which are analogous to channels on an audio mixer. There are multiple types of audio tracks, each with its own specialized feature set. The output of each track is mixed together and fed to the SDL_Mixer track via the custom music player function. The audio mixing engine uses 16-bit integer calculations and brickwall limiting to ensure there are no numeric overflows (and their resulting distortion). All of the sound generation and mixing functions are C functions (written in Cython) that run in the SDL2 audio thread.

It is important to understand the threading models of both SDL2 and Python to avoid common threading problems. Python supports multiple threads, however it uses a mechanism called the "global interpreter lock" (GIL) to ensure that only one thread runs in the Python interpreter at once. This simplifies many low-level details. SDL2 creates its own audio thread in which to receive and process audio data and send it to the audio hardware. As this audio thread is not a Python thread, it does not interact with the GIL and therefore is unable to access any Python objects within its context. This means that only C types and data structures may be utilized in the SDL2 audio callback function; no Python objects can be used. Because the MPF MC is a Python application, a Python extension library is the only choice in which to use the GStreamer, SDL_Mixer and SDL2 libraries. Since the extension library utilizes both Python and C objects, the GIL needs to be managed in the audio library along with thread protection to avoid race conditions and deadlocks. These design constraints led to the choice of using Cython (http://cython.org/) as the language to implement the MPF MC audio library. Cython is a superset of the Python language that additionally supports calling C functions and using C types, an ideal choice for wrapping external C libraries and using them in a Python application.

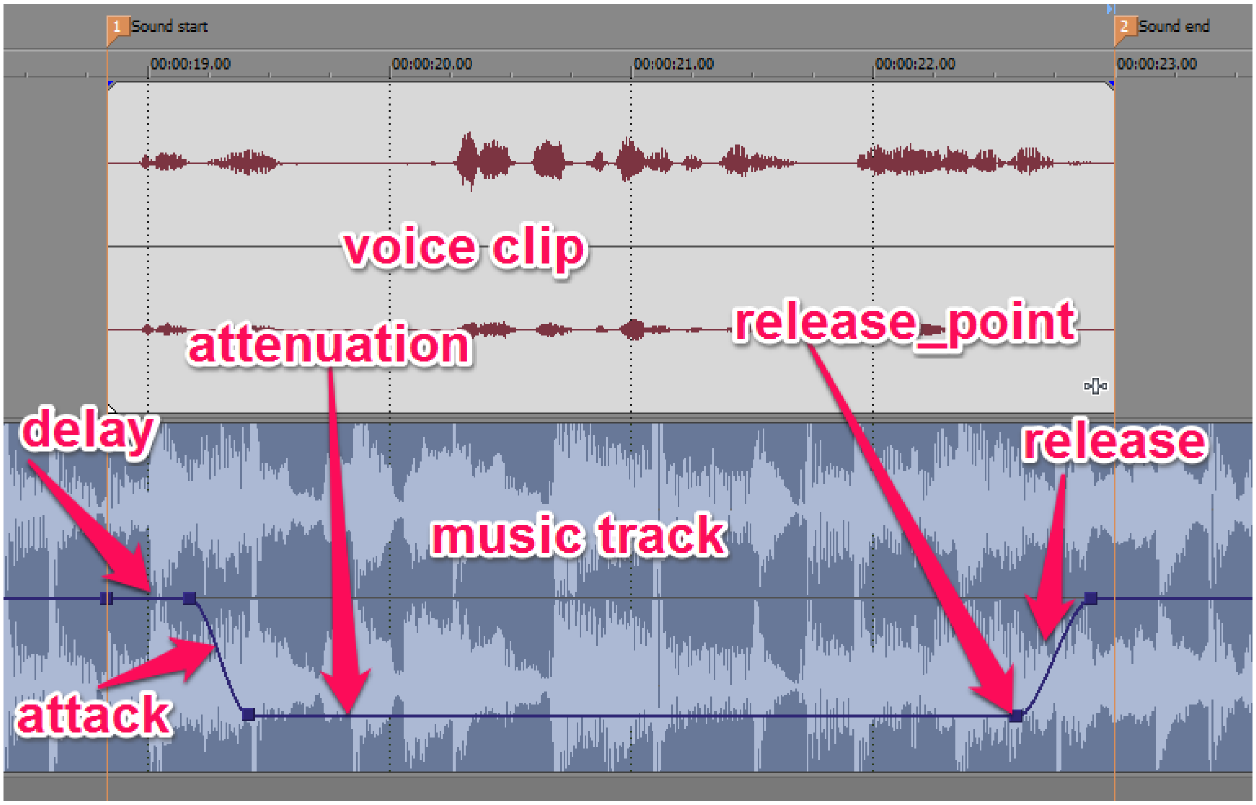
Sounds are MPF assets and are created by the MPF asset loader. The actual sound loading code is contained in the audio library and is performed by SDL_Mixer and GStreamer. A Python container object wraps the C object returned by the loading process. This wrapper allows the sound data to be managed by a Python object. The audio library extracts the C object when necessary and passes it to the audio thread where it can be used to generate audio.

The *sound_player:* enables MPF events to trigger sound actions, such as play, stop, and stop looping. It is a config_player and runs as a plug-in in MPF and also creates event handlers in MPF MC. The audio library also generates MPF events for various sound events (sound played, stopped, looping, etc.) and sends them to MPF via BCP.

Ducking

Ducking is an audio effect that lowers the level of one audio signal based upon the level of another audio signal (one sound "ducks" out of the way of another). It is used to allow particular sounds to be heard more clearly when there is other audio playing at the same time. In the context of a pinball machine, a common use of ducking is to lower the volume of the background music while an important callout is played (such as "Extra Ball!") and then return the volume when the callout is finished. When done professionally, you should not really be able to notice that the music volume is being lowered, but you'll be able to hear the callout prominently.

By default ducking is not enabled for any sounds in MPF. Ducking settings can be optionally set for each sound asset in the machine. To best illustrate ducking and its parameters, here is a diagram:



The voice clip in the top track of the diagram illustrates a callout that we wish to add ducking settings to. The bottom track is playing music. The following parameters control the ducking behavior of the voice clip:

- `target` - The track name to apply the ducking to when the sound is played. In the example above the *music* track is the target.
- `delay` - The duration to delay after the sound starts playing before ducking starts. This value may be specified as a time string or a number of samples.
- `attack` - The duration of the period over which the ducking starts until it reaches its maximum attenuation (attack stage). This value may be specified as a time string or a number of samples.
- `attenuation` - The attenuation (gain) to apply to the target track while ducking. This controls how quiet to make the target track while the sound is playing.
- `release_point` - The point relative to the end of the sound at which to start the returning the attenuation back to normal (release stage). This value may be specified as a time string or a number of samples. A value of 0.5 seconds means to begin to release the ducking 0.5 seconds prior to the end of the sound.
- `release` - The duration of the period over which the ducking goes from its maximum attenuation until the ducking ends (release stage). This value may be specified as a time string or a number of samples.

Ducking settings are specified for each desired sound in the *sounds:* section of the configuration files. It often takes some trial and error to get the ducking parameters set just right for each sound.

Tracks

The audio system in MPF is very similar to a sound mixing board that you control via configuration settings and events. It is divided into tracks (similar to channels on a mixer), each of which has its own properties such as name and volume. With the release of MPF 0.50, there are now multiple types of audio tracks supported by the audio system, each with specialized features.

Track types

The following types of audio tracks are available in MPF:

- `standard` - Standard audio tracks are the most commonly used and have a variety of playback features to support most pinball audio needs. Standard tracks have a setting to limit the number of sounds that may be played simultaneously. If a standard track is busy playing its limit of simultaneous sounds, pending sounds can be added to a queue where they wait to be played until the track can play them. Several settings control a sound's behavior when a track is busy. *Sounds* are audio assets and can be played by standard tracks.
- `sound_loop` - New in MPF 0.50, sound_loop tracks are optimized for live looping music control driven by events. This specialized track type can synchronize playback of multiple looping sounds simultaneously in layers and supports gapless switching to a new set of loops. Sound loops are designed to build music that dynamically changes based on events in your game. Sound used in sound_loop tracks must be loaded in memory (streaming sounds are not supported). Sound loop tracks use *sound_loop_sets* which are special groups of *sounds* to control the playback and looping of audio files.
- `playlist` - New in MPF 0.50, playlist tracks provide a comprehensive set of music playing capabilities that include named playlists (lists of sound assets), playback mode (sequential or random/shuffled), crossfades between songs/playlists, and more. Playlist tracks use *playlists* which contain a list of *sounds* (audio assets) video or audio files that can be played back sequentially or in random order and can be set to repeat or stop after all sounds have been played.

Note: All tracks can be a *ducking* target regardless of the type of track.

Example track configuration:

```
sound_system:
  buffer: 2048
  frequency: 44100
  channels: 2
  tracks:
    music:
      volume: 0.5
      simultaneous_sounds: 1
      events_when_stopped: music_track_stopped
      events_when_played: music_track_played
      events_when_paused: music_track_paused
    sfx:
      volume: 0.4
      simultaneous_sounds: 8
```

(continues on next page)

(continued from previous page)

```
      preload: true
    voice:
      volume: 0.6
      simultaneous_sounds: 1
      preload: true
    loops:
      type: sound_loop
      volume: 0.6
    playlist:
      type: playlist
      volume: 0.6
      crossfade_time: 2s
```

How to setup sound for your machine

This guide explains the basic steps to setup sound for your machine. Sound support is part of the MPF media controller and only available if you're using MPF-MC for your media controller. Please ensure your system is properly setup to play sound (drivers are installed and configured) before proceeding with this guide.

1. Configuring the *sound_system*

The first step in the process of setting up sound for your machine is to setup the `sound_system:` section of your machine configuration file (see *sound_system:* for more detailed information). Generally you can just use the default values for the settings in the section. However, you do need to define the tracks the sound system will use. Tracks can be thought of as channels on an audio mixer with their own volume and other settings. The example below shows a typical pinball machine sound setup with three tracks: *music*, *voice*, and *sfx*. The `simultaneous_sounds:` setting controls how may sounds may be played at the same time on each track. It is recommended that you only allow one music and one voice clip to be played at a time and that many sound effects (sfx) can be played simultaneously so that is what we have configured in the example below.

Example:

```
sound_system:
  tracks:
    music:
      type: standard
      simultaneous_sounds: 1
      volume: 0.5
    voice:
      type: standard
      simultaneous_sounds: 1
      volume: 0.7
    sfx:
      type: standard
      simultaneous_sounds: 8
      volume: 0.4
```

2. Configuring your sound asset folders

The next step is to configure your sound asset folders. First you will need to create a folder named sounds directly under your machine folder. The recommended way to organize your sound files is to create sub-folders for each track in the sounds folder (`music`, `sfx`, and `voice`). If you are going to be using a lot of sounds you can create as many sub-folders beneath each track folder as you like. It can help you stay organized and be able to locate your sounds.

File system directory structure example:

```
machine_folder
    sounds
        music
        sfx
        voice
```

Now that our sound asset folders have been created, it's time to let MPF know where to look for sound files when it starts and what basic settings to apply to each sound it finds. This is done by adding a `sounds:` section to the `assets:` section in our machine configuration file. The example below illustrates what this should look like in your machine configuration file. The `default:` setting contains the default settings that should be applied to all sound assets. In this example below, `load:` should be assigned a value of `on_demand` for all sound assets. Next we enter a setting for each sub-folder located in our sounds directory and specify the settings we want applied to each sound asset found in those sub-folders. In our case we have created sub-directories for each track and want the sounds contained in them to play on their respective tracks (*music*, *sfx*, and *voice*) so we set the `track:` setting accordingly.

`assets:` section in machine configuration file:

```
assets:
  sounds:
    default:
      load: on_demand
    music:
      track: music
    sfx:
      track: sfx
    voice:
      track: voice
```

When your machine launches, the asset manager will now search for supported audio files in the specified directories and assign the proper settings to each file it finds. We're well on our way to actually hearing some sound!

3. Put some sounds in your sound folders

You probably don't need much assistance with this obvious step, but let's go through the process anyway just in case. As of version 0.33, MPF supports 16-bit .wav (Wave), .ogg (Ogg Vorbis), and .flac (FLAC) audio files (we hope to add other formats in future releases such as .mp3). Locate some supported audio files and place them in the appropriate track folders that you created in the previous step (a good site to find free public domain sounds is www.freesound.org). Put all music files in the `music` folder, voice callouts in the `voice` folder, and all other sound effects in the `sfx` folder. You are welcome to create any sub-folders you desire and put sounds in them to help keep things organized.

4. Setting the default master volume level

The master volume (applied to all tracks in the sound system) can be adjusted from the service switches or custom events. MPF stores the master volume level as a machine variable, so the selected volume will persist each time the game boots up.

The master volume ranges from 0.0 (silent) to 1.0 (full), and defaults to 0.5 (50%). You can set your own default volume by overriding the machine variable settings in your machine config file.

```
machine_vars:
  master_volume:
    initial_value: 0.25   # Set this to any value you want
    value_type: float
    persist: true         # If false, the volume will reset to default
                          # each time the machine boots up
```

5. Additional configuration for selected sounds

Now when you start your machine you will have some sounds available (assuming you placed some supported sound files in your sounds folder during the last step) and they will all have some very basic default settings. It is very likely that you won't be happy with the default settings for all of your sounds so let's create some more tailored settings for a few of them.

Renaming some sounds

Your sounds now all have names based on their file names (without the extensions), and by default that is how they must be referenced in your config files. Perhaps some of your file names are either a bit cryptic or contain additional text that you'd like to shorten. One option is to simply rename any files you'd like in the operating system. Another option is to setup some configuration options in your config files to reference the sound file by a different name which is what we will do next.

I downloaded a triangle sound from www.freesound.org that has an undesirable filename: `22783__franciscopadilla__80-mute-triangle.wav`. I would rather just refer to it in my config files as `triangle` and not `22783__franciscopadilla__80-mute-triangle` (which is what it will be by default). In my `sounds:` section of my machine configuration file (see *sounds:* in the documentation for more details) I can put the following text:

```
sounds:
  triangle:
    file: 22783__franciscopadilla__80-mute-triangle.wav
```

That simple configuration change will allow the sound as to be referred to as `triangle` wherever you refer to that sound in other configuration locations. *Note*: be sure to include the complete file name, including the extension when using the `file:` setting.

Setting the volume of a specific sound

A very common adjustment to make is to set the volume for each and every sound you load in your machine. This allows you to balance out sounds from various sources rather than trying to adjust the levels in each sound file using audio editing software. Building on the example above, let's set the volume of the *triangle* sound in our config file:

```
sounds:
  triangle:
    file: 22783__franciscopadilla__80-mute-triangle.wav
    volume: 0.85
```

`volume:` controls the volume of the sound and works in conjunction with the track volume and the master volume. Volume can either be entered as a number between 0.0 and 1.0 or as a decibel level (see *Instructions for entering gain values)* for more information). You will probably have to spend some time adjusting the volumes of many sounds in your machine to get everything to sound just the way you want it.

Note: If you hear distortion in your sounds when they are played back in a mix, be sure to try lowering the volume as you may be experiencing clipping.

Other sound settings

There are many other settings you may wish to change for some sounds in your machine.

- How do you cause your sound to loop 3 times every time it is played? Add `loops: 3` to the config section for your sound. How do you loop a sound indefinitely? Add `loops: -1`.
- How do you adjust the which sounds can preempt other sounds and how long a sound may wait to be played before it is discarded? Use the `priority:` and `max_queue_time:` settings.
- How do you send events to MPF when a sound begins or finished playing? Use the `events_when_played:` and `events_when_stopped:` settings.
- What about ducking? Just what is it anyway? Learn about *ducking* in the documentation.

The documentation for the *sounds:* configuration section contains further information about all these settings.

Example `sounds:` configuration demonstrating most common settings:

```
sounds:
  triangle:
    file: 22783__franciscopadilla__80-mute-triangle.wav
    volume: 0.85
    max_queue_time: 0
  laser:
    volume: 0.5
    loops: 3
    max_queue_time: 0
  extra_ball:
    file: extra_ball_12753.wav
    events_when_started: extra_ball_callout_started
    events_when_stopped: extra_ball_callout_finished
    volume: 0.8
    priority: 50
    max_queue_time: None
    ducking:
      target: music
      delay: 0
      attack: 0.3 sec
      attenuation: 0.45
      release_point: 2.0 sec
```

(continues on next page)

(continued from previous page)

```
    release: 1.0 sec
  slingshot_01:
    volume: 0.5
    max_queue_time: 0
  song_01:
    volume: 1.0
    priority: 100
    about_to_finish_time: 2s
    events_when_about_to_finish: song_01_about_to_finish
```

6. Hooking up an MPF event to play a sound

Now that your sounds have been setup and are available in your machine, the next step is to configure them to be played. The sound player was designed to do just this (associate a sound action, such as play or stop, with an MPF event). The sound player can be configured in either the machine configuration file, a mode configuration file, or even in a show step (or in all of them). To keep things simple here, let's configure the sound player in the machine configuration file.

The scenario in this example is we want our song from the previous example (`song_01`) to play infinitely when the *attract* mode starts and stop when the *attract* mode stops. Create the following entries in the `sound_player:` section of the machine config file:

```
sound_player:
  mode_attract_started:
    song_01:
      action: play
      loops: -1
  mode_attract_stopped:
    song_01:
      action: stop
```

That's it. The `song_01` sound will be played on the music track whenever *attract* mode is started and will stop whenever *attract* mode is stopped. The `mode_attract_started` section refers to a standard MPF event that is sent whenever a mode named *attract* is started and `mode_attract_stopped` is a standard MPF event that is sent whenever a mode named *attract* is stopped. For more information, see the *sound_player:* documentation.

Finished

Congratulations! You have completed your the basic sound system setup and should have some simple audio playing in your machine.

References

- *Sound & Audio*
- *Ducking*
- *Tips & tricks*
- *sound_system:*

- *tracks*
- *sounds:*
- *sound_player:*
- *sound_loop_sets:*
- *sound_loop_player:*
- *playlists:*
- *playlist_player:*
- *Instructions for entering gain values*

Sound & Audio Tips & Tricks

This page contains a collection of miscellaneous tips and tricks when working with the sound & audio features in MPF.

Common Digital Audio Terms

Bit Depth The number of bits used to represent and store a single sample. Bit depth (also commonly referred to as sample resolution) determines the number of possible levels that can be captured during digitalization. 16-bit represents 65,536 (2 to the 16th power) possible values. The MPF-MC audio library only supports 16-bit audio files.

Brickwall limiter A brickwall limiter is used to ensure an audio signal does not exceed a certain threshold. Any input value exceeding the threshold is set to the threshold value. This is used in the mixing engine to ensure 16-bit integers do not exceed their maximum value and wrap around (which adds ugly sounding distortion). It is important to set sound and track volume levels properly in order to avoid the clipping that brickwall limiting adds when mixing signals that are too loud.

Ducking Ducking is an audio effect that lowers the level of one audio signal based upon the level of another audio signal (one sound "ducks" out of the way of another).

FLAC Free Lossless Audio Codec (FLAC) is an audio file format which allows digital audio to be losslessly compressed such that file size is reduced without any information being lost.

Normalization Normalization is the process of changing an audio recording's overall volume by a fixed amount to reach a target level.

Ogg Vorbis (OGG) A free and open-source audio coding format. Ogg Vorbis is a lossy compressed audio file format.

Sample Rate The number of samples per second taken from a continuous signal to make a discrete signal. A common example of sample rate is CD audio which is recorded at 44,100 Hz (44,100 samples per second).

WAV Waveform Audio File Format is a Microsoft and IBM audio file format standard for storing an audio bitstream on computers. WAV is a lossless uncompressed audio file format.

Preparing your sound files for use in MPF

The custom audio library supports several audio file formats for sounds, however only 16-bit audio files can be used (this is a SDL_Mixer limitation). If you have sounds that do not have a bit depth of 16-bits then you must use audio editing software to resample your files to 16-bits. Audacity is a full-featured, free, open source, cross-platform audio software application for recording and editing. It is fully capable of performing all the necessary steps to prepare your audio files for use in MPF.

Tip: Do all your audio editing in a lossless audio format (such as WAV) and preserve a master copy in that format as well in case you may want to perform any future editing. Convert a copy of all your sound files to your machine's sample rate and bit depth (ex: 44,100 Hz 16-bit) for use in MPF. Use WAV format for the fastest loading sound files and Ogg Vorbis for the smallest file sizes (if storage space is at a premium).

Here is a typical workflow for preparing your sound files:

1. Make a backup copy of all your original sound files (in case you accidentally mess one up while editing it you'll have a backup).
2. Trim silence from the beginning and end of your sound files. Removing the silence from the file will make the sound feel more responsive and will take up less memory.
3. Normalize your sound files using peak normalization to 0 dB. This will maximize the volume level of your sound files without clipping.
4. Resample all your audio files to a bit depth of 16-bits and use the sample rate you will be using in your machine (a typical sampling rate is CD quality 44,100 Hz). This will save processing power when loading your samples are no resampling will need to take place.
5. Save the files in your desired file format (WAV is recommended for loading speed).

Your files are now ready to begin using in MPF. The read the other basic steps for setting up sounds, see *How to set up sound*.

Review `max_queue_time` Settings for Long Sounds/Music

The `max_queue_time` settings for sounds can lead to some unexpected behavior, especially for longer sounds (like music). This setting specifies the maximum time a sound can be queued before it's played. On a track that supports only a single sound at a time (like a typical music track), playing a sound with a priority that is less than or equal to the currently playing sound will have to wait until the current sound is finished (it will be added to the queue). That may be acceptable to you, but you may also be surprised when you hear the sound a minute or two later.

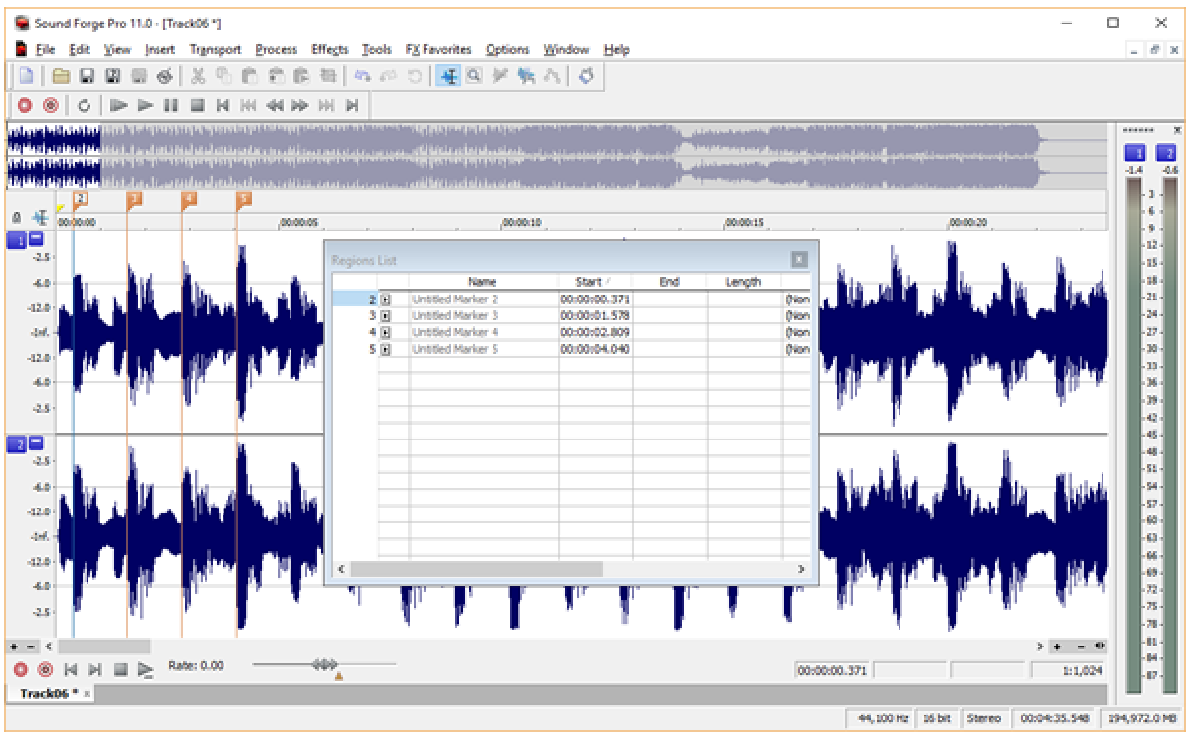
It is suggested you review all your `max_queue_time` settings to make sure they make sense for the sound and situation in which they will be played. The default setting of `None` means the sound will eventually be played, no matter how long the wait in the queue is. A value of `0` specifies the sound will be immediately discarded if the track is already busy playing its maximum number of sounds. A value of `2 secs` specifies the sound will wait in the queue for 2 seconds to be played before being discarded. Sound effects for things like slingshots and pop bumpers probably don't make much sense if they are played more than 250 milliseconds after they are hit so setting `max_queue_time` to a value between `0` and `250 ms` is recommended. On the other hand, an extra ball callout is probably fine to play a few seconds after the ball is earned. Go through your sounds and consider how to set this setting for each one.

For more information, see the *sounds* documentation.

Synchronizing Sound With an LED Show

The key to synchronizing an LED show with a music track is to determine at what times in the sound file you want events (such as LED color changes) to occur. There are many ways to do this, but here are a few suggestions:

- Use your favorite sound or editing software to open your music track and place markers in all the locations where you want LED changes to occur. This may take some trial and error and listening to portions of your music over and over again until you get it right. Once your markers are in place, export them to a text file (if your software supports it), or write down the times of each marker. Use the times as step times in your show and assign the LED settings you want in each step. This is a bit of a tedious process, but should give you nice synchronization when the show is played at the same time as the music track (you can even put the sound play action in the first step of your show). I work on a PC and use Sony Sound Forge for sound editing, but there are many good editors available on every platform that support inserting markers. Here is a screenshot of the process in the editor I use:

  This feature is also available in Audacity (free open-source cross-platform sound editing software) and many video editing packages.

- As an alternative, you can determine the tempo of your song in beats per minute (BPM) and from that number calculate the time for each beat. Once you have the time for each beat, you can use it to calculate various show step times (assuming you want LED changes to occur on the beat). There are some tools out there that will calculate the BPM of your song for you, but are not always very accurate depending upon the content of your song.

For more information on creating shows for your LED, see the *Shows* documentation

Pausing Background Music While a Video is Playing

With the addition of the new *track_player* config player in 0.32, it is now possible to control audio tracks using MPF events. One common use of this new functionality is to pause your music track while you play a video and resume the music when the video is finished playing.

The basic concept is to add an event to the video that is triggered when the video is played and one when the video is stopped. Those events are then added to the `track_player` section of your config file:

```
track_player:
  my_video_is_playing:
    music:
      action: pause
      fade: 1 sec
  my_video_has_stopped:
    music:
      action: play
      fade: 1 sec
```

That's all there is to it. Now whenever the `my_video_is_playing` MPF event is posted, the music track will be paused. It will be resumed when the `my_video_has_stopped` MPF event is posted.

When Two Drop Targets Are Hit Simultaneously How Do I Keep Two Sounds From Playing

A common scenario with drop targets is to play a sound when each target is hit. Frequently a player will hit two targets with a single shot dropping them both virtually at the same time. In this situation playing a sound for each target is not always desired. Instead, it would be nice to only have a single sound played when the targets are hit within a short time window.

One possible way to solve this in MPF is to use *counters*. Counters have a *multiple_hit_window* setting that prevents accidental double hits within the configured time period. Instead of using the target hit event to trigger the sound, the target hit event will trigger the counter which in turn will post a hit event that can be used to trigger the sound. Here is an example:

```
coils:
  reset_drop_targets:
    number: 1

switches:
  sw_drop_target_1:
    number: 1
  sw_drop_target_2:
    number: 2
  sw_drop_target_3:
    number: 3

drop_targets:
  target_1:
    switch: sw_drop_target_1
    reset_coil: reset_drop_targets
  target_2:
    switch: sw_drop_target_2
    reset_coil: reset_drop_targets
  target_3:
```

(continues on next page)

(continued from previous page)

```
    switch: sw_drop_target_3
    reset_coil: reset_drop_targets

counters:
  drop_target_counter:
    count_events: drop_target_target_1_hit, drop_target_target_2_hit, drop_target_target_2_hit
    multiple_hit_window: 500ms
    events_when_hit: drop_target_counter_hit

sounds:
  drop_target_sound:
    file: blip1.ogg
    volume: 0.75

sound_player:
  drop_target_counter_hit:
    drop_target_sound:
      action: play
```

Alternatively, you could also define a separate track which allows only one concurrent sound at a time. See *Tracks* for details.

How to play a sound with variations

One of the ways to make your machine more professional is to use different variations of sounds in your machine. This will add variety and make your audio less predictable and more "alive". This guide explains how to play a sound with multiple variations in your machine. Sound support is part of the MPF media controller and only available if you're using MPF-MC for your media controller. This guide assumes you have already configured your sound system for your machine and are familiar with the basic sound setup concepts. If not, please start with the *Setting up sound for your machine* guide first.

1. An brief introduction to sound pools

Sound pools allow you to group multiple sounds together and treat the pool as a single sound. Each time a sound pool is played, it selects a sound from its group of sounds. The selection can be configured to be random or in a particular sequence. For more complete information, please read the *sound_pools* documentation.

Although sound pools can be used to play a random music track or random callout when an event occurs, in this guide we will be using a sound pool to play variations of a sound when a slingshot is hit.

2. Add a sound and some variations

Before we can create our sound pool, we first need to configure the individual sounds that will make up our pool. We've decided we want to have a small ding (like a triangle hit) play whenever the slingshot is hit. Let's start by adding our basic sound to our sound assets. The hardest part of this process is to either generate or find the sound you want (we won't go into that process here). Once you have your sound file, put it in the appropriate sound asset folder. I found a simple triangle sound on www.freesound.org that we'll use here, *13147__looppool__triangle1.wav*. Place the file in your sound effects track folder (`<machine_folder>/sounds/sfx`). Now we'll add it to your machine configuration file, but give it an easier name to remember (`triangle_01`) using the `file:` setting (or you could simply rename the file to *triangle_01.wav* and omit the *file:* setting):

```
sounds:
  triangle_01:
    file: 13147__looppool__triangle1.wav
    volume: 0.7
```

Now add a few variations of the sound. I used my favorite sound editor to slightly adjust the pitch and frequency content of the triangle sound file, creating three variations. You can also just find some other similar sounds on the internet. After you have your variations, place them in the same directory as your first sound file. We are now ready to add them to the `sounds:` section in the machine configuration file (I named the sound variations *triangle_02*, *triangle_03*, and *triangle_04*:

```
sounds:
  triangle_01:
    file: 13147__looppool__triangle1.wav
    volume: 0.7
  triangle_02:
    volume: 0.7
  triangle_03:
    volume: 0.7
  triangle_04:
    volume: 0.7
```

3. Configure the sound pool

We now have 4 variations of the same basic triangle sound. It's time to put them all into a single sound pool object so we can treat them as a single sound. To do so, we need to add a `sound_pools:` section to our machine configuration file as follows:

```
sound_pools:
  triangle:
    type: random
    sounds:
      - triangle_01
      - triangle_02
      - triangle_03
      - triangle_04
```

We now have a sound pool asset called `triangle` that acts just like a sound asset, except that each time `triangle` is played, one of the 4 sound variations contained in the sound pool will randomly be selected to be played. Want to add more variations or take one out? It's just as simple as modifying the list of sounds in the sound pool.

This is great, but let's adjust the sound pool settings a bit to fine tune its behavior. We really want the main sound (`triangle_01`) to be played more often than the other sounds. How can we make that happen? It's very easy to do. We can add weights to each sound in the pool that specify the probability of each sound being selected. Let's look at our `sound_pools:` section again:

```
sound_pools:
  triangle:
```

(continues on next page)

(continued from previous page)

```
    type: random
    track: sfx
    sounds:
      - triangle_01|5
      - triangle_02|2
      - triangle_03|2
      - triangle_04|1
```

Notice we've added a pipe character (|) to the end of each sound followed by a numeric value. These values assign a relative weight to each sound that will be used in the random selection process. `triangle_01` has a relative weight of `5` out of a total weighting of `10` (simply add all the weight values), therefore its probability of being selected is `50%`. The `|1` appended to `triangle_04` is unnecessary because a relative weight of `1` is the default value for all sounds in the pool that do not have explicit weight values assigned.

Sometimes you may want to have sounds included based on conditional events. You can add a condition to any sound and the sound pool will only include that sound if the condition evaluates to true at playback time. If the selection is random, excluded events will not be weighted in the distribution. If the selection is sequential, excluded events will simply be skipped.

```
sound_pools:
  triangle:
    type: random
    track: sfx
    sounds:
      - triangle_01
      - triangle_02{current_player.triangles_found>1}|2
      - triangle_03{current_player.triangles_found>2}
      - triangle_04{device.achievements.supertriangle.state=="complete"}|5
```

Sound conditions are formatted the same as all *conditional events*. Any sound in a pool can have a weight, a condition, both, or neither.

For additional sound pool setting options, take a look at the *sound_pools* documentation.

4. Configuring the sound player

We have our sounds and sound pool configured. To trigger the sounds with MPF events, the sound player can be used. The sound player was covered in the previous tutorial and will not be covered again here. You can also read the *sound_player* documentation.

CHAPTER 12

Shows

In MPF, *shows* are containers that hold steps of instructions for things that can be "played" in a certain order with specific timings.

You can do almost anything in a step in a show, including setting the color of LEDs, playing sounds, showing slides on the display, posting events, firing drivers, etc.

You're going to use shows a lot.

Note: Prior to MPF 0.30, "light shows" and "display shows" were two independent things. In MPF 0.30+, shows are now universal. There's only one type of show, and it can be used to do anything.

Shows are controlled and run by the MPF game engine, and if a show contains actions in a step for the media controller, such as display or sound actions, then those actions are sent via BCP to the media controller when that step is played.

Shows are configured via the YAML formatting just like config files. You can add the definitions for simple shows into your config files directly, or you can create standalone shows files that you store in your machine's'shows' folder.

It is totally viable to create simple shows by hand. However, there is a *MPF Showcreator* to create complex light shows.

Read on for more info on how shows work:

Show configuration format

Shows are defined via nested key/value pairs in YAML files.

A show contains multiple steps, and each step contains a time (for when that step should run) and instructions (for what actions should happen in that step).

Here is a very simple show with two steps. The first step sets the color of *led1* to *red*, then one second later, it turns *led1* off again. Then after another second, the show is over. (Most likely you'd configure a show like this to *loop*, meaning this should could be used to flash *led1* on and off.)

```
##! show: my_show
- time: 0
  lights:
    led1: red
- time: +1
  lights:
    led1: off
- time: +1
```

There are *lots of different actions you can configure in a show step* (LEDs, lights, sounds, coils, display slides, etc.), but for now we'll just use this very simple show as an example.

Defining steps

Shows are configured via YAML-like format, just like config files.

In the example show above, note that each *step* in the show starts with a key/value pair that's separated with a dash, then a space. So you could say that the example show above has three steps:

Step 1:

```
##! show: my_show
- time: 0
  lights:
    led1: red
```

Step 2:

```
##! show: my_show
- time: +1
  lights:
    led1: off
```

Step 3:

```
##! show: my_show
- time: +1
```

Important: YAML formatting can be tricky. It's important that you include a space between the dash and the key name. `-time: 0` will not work and give you an error (since there is no space between `-` and `time`.). Also, make sure the individual setting names are all aligned vertically. (In the example above, *time:* and *lights:*) are left-aligned.

Setting step time

The `time:` setting in each step represents the time when that step *starts*. The first step will always be `time: 0`

If you just enter a number for the *time*, that number represents seconds. However, you can enter the time in *standard MPF time format*, which could be *ms*, *secs*, etc. The following are all valid *time* entries:

- `time: 1` (1 second)
- `time: 1.0` (1 second)
- `time: 1s` (1 second)
- `time: 1000ms` (1 second)

If you do not enter a `time:` setting for a step, MPF automatically uses `time: +1`.

When shows are played, it's possible to specify a *speed* setting which is a multiplier for how fast the show is played. The default is `1.0` which would use the time values entered here, but keep in mind that it's possible to play a show back at any speed. You can even change the speed of a running show while it's in progress.

Tip: The precision of shows is limited to clock speed that MPF runs at. By default, MPF runs at 60fps, which means that each "tick" of MPF is about 16ms. So in that case, you can't get resolution of shows more precise than that.

Absolute time

The time value for each step indicates when this step will play measured in time since the start of the show. This is useful if you're synchronizing show steps with sound or video.

Relative time

Sometimes it's more convenient to specify the timing of a step in a show relative to the step before it. To do that, enter the *time* value with a + in front of it, like this:

```
##! show: my_show
- time: +1
```

Relative step times are nice because you can adjust the timing of one step and then all the other relative steps after it are shifted back or forwards automatically.

You can mix-and-match incremental and absolute times in the same show, and you can also combine the plus sign for relative times with seconds or millisecond values. For example:

```
##! show: my_show
- time: 0   # plays right away, at 0 seconds
  # ...
- time: +1  # plays 1 sec after the previous, 1 sec after show start
  # ...
- time: +1  # plays 1 sec after the previous, 2 secs after show start
  # ...
- time: 4   # plays 4 secs after show start, 2 secs after the previous
  # ...
- time: +1  # plays 1 sec after the previous, 5 secs after show start
  #...
```

Note that since shows use YAML formatting, you can use the hash sign (#) to add comments which MPF ignores.

Setting step duration

Instead of specifying the "time" when a step starts, you can also specify the "duration" of how long a step lasts (which is essentially specifying when a step ends). The difference is subtle, but each is useful in different situations.

For example, the following to shows are identical:

```
##! show: my_show
- time: 0
  lights:
    led1: red
- time: +1
  lights:
    led1: off
- time: +1
```

```
##! show: my_show
- duration: 1
  lights:
    led1: red
- duration: 1
  lights:
    led1: off
```

You can also mix and match "time" and "duration" settings in the same show (and even in the same step). The only thing you can't do is have a "time" setting in a step that follows a step with "duration" (since those two values would essentially mean the same thing and it would be confusing).

Setting the duration of the final step

Most people find it easiest to just use either "time" or "duration" consistently throughout a show. The only practical difference you need to think about is how the final step works.

For example, with "time"-based steps, you're specifying the time when a step starts. So when does a step stop? When the next one starts. But what about your last step in the show? How long should it run for? If you just use time-based steps, you'd still want to specify a "duration" for the final step, like this:

```
##! show: my_show
- time: 0
  lights:
    led1: red
- time: +1
  lights:
    led1: green
- time: +1
  duration: 1
  lights:
    led1: blue
```

"Holding" the final step

You can set a `duration: -1` for an "infinite" duration of a step. (Think of this like a hold or pause.) This is most useful in shows that you want to run and then hold something in their final state. For example, maybe you want a show that runs once (no loop) and flashes a light which then stays on. You could do that like this:

```
##! show: my_show
- time: 0
  lights:
    led1: red
- time: +250ms
  lights:
    led1: off
- time: +250ms
  lights:
    led1: red
- time: +250ms
  lights:
    led1: off
- time: +250ms
  lights:
    led1: red
  duration: -1
```

In this example, the LED would stay on (red) until that show was manually stopped or until the mode was stopped (if the `show_player:` entry was in a mode config file).

What can you put in shows?

In the *Show configuration format* page, we showed how *time* values work in shows and included some simple examples using *lights*. However in MPF, you can put almost anything in shows, including:

- Lights
- Coil & drivers
- Sounds
- Slides (for the display)
- Shows (one show can spawn other shows and/or act like a playlist)
- Events
- Random events (randomly post an event from a list of events)
- Flashers
- GI (general illumination)
- BCP commands & triggers
- Widgets (to be added or removed from slides)

The full gamut of options for each of these things is available to you in a show step.

For example, you can configure lights to change color, set their fade, turn off, etc. You can show slides on your display or DMD, or remove existing slides. You can post events that trigger other shows or other things to happen. You can start and stop sounds and music. The list goes on and on...

Technically-speaking, the list above is actually a list of things that MPF calls *config players*.

Config players in MPF have nothing to with the actual human players of your machine, rather, they are things that "play" configurations.

Config players are used in the `*_player:` section of your config files *and* as steps in shows. For example, the *light player* is used to "play" a config to lights, and it's available to you outside of shows in the `light_player:` section of your config file as well as in the `lights:` section of a show.

That naming convention is the same for all the config players. You play sounds via the `sound_player:` section of a config file or the `sounds:` section of a show. Slides are played via the `slide_player:` section of a config file or the `slides:` section of a show, etc.

All of the individual config players are documented in the *config players* section of the documentation. You can read details about each config player there, as well as specific instructions for how to include that kind of player in a show.

Creating standalone show files

You can create a subfolder called *shows* in your machine config folder or within a mode config folder. Then inside that folder, you can create separate files, where each file is its own show. The files need to have a `.yaml` extension, and the name of the file before the extension is the name of the show as you'd refer to it in your MPF configs.

A few notes for creating show files:

- MPF config files are case sensitive. It is best to stick to some kind of convention (such as lowercase names with underscores).
- Show names are "machine-wide" within MPF. This means that if you have two different shows with the same name in different locations, MPF will get confused.
- Valid characters for show names are z-x, 0-9, and the underscore. Python objects cannot contain dashes in their names, meaning your show file names cannot include dashes.

Here is a sample show file. This file might be called something like `flash_red.yaml` and would be located in your machine's `/shows` folder:

```
##! show: my_show
#show_version=5
- time: 0
  lights:
    led1: red
- time: +1
  lights:
    led1: off
- time: +1
```

Notice it's essentially the same show we used as an example in the section on show config formats. However there's one important change.

Since this is a standalone show file, we need to tell MPF what "version" of the show format this file is. MPF versions 0.56 use `show_version=5`. If we ever change something in the show format, then we'll increment the version. (Don't worry though, we have and automated migration tool that converts shows to the new formats. That's actually part of the reason we include the show_version in the show files)

The bottom line is that when you create a .yaml show file, the first line of the file must be `#show_version=5` so MPF knows it's working with the proper type of file.

Beyond that, the show file follows the show format covered elsewhere in this documentation. You can nest show files into subfolders under the `/shows` folder if you want to, and in can put `/shows` folders in both your machine-wide and mode-specific folders. (The `/shows` folder should be in the root of your machine config or the root of a mode folder. It does *not* go inside the `/config` folder.)

Creating shows in config files

In addition to being able to *create standalone show files*, MPF also lets you define your shows right in-line in your config files.

You can do this in the `shows:` section of a config file. (This can be done in a mode-based config or in your machine-wide config).

The actual format for a show in a config file is identical to the format of a standalone show file on disk. Basically you add a `shows:` section to a config, create sub-sections based on show name, and then add normal show items to the config. For example:

```
shows:
  flash_red:
    - time: 0
      lights:
        led1: red
    - time: +1
      lights:
        led1: off
    - time: +1
  blue_green_cycle:
    - time: 0
      lights:
        led2: blue
    - time: +1
      lights:
        led2: green
    - time: +1
```

The section above contains two shows: *flash_red* and *blue_green_cycle*.

Shows in files versus shows in configs

Now that you see it's possible to create shows as standalone YAML files in your *shows* folder and also in a *shows:* section of a config file, you're probably wondering what the difference is and when you should use one versus the other?

The answer is pretty simple: There is no difference.

When MPF boots up, it creates the shows objects from your show files and the show sections from configs. But once those shows are created, they are identical. No difference whatsoever. So really you can uses whichever format you want (or mix and match them). We typically create bigger and more complex shows as their own YAML files, and smaller, simpler shows in-line in the machine or mode config. But again, it really doesn't matter.

The only real difference is that if you load shows from YAML files, you can dynamically load and unload shows throughout the lifespan of MPF. (For example, you might configure it so a mode loads the shows it needs into memory when the mode starts, and then unloads them when the mode ends.) If you have lots and lots of shows and not very much memory, this could help conserve memory since shows are only loaded when they're needed. That said, individual shows don't take up too much memory (certainly far less than sounds and images), so in most cases this is probably moot.

One "gotcha" to keep in mind is that MPF maintains a global list of shows, so you can't have the same show name twice (even if one is loaded from a show file and one is in a config file). If you do this, then whichever show you load last will be overwrite the previous one, and you'll be confused.

Referencing Slides/Widgets in Shows

You can add slides/widgets in shows. However, you cannot reference slides/widgets which were defined in show files or show sections from outside of the show. This worked in the past but it caused issues if this show hasn't been loaded yet. Nevertheless, you can reference (named) slides/widgets from slide/widget sections in any show.

Using "tokens" for run-time variable replacement in shows

One of the most powerful features of MPF shows is that you can build shows that contain "placeholder" tokens which are dynamically replaced with actual values when a show starts.

This lets you build reusable shows that you can then use in lots of different situations with different lights, slides, sounds, etc.

Shows without tokens

To understand how tokens work, let's first look at a show that does not include any tokens, like this:

```
##! show: my_show
- time: 0
  lights:
    led_01: red
- time: 1
  lights:
    led_01: off
```

The example show above is simple. When it starts, it sets *led_01* to red, then 1 second later, it turns it off. You can run this show in a loop to flash *led_01* between red and off.

If you called this show *flash_red*, you could play it via the *show_player:* section of your config, like this:

```
show_player:
  some_event: flash_red
```

The problem with this show is that it's hard-coded. It only works for *led_01*, and it only cycles the colors between red and off.

So what if you want to flash *led_01* between yellow and off? Or what if you want to flash a different LED? With a show like the example above, you'd have to write a new show for every LED with every possible color combination you'd ever want. :(

Adding tokens to shows

This is where tokens come in. Consider a slightly modified version of the show above using a token instead of a hard-coded LED name:

```
##! show: my_show
- time: 0
  lights:
    (led): red
- time: 1
  lights:
    (led): off
```

Notice the second show is identical to the first, except every reference to `led_01` has been replaced with `(led)`.

When MPF plays a show, it looks for words in the show contained in parenthesis, and then it can use those parenthesis to replace values on the fly.

So in the second show here, when you run the show, you could tell it "replace the "leds" token with the value "led_02", which would make a show like this:

```
##! show: my_show
- time: 0
  lights:
    led_02: red
- time: 1
  lights:
    led_02: off
```

The actual way that you start and send tokens to shows varies depending on what you're doing in MPF. (Typically they're tied to shots or events.)

For example, here's how you'd do it via the *show_player:*. (In this example, we also add `loops: -1` which will cause the show to loop (repeat) indefinitely.

```
show_player:
  some_event:
    flash_red:
      loops: -1
      show_tokens:
        led: led_02
```

MPF can run multiple instances of a show at the same time, so you could run the above show multiple times (at the same time), passing different tokens to each one, meaning you could use the same show to flash lots of lights at once:

```
show_player:
  some_event:
    flash_red:
      loops: -1
      show_tokens:
        led: led_02
  some_other_event:
    flash_red:
      loops: -1
      show_tokens:
        led: led_03
```

Putting multiple values into a single token

You can also use tags to insert multiple values into a single token. For example, consider the following section from your machine config:

```
lights:
  led_01:
    number: 00
    tags: tag1
  led_02:
    number: 01
    tags: tag1
```

You can see that both *led_01* and *led_02* have the *tag1* tag applied. So if you play the show above (with the *leds* token), you can actually pass the tag name to the token instead:

```
show_player:
  some_event:
    flash_red:
      loops: -1
      show_tokens:
        led: tag1
```

This would result in a show that was equivalent to:

```
##! show: my_show
- time: 0
  lights:
    led_01: red
    led_02: red
- time: 1
  lights:
    led_01: off
    led_02: off
```

Token names are arbitrary

The token show we've been working with so far includes a token called *leds*. That's a good name for the token since it explains what it's for. However, MPF doesn't care what the actual token name is. All it's doing is a find-and-replace when the show starts with whatever token names it was passed.

For example, this is a perfectly valid show:

```
##! show: my_show
- time: 0
  lights:
    (corndog): red
- time: 1
  lights:
    (corndog): off
```

In this case, you'd just pass a value for the *corndog* token when you play the show:

```
show_player:
  some_event:
    flash_red:
      loops: -1
      show_tokens:
        corndog: led_02
```

Tokens can be values too

You can use tokens anywhere in a show. The actual find-and-replace is pretty simple, just looking for words in parentheses and then substituting them with the tokens key/value pairs that were passed when the show starts.

You can also pass multiple tokens. Consider the following show:

```
##! show: my_show
- time: 0
  lights:
    (led): (color1)
- time: 1
  lights:
    (led): (color2)
```

Notice there are three tokens in this show: *led*, *color1*, and *color2*. You might call this show *color_cycle*, which you could then play like this:

```
show_player:
  some_event:
    color_cycle:
      loops: -1
      show_tokens:
        led: led_02
        color1: green
        color2: blue
```

Tokens vs Tags

Almost all devices support tags. In *config players* such as *light_player* you can also reference multiple lights by their tags.

The bottom line

As you can see, tokens are very powerful. Again, keep in mind there are many different ways to start shows in MPF, and all of them have ways to pass tokens to shows.

Starting & stopping shows

Now that you know how to create shows, how do you start and stop them?

The easiest way is with the *show_player:* section of either a machine-wide or mode config files.

You can use the show player to start, stop, pause, resume, advance, step back, and update shows. (That's a lot!) You can also use it to set the playback speed, set up show synchronization, and set up show repeats and looping.

Note that any shows which were started via a `show_player:` section in a mode config file will automatically be stopped when that mode stops.

So check the *show_player:* documentation for details.

Synchronizing multiple shows

One thing you might notice in professional pinball machines is that all the lights flash in sync with each other. But in MPF, if you have lots of separate shows, then you'll notice they all sort of start randomly when they start, and it looks bad because they're not all perfectly aligned with each other.

MPF solves this by incorporating a "sync_ms" setting when playing shows. When you add this setting to a show and then play it, MPF will not start the show until the next exact multiple of that number from zero.

For example, if you have `sync_ms: 500`, then MPF will start a show at the exact second or half second. (e.g. the seconds value of the current time will either be .0 or .5).

If you have `sync_ms: 250`, then shows will be delayed and start at the nearest quarter second, either .0, .250, .5, or .750 past the second.

You only need to use the sync_ms setting for the specific shows you want to keep in sync. Typically this would be used for light or LED shows, as new shows starting should align nicely to existing shows that are already running.

The value of sync_ms you should use should be one complete "cycle" of the show. For example, if you flash your lights or LEDs at a rate of 250ms on / 250ms off, then you should use `sync_ms: 500` to ensure every show starts at the nearest 500ms point, thus ensuring that all lights will be "on" or "off" at the same time. (If you set `sync_ms: 250` in this case, then your shows will be in sync but they might be offset from each other.)

If your show is 200ms on / 200ms off, set sync_ms to 400. If your show is 400ms on / 250ms off, set sync_ms to 650. Etc.

If you're wondering whether sync_ms is bad because it delays a show start, and you don't want a show to be delayed, don't worry about it. The main use for sync_ms is when you have lights or LEDs that are flashing repeatedly, and in those cases, there's so much other stuff happening when they start flashing that no one is going to notice a delay of a fraction of a second when the show starts. (This is how is has to work anyway since you want the lights to be in sync.)

Playing Shows in a Show

Sometimes it can be useful to play other shows inside your show. Luckily, a show can use any *Config Players* and there is a *Show player*.

This is an example of an attract mode:

```
##! show: my_show
- duration: 3s
  shows:
    attract_show_collectlights:
      loops: 1
      speed: 10
      show_tokens:
        color: blue
- duration: 3s
  shows:
    attract_show_collectlights:
      loops: 1
      speed: 10
      show_tokens:
        color: red
```

It will first run a show in blue and then the same show in red. You would usually also add some sounds and slides which can be also in other shows. The organisation of your shows is up to you. This allows you to reuse shows with different parameters.

Video about shows:

https://youtu.be/Ou5xqCAthZY

You should have a look at *Config Players* to find more information about all the elements which are possible in shows (i.e. lights, slides, widgets or sounds).

Videos about shows:

https://youtu.be/Ou5xqCAthZY

https://youtu.be/bjDWm_pO9_I

https://youtu.be/9hMsnGfUliM

CHAPTER 13

Assets

Assets are files that your machine uses that are loaded from disks, such as show YAML files, images, and sound files. MPF has lots of flexibility for how assets are loaded and unloaded. (For example, if you're running MPF on a machine that doesn't have a lot of memory, you may not be able to load all the assets at startup and may instead have to dynamically load and unload assets throughout the game.)

MPF also has the ability to automatically "discover" various types of assets in your machine folder, meaning you don't have to manually type every single asset file name into your config files. You can even set asset properties based on what folder and/or subfolder they're in. (For example, audio files in `/sounds/fx` are automatically played on the sound effects track, while sound files in `/sounds/voice` are played on the voice track.

As of MPF 0.33, assets can be in nested subfolders too. For example:

```
\sounds
\sounds\fx
\sounds\fx\pops
\sounds\fx\slings
\sounds\voice\red
\sounds\voice\ted
\sounds\voice\bob
```

MPF also supports "asset pools" for sound and image assets which allow you to group multiple asset files into a single asset name that you use in MPF. This lets you add "variation" to assets during game play. For example, if you have a laser sound when a pop bumper is hit, you could actually have four different laser sound files that are each slightly different which you pool into the "laser" asset which is associated with the pop bumper, and then each time the pop bumper is hit you get one of the four sounds played at random instead of the same sound over and over.

Creating "pools" of assets

Help us to write it

Bitmap Fonts (asset type)

See *Bitmap Fonts* and *bitmap_fonts:*.

Images (asset type)

See *Image Widget*.

Shows (asset type)

See *Shows* and *shows:*.

Sounds (asset type)

See *Sounds, Music & Audio*.

Videos (asset type)

See *Video widget*.

CHAPTER 14

Config Players

An important concept to using the YAML-based configuration files is something we call *config players*.

Config players in MPF have nothing to with the actual human players of your machine, rather, they are things that "play" based on configurations.

Config players are used in both the machine-wide and mode-specific config files, and also in show steps.

- In a config file, the config players are setup via the `<config_player_name>_player:` section of the file.
- In show steps, config players are accessed via the `<config_player_name>s:` setting.

Some examples:

- You play sounds via a config file in the `sound_player:` section, and you play sounds from a show step via the `sounds:` setting for that step.
- You show slides on a display via a config file in the `slide_player:` section, and you show slides from a show step via the `slides:` setting for that step.
- You set the color of LEDs via a config file in the `light_player:` section, and you set colors from a show step via the `lights:` setting for that step.
- You set player and machine variables based on events in the `variable_player:` section (this is commonly used for scoring in your machine), and you set variables from a show step via the `variables:` setting of that step.
- etc.

Video about events in MPF:

https://youtu.be/G3UbVP8gFU0

Video about shows:

https://youtu.be/Ou5xqCAthZY

Standalone Config Player

General syntax looks like this in a standalone player:

Normal syntax

```
example_player:
  event_which_is_posted_elsewhere:
    <depends on the player>
```

For example (show_player; short syntax):

```
show_player:
  event_which_is_posted_elsewhere:
    your_show: play
```

Another example (show_player; long syntax):

```
show_player:
  event_which_is_posted_elsewhere:
    your_show:
      action: play
      sync_ms: 1000
```

One line syntax

This is not supported in all players. This usually performs the default action on the element:

```
example_player:
  event_which_is_posted_elsewhere: <depends on the player>
```

An example (show_player):

```
show_player:
  event_which_is_posted_elsewhere: your_show
```

Subscription syntax

This is not supported for all variables and all players. It will perform the action (i.e. play a show or enable a light) when the condition becomes true. Later it will remove/stop the action (i.e. stop the show or disable the light) when the condition becomes false.

```
example_player:
  "{machine.test_machine_var == 23}":
      <depends on the player>
```

An example (light_player):

```
light_player:
  "{current_player.score > 1000000}":
    score_1M: white
```

See *Conditional Events* for details about conditionals.

Config Player in a Show

All config players also work in shows. However, you need to skip the event which triggers the player since the action is triggered by the show.

Normal syntax

This supports the same syntax as above (just without the event). Also note that instead of `example_player:` it becomes `examples:`.

```
- duration: 2s
  examples:
    <depends on the player>
```

For example (show_player; short syntax):

```
##! show: test
- duration: 2s
  shows:
    your_show: play
```

Another example (show_player; long syntax):

```
##! show: test
- duration: 2s
  shows:
    your_show:
      action: play
      sync_ms: 1000
```

One line syntax

There is no one line syntax in shows.

There are several different config players in MPF and MPF-MC. Click on each below for specific details of how to use them, with explanations of how to use them in config files and in shows.

Blinkenlight player

The *blinkenlight player* is a *config player* that's used add or remove colors from a blinkenlight.

Note: What is a blinkenlight?

A blinkenlight is a flashing light on your pinball machine. But it's different than a normal flashing light that you might have in a show, because with a blinkenlight, multiple modes (or shows) can add colors to the light, and the blinkenlight will flash all the colors in a sequence.

Blinkenlights are useful if you have a game where multiple modes could be running and they all share the same shot(s). Each mode could have a different color, and the light for the shot could flash between all the colors, to show the player that shooting that shot will score all those modes at once.

For example, this is the kind of thing that happens in Stern's Game of Thrones. If you start multiple houses at the same time, and they share shot(s) the sigils in front of those shots will flash different colors to show you that the same shot will score different modes. The left ramp might alternate between white (Stark) and red (Lannister) to show you that both modes will score that shot.

This is an example of a blinkenlight player:

```
blinkenlight_player:
  some_event:
    my_blinkenlight: red
  some_other_event:
    my_blinkenlight:
      action: add
      key: blue_color
      color: blue
```

In the example above, when the event called `some_event` is posted, the color red will be added to the blinkenlight `my_blinkenlight`. When the event `some_other_event` is posted, the color blue will be added to the same blinkenlight. This color has the key `blue_color`, which is useful if we want to remove the color later.

Note that the `some_event` example has a shortened config than the `some_other_event` example. This shortened version is called an express config. The blinkenlight's express config is the same thing as doing an "add" action with the color you specify. It's just a shorter way to do the same thing. You can read more about express configs *here*. Also, see the section below called Express Config for more information on the express config.

To use a blinkenlight player, you first have to define a blinkenlight within your machine config (see the blinkenlight page *here*). The blinkenlight's config will set which actual light in your pinball machine is controlled by the blinkenlight. When you add a color to the blinkenlight from a blinkenlight player, the light will start to flash that color. As more colors are added, the light will cycle through all the colors.

Note: It makes the most sense to specify an RGB LED as the underlying light for a blinkenlight, since the whole idea is for the light to flash different colors. There's nothing that stops you from adding a regular light, though. Instead of colors, you could specify different brightness levels to flash. Either way, we refer to those as "colors" in this documentation.

A Blinkenlight's cycle

The blinkenlight will cycle through all the colors that have been added to it. You have control over how this cycle works. All of the options to control a blinkenlight's cycle are controlled from the `blinkenlights` section of your config file, and not the `blinkenlight_player` section.

For example, you can specify how long each color is displayed by setting the color_duration property of the blinkenlight. This will cause each color to be displayed for a specified length of time, and the more colors you add, the longer the blinkenlight's cycle will take. Alternatively, if you want the blinkenlight's cycle to last a certain amount of time regardless of how many colors are added to it, you can set the cycle_duration instead.

When the blinkenlight only has one color, it will flash that single color in a sequence like this: on off on off and so on. When more than one color is added, however, you might not want the light to turn off at the end of each cycle. For example, if your blinkenlight has two colors, red and green, you might want the cycle to be: red green off red green off, or you might simply want it to be red green red green. You can control this behavior with the off_when_multiple setting of your blinkenlight. Setting this value to True will turn the light off at the end of each cycle. The default value is False, which will not turn the light off when multiple colors are present.

Blinkenlights will stay in sync with each other automatically, if you have multiple blinkenlights with the same settings and the same number of colors. So, for example, if you have two blinkenlights with the same cycle_duration, and each one has 3 colors added to it, then they will start their cycles at the same time, and end their cycles at the same time. This way, the blinkenlights will stay in sync with each other. This is useful if you have a blinkenlight on your left and right ramps, for example, and you want them both to flash the same colors. You would want them to show the same colors at the same time.

A cycle example

Let's say you have a blinkenlight that is set up like this:

```
blinkenlights:
  blinkenlight_1:
    cycle_duration: 1s
    off_when_multiple: false
    light: l_left_ramp_arrow
```

In this case, blinkenlight_1 has a cycle_duration value of 1s. That is, each cycle lasts 1 second, regardless of how many colors the blinkenlight has. Now, let's say you use a blinkenlight_player to add the color red to the blinkenlight. Now the blinkenlight's cycle would look like this:

1 second		1 second		1 second		1 second	
red	off	red	off	red	off	red	off

If green color is added to the blinkenlight, the cycle would change to this:

1 second		1 second		1 second		1 second	
red	green	red	green	red	green	red	green

Now let's say a third color (blue) is added:

1 second			1 second			1 second			1 second		
r	g	b	r	g	b	r	g	b	r	g	b

Note that each color now is only 1/3 of a second long, since there are three of them per cycle now.

Now, blue is removed from the blinkenlight, while the blinkenlight is currently showing a blue color during the second cycle:

1 second			1 second				1 second		1 second	
r	g	b	r	g	b	g	red	green	red	green

Notice how blue is displayed when the color is removed, and the light immediately switches to green, since green should be displayed at that point in time now that the blinkenlight only has 2 colors. So the end result is green "flashes" very briefly before red is displayed again and the red/green cycle starts.

Using Blinkenlights in shows

You can also use blinkenlight_player from within a show. This lets you add colors to a blinkenlight during a show. It probably doesn't make sense most of the time to do this, because colors you add to a blinkenlight will only stick around while the show is active. Once the show ends, the colors you added during that show will automatically be removed from the blinkenlight.

Note: This is true of colors added during modes as well. If a mode in your game adds colors to a blinkenlight, those colors will be automatically removed from the blinkenlight when the mode ends. If you restart the mode, those colors won't come back automatically, however, so keep that in mind. You might need to add the colors again when the mode restarts, depending on how your game works.

Example blinkenlight player from a show:

```
##! show: test
- time: 0
  blinkenlights:
    my_blinkenlight: red
```

Usage in config files

In config files, the blinkenlight player is used via the `blinkenlight_player:` section.

Usage in shows

In shows, the blinkenlight player is used via the `blinkenlights:` section of a step.

Express Config and Keyless Colors

As mentioned above, the express config for `blinkenlight_player` performs the `add` action. So, the color you specify as the express config value will be the color to add to the blinkenlight. However, if you add a color this way, there is no `key` value for the color. Or, more specifically, the `key` value will be empty. We could refer to colors without a `key` value as a keyless color. If you later use the `remove` action and don't specify a `key` to remove, then the keyless color will be removed.

This is better explained with an example. Consider this `blinkenlight_player`:

```
blinkenlight_player:
  some_event:
    my_blinkenlight: red
  some_other_event:
    my_blinkenlight:
      action: remove
```

In this case, the color red will be added to my_blinkenlight when the some_event event is posted. This color doesn't have a key value, so this color is keyless. When the some_other_event event is posted, the remove action is performed. Since this remove action also didn't specify a key value, then MPF will look for a keyless color and remove that color from the blinkenlight. In this case, the color red will be removed.

Note that keyless colors are only valid within the context of the mode or show that is performing the keyless action. So, a remove action from mode1 will not remove the keyless color that was added by mode2. It will only remove the keyless color added by mode1.

There's a special value you can use in the express config to remove a keyless color. Instead of using the full config and specifying an action: remove as we did above, you can use the special color stop or remove in the express config to do the same thing. The following is equivalent to the example above:

```
blinkenlight_player:
  some_event:
    my_blinkenlight: red
  some_other_event:
    my_blinkenlight: remove
```

In this case, the red color is added to the blinkenlight when some_event is posted, and then removed when some_other_event is posted.

Config Options

See *blinkenlight_player:* for config details.

Coil player

The *coil player* is a *config player* that's used pulse, enable, or disable coils and drivers.

This is an example:

```
coil_player:
  some_event: coil_1
  some_other_event:
    coil_2:
      action: enable
      hold_power: .5
```

In the example above, when the event called some_event is posted, coil_1 will pulse. When the event some_other_event is posted, coil_2 will enable (be held on) at power level 0.5 (means 50% of maximum power).

Note that the some_event: coil_1 is entered in a different way than the some_other_event:. The first one has a simple key/value pair, whereas the second has a complete nested sub-configuration.

The first example shows the "express" config, while the second shows the full config. (What's an "express config?" Details *here*.

The coil player's express config is the "pulse" action.

Example coil player from a show:

```
##! show: test
- time: 0
  coils:
    coil1: pulse
```

Usage in config files

In config files, the coil player is used via the `coil_player:` section.

Usage in shows

In shows, the coil player is used via the `coils:` section of a step.

Config Options

See *coil_player:* for config details.

Using LEDs as display (display_light_player)

You can map any display to your playfield LEDs or any LEDs (e.g. a LED matrix) in your machine. This enables you to leverage any MC features and display them on any LEDs (or more specifically any lights) in your machine.

Video about display_light_player:

https://youtu.be/38hc7IIfVJI

To use this in a show you can use this:

```
##! show: test_show
- display_lights:
    your_source_display:     # use any display defined in your machine
      lights: "*"            # map all lights. you can also use a tag
```

Or standalone:

```
display_light_player:
  your_event:
    your_source_display:
      lights: "*"
```

Then map your lights to a position on the display:

```
lights:
  l_light1:
    number: 1
    x: 0.3595817467355206
    y: 0.026751757949132805
  l_light2:
    number: 2
    x: 0.34303657433971446
    y: 0.02873336964906857
```

You can map those in the MPF monitor and then copy the locations using the script in `tools/monitor_to_config.py` or manually. You may need to adjust config names in the script (improvements welcome).

Usage in config files

In config files, the display light player is used via the `display_light_player:` section.

Usage in shows

In shows, the display light player is used via the `display_lights:` section of a step.

Config Options

See *display_light_player:* for config details.

Event player

The *event player* is a *config player* that's used to post events.

Video about events in MPF:

https://youtu.be/G3UbVP8gFU0

Basic Event Playing

```
event_player:
  ball_starting:
    - cmd_flippers_enable
    - cmd_autofire_coils_enable
    - cmd_drop_targets_reset
  ball_ending:
    - cmd_flippers_disable
    - cmd_autofire_coils_disable
  tilt:
    - cmd_flippers_disable
    - cmd_autofire_coils_disable
  slam_tilt:
```

(continues on next page)

(continued from previous page)

```
    - cmd_flippers_disable
    - cmd_autofire_coils_disable
```

The event player settings above will post the events *cmd_flippers_enable*, *cmd_autofire_coils_enable*, and *cmd_drop_targets_reset* when the *ball_starting* event is posted. Similarly they will post events to disable the flippers and autofire coils when ball end and tilt events are posted.

To use this, simply create an `event_player:` entry in your config file. Then create sub- entries for each event you want to trigger other events, and add a list of one or more events that should be posted automatically under each trigger event.

Remember that you can create this event_player: section in either your machine-wide or in mode-specific config files. For example, if you want a target called "upper" to reset when a mode called "shoot_here" starts, you could create an entry like this in the shoot here mode's shoot_here.yaml mode configuration file:

```
##! mode: shoot_here
event_player:
  mode_shoot_here_started: cmd_upper_target_reset
```

Conditional Event Playing

Events in the event player can be conditional, to allow precise control over when an event is played:

```
##! mode: base
event_player:
  mode_base_started{current_player.score>10000}:
    - start_mode_superbonusround
    - play_show_richy_rich
  start_mode_battle{device.achievements.ironthrone.state!="completed"}: start_mode_choose_battle
  start_mode_battle{device.achievements.ironthrone.state=="completed"}: start_mode_victory_lap
```

In the above example, both "start_mode_superbonusround" and "play_show_richy_rich" will only be posted if the player's score is over 10,000 when base mode starts. And if the battle mode is started, either "start_mode_choose_battle" or "start_mode_victory_lap" will be posted depending on whether the *ironthrone* achievement has been completed.

Conditions can also be applied to events within a list, to allow one event to trigger a variable number of handlers:

```
##! mode: base
event_player:
  reenable_nonrecruit_modes:
    - start_mode_shadowbroker_base
    - start_mode_n7_assignments
    - start_mode_overlordlight{device.achievements.collectorship.state!="complete"}
    - start_mode_arrival{device.achievements.collectorship.state=="complete"}
    - start_mode_shopping{current_player.cash>=1000}
```

In the above example, both "start_mode_shadowbroker_base" and "start_mode_n7_assignments" will be posted every time. One of either "start_mode_overlord" or "start_mode_arrival" will be posted, depending on whether the player has completed the collectorship achievement. And if the player_var "cash" is high enough, "start_mode_shopping" will also be posted.

In many cases, conditions can be applied to either the triggering event or the handling event. For more information and examples of conditions, see *conditional events*.

Dynamic Values in Events

There are numerous ways to include dynamic values (player variables, device states, mathematical calculations) in events.

Dynamic Event Names

An event name can use parenthetical values to dynamically determine the event.

```
event_player:
  mode_dynamo_started:
    # Player variables can be dropped into event names
    - play_dynamo_show_phase_(current_player.phase_name)
    # Machine and device states can be used
    - dynamo_started_with_state_(device.achievements.dynamo.state)
    # Dynamic evaluations can be done to calculate values
    - player_score_is_("high" if current_player.score > 10000 else "low")
```

In the above example:

- With the player variable phase_name having a value of “attackwave”, starting the mode would post the event *play_dynamo_show_phase_attackwave*
- If the “dynamo” achievement was completed, starting the mode would post *dynamo_started_with_state_completed*. If the achievement was instead disabled, the event would be *dynamo_started_with_state_disabled*
- If the player’s score is over 10,000 the event *player_score_is_high* will be posted, otherwise the event *player_score_is_low* will be posted.

Any *dynamic values* can be used. Because event names are always strings, all dynamic values will be converted to their string equivalent.

Dynamic Event Arguments

An event post can include arguments to provide event handlers with additional information about the event. An event configured as an object will post the object properties as its arguments:

```
event_player:
  mode_carchase_started:
    # Objects can be expanded for a key/value pair per line
    set_environment_sounds:
      env_name: driving
    # Objects can be inline for brevity
    set_initial_laps_count: {count: 10}
```

You can go a step further and include dynamic values as the values for event arguments. To indicate that an argument’s value is dynamic, use the `value:` property.

```
event_player:
  mode_dynamo_started:
    set_dynamo_phase:
      phase_name: {value: current_player.dynamo_phase}
```

In the above example, if the player variable dynamo_phase had the value "attackwave", the event would be posted as such:

```
Event: ======'set_dynamo_phase'====== Args={'phase_name': 'attackwave', priority': 0}
```

Because dynamic values can come from a variety of sources, you will need to explicitly define types for the value's format. Acceptable types are **int**, **float**, **bool**, and **string**. If no type is configured, the value will be posted as a string.

```
event_player:
  mode_dynamo_started:
    # This event arg will be correctly typed
    set_dynamo_round_with_type:
      round_number:
        value: device.counters.dynamo_rounds.value
        type: int
    # This event arg will be converted to a string
    set_dynamo_round_without_type:
      round_number:
        value: device.counters.dynamo_rounds.value
```

Priority:

Note that as with other config players, event player can accept priorties for events to be posted. This can be useful in scenarios such as where a player variable must be updated prior to a condtional check, so that they happen in the desired order.

```
event_player:
  mode_dynamo_started:
    reset_pv_tokens_collected_to_0
      priority: 50
    play_slide{current_player.pv_tokens_collected <= 5}:
      priority: 5
      slide: dynamo_collect_more_tokens_slide
```

Usage in config files

In config files, the event player is used via the `event_player:` section.

Usage in shows

In shows, the event player is used via the `events:` section of a step.

Config Options

See *event_player:* for config details.

Flasher player

The *flasher player* is a *config player* that's used to flash lights.

Usage in config files

In config files, the flasher player is used via the `flasher_player:` section.

Usage in shows

In shows, the slide flasher is used via the `flashers:` section of a step.

Config Options

See *flasher_player:* for config details.

GI (general illumination) player

`gi_player` has been removed in 0.50. Use *light_player* instead.

Hardware Sound player

The *hardware sound player* is a *config player* that's used to control sounds. (This player is part of the MPF media controller and only available if you're using MPF-MC for your media controller.)

Usage in config files

In config files, the sound player is used via the `hardware_sound_player:` section. Event names that will trigger sound actions are nested sub-headings and sound names are either listed as nested sub-headings below that.

Usage in shows

In shows, the sound player is used via the `hardware_sounds:` section of a step.

Optional settings

Additional information may be found in the *hardware_sound_player* configuration reference documentation.

LED player

led_player and *matrix_light_player* were replaced with *light_player* in MPF 0.50. See *lights:* for details.

Light player

The *light player* is a *config player* that's used to set the brightness and color of lights (including turning them on and off).

Usage in config files

In config files, the light player is used via the `light_player:` section.

The `light_player:` section of your config is where you can control lights in config or shows. Example in config:

```
light_player:
  some_event:
    led1:
      color: red
      fade: 200ms
    led2:
      color: ff0000
      fade: 2000ms
```

Usage in shows

In shows, the light player is used via the `lights:` section of a step.

```
shows:
  red_color:
    - lights:
        l_light: red
show_player:
  turn_light_red_event: red_color
```

Setting multiple lights

```
lights:
  l_target1:
    number:
  l_target2:
    number:
shows:
  rainbow:
    - lights:
        (leds): red
    - lights:
```

(continues on next page)

(continued from previous page)

```
        (leds): orange
    - lights:
        (leds): yellow
    - lights:
        (leds): green
    - lights:
        (leds): blue
    - lights:
        (leds): purple
      duration: 3s
show_player:
  play_rainbow_show_on_targets:
    rainbow:
      show_tokens:
        leds: l_target1, l_target2
```

The show `rainbow` will turn your LED(s) in the placeholder `(leds)` to a different color every second (because 1s is the default duration of a step). The last step (purple) will stay for 3s. When you post `play_rainbow_show_on_targets` the show is played on two lights which are referenced directly.

Setting lights via tags

```
shows:
  rainbow:
    - lights:
        (tag): red
      duration: 1s
    - lights:
        (tag): orange
      duration: 1s
    - lights:
        (tag): yellow
      duration: 1s
    - lights:
        (tag): green
      duration: 1s
    - lights:
        (tag): blue
      duration: 1s
    - lights:
        (tag): purple
      duration: 1s
show_player:
  play_rainbow_show_via_tag:
    rainbow:
      show_tokens:
        tag: drops
```

In `play_rainbow_show_via_tag` we reference (two) lights via the tag `drops`.

Fade lights between steps

There are two syntax to express fades. Short syntax which is `(color)-f(time)(unit)` (i.e. `red-f200ms`) or extended syntax which is a dict with two entries for `color` and `fade`. Here is an example for the short syntax:

```
shows:
  rainbow_with_fade_f_syntax:
    - lights:
        l_rgb: red-f1s
      duration: 1s
    - lights:
        l_rgb: orange-f1s
      duration: 1s
    - lights:
        l_rgb: yellow-f1s
      duration: 1s
    - lights:
        l_rgb: green-f1s
      duration: 1s
    - lights:
        l_rgb: blue-f1s
      duration: 1s
    - lights:
        l_rgb: purple-f1s
      duration: 1s

show_player:
  play_rainbow_show: rainbow_with_fade_f_syntax
```

And an example with extended syntax:

```
shows:
  rainbow_with_fade_extended_syntax:
    - lights:
        l_rgb:
          color: red
          fade: 1s
      duration: 1s
    - lights:
        l_rgb:
          color: orange
          fade: 1s
      duration: 1s
    - lights:
        l_rgb:
          color: yellow
          fade: 1s
      duration: 1s
    - lights:
        l_rgb:
          color: green
          fade: 1s
      duration: 1s
```

(continues on next page)

(continued from previous page)

```
    - lights:
        l_rgb:
          color: blue
          fade: 1s
      duration: 1s
    - lights:
        l_rgb:
          color: purple
          fade: 1s
      duration: 1s

show_player:
  play_rainbow_show: rainbow_with_fade_extended_syntax
```

In most cases simple syntax is sufficient. Extended syntax is easier to use with placeholders.

Config Options

See *light_player:* for config details.

Playlist player

The *playlist player* is a *config player* that's used to control playlists. (This player is part of the MPF media controller and only available if you're using MPF-MC for your media controller.)

Usage in config files

In config files, the playlist player is used via the `playlist_player:` section. Event names that will trigger playlist actions are nested sub-headings and playlist names are either listed as nested sub-headings below that.

Usage in shows

In shows, the sound player is used via the `playlists:` section of a step.

Optional settings

Additional information may be found in the *sound_player* configuration reference documentation.

Queue Event player

The *queue event player* is a *config player* that's used to play queue events.

Video about events in MPF:

https://youtu.be/G3UbVP8gFU0

Usage in config files

In config files, the event player is used via the `queue_event_player:` section.

Usage in shows

None. (It's not valid in shows since it doesn't make sense in shows.)

Config Options

See *queue_event_player:* for config details.

Queue Relay player

The *queue relay player* is a *config player* that's used to block queue events.

Video about events in MPF:

https://youtu.be/G3UbVP8gFU0

Usage in config files

In config files, the event player is used via the `queue_relay_player:` section.

Usage in shows

None. (It's not valid in shows since it doesn't make sense in shows.)

Config Options

See *queue_relay_player:* for config details.

Random event player

The *random event player* is a *config player* that's used to post random events from a list of events.

This is an example:

```
# in your global config:
random_event_player:
  play_random_event_global:
    scope: machine
    events:
      - event1
      - event2
      - event3
##! mode: base
```

(continues on next page)

(continued from previous page)

```
# in your mode:
random_event_player:
  play_random_event:
    events:
      - event1
      - event2
      - event3
  play_random_event_with_weight:
    events:
      unlikely_event1: 2
      unlikely_event2: 3
      likely_event1: 45
      likely_event2: 50
```

When *play_random_event* is posted a random event is posted out of the list *event1*, *event2* or *event3*.

Usage in config files

In config files, the random event player is used via the `random_event_player:` section.

Usage in shows

In shows, the random event player is used via the `random_events:` section of a step.

Config Options

See *random_event_player:* for config details.

Segment Display player

The *segment display player* is a *config player* that's used to show text or numbers on segment displays.

Usage in config files

In config files, the segment display player is used via the `segment_display_player:` section.

Usage in shows

In shows, the segment display player is used via the `segment_displays:` section of a step.

Config Options

See *segment_display_player:* for config details.

Show player

The *show player* is a *config player* that's used to start, stop, pause, resume, advance, and/or update shows.

Video about shows:

https://youtu.be/Ou5xqCAthZY

This is an example:

```
show_player:
  some_event: your_show_name
  some_other_event: another_show
```

In the example above, when the event *some_event* is posted, the show called `your_show_name` will be played (started). When the event *some_other_event* is posted, the show called `another_show` will be played.

Notice that the config above has simple key/value pairs in the form of *event: show*. You can list as many of those as you want in the show player, and when each event is posted, it will start the show with the same name.

However there are times when you might want to specify additional options for a show. Perhaps you want to change the playback speed, or configure how it repeats. In that case, instead of putting the show name on the same line as the event, you can put the show name on a new line under the event, and then add additional settings under it, like this:

```
show_player:
  some_event:
    your_show_name:
      loops: 0
  some_other_event:
    another_show:
      speed: 2
      sync_ms: 500
```

In the example above, the show `your_show_name` will play when the event *some_event* is posted, but instead of playing with the default settings only, it will also play with the setting `loops: 0` (meaning it will not loop and just play once). Same for the other show above, which will play with a `speed: 2` and `sync_ms: 500`.

You can also mix-and-match formats, like this:

```
show_player:
  some_event: your_show_name
  some_other_event:
    another_show:
      speed: 2
      sync_ms: 500
```

Show keys

Each show played by a show player will be referenced internally using an unique `key`. The show_player will use the show name as key for the show by default if you do not specify a `key` (fine in most cases). This way it refences the show when starting or stopping it:

```
show_player:
  start_my_show:
    your_show_name: play
  stop_my_show:
    your_show_name: stop
```

In this example the event `start_my_show` will start `your_show_name` with key `your_show_name`. The event `stop_my_show` will then stop the same show using the key `your_show_name`. This simple mechanism will work fine for most cases.

However, in some cases you want to play multiple instances of one show in a single show. You can manually assign keys to run distinct shows. That way you can also specifically stop them later:

```
show_player:
  start_my_show1:
    your_show_name:
      action: play
      key: show1
      show_tokens:
        leds: my_led1
  start_my_show2:
    your_show_name:
      action: play
      key: show2
      show_tokens:
        leds: my_led2
  stop_my_show1:
    show1: stop
  stop_my_show2:
    show2: stop
```

In this example `start_my_show1` and `start_my_show2` will start separate instances of `your_show_name` which can indendently be stopped using `stop_my_show1` and `stop_my_show2`. If you omit `key` in this example `start_my_show1` and `start_my_show2` would stop the other and you would either see `your_show_name` with `my_led1` or `my_led2` but not both at the same time.

A key is only unique to one show_player so different modes will not interfere.

Usage in config files

In config files, the show player is used via the `show_player:` section.

Usage in shows

In shows, the show player is used via the `shows:` section of a step. (Yes, you can include shows in shows, meaning you can essentially use a parent show like a playlist, or as a controller that starts and stops other shows.)

Config Options

See *show_player:* for config details.

Slide player

The *slide player* is a *config player* in the MPF media controller that is used to play slide content, including showing slides, hiding slides, and removing slides. (This player is part of the MPF media controller and only available if you're using MPF-MC for your media controller.)

Note that the slide player is a *config_player*, so everything mentioned below is valid in the `slide_player:` section of a config file *and* in the `slides:` section of a show step. You can test slides and widgets interactively using *Interactive MC (iMC)*.

Full instructions on how to use the slide_player are included in the *How to Show a Slide on a Display* guide. The documentation here is for reference later.

Generically-speaking, there are two formats you can use for slide_player entries: "express" and "full" configs. Express configs will look like this:

```
slide_player:
  event1: slide1
  event2: slide2
  event3: slide3
```

Full configs will look like this:

```
slide_player:
   event1:
      slide1:
         <settings>
   event2:
      slide2:
         <settings>
   event3:
      slide3:
         <settings>
```

In both cases, these configurations are saying, "When *event1* is posted, show *slide1*. When *event2* is posted, show *slide2*. Etc."

This "express" config is down-and-dirty, with no options, to just show slides. The full config lets you specify additional options (based on the settings detailed below).

For example, the following config will show *slide_1* when *some_event* is posted, but it will also override the default settings and show the slide on the display target called *display1* and at a priority that's 200 higher than the base priority.

```
slide_player:
  some_event:
    slide_1:
      target: display1
      priority: 200
```

Showing dynamically-created slides

Both of the examples so far assumed that you were using the slide player to show a slide that had already been defined in the *slides:* section if your config. However you can also define slides right in-line in your slide player.

The following config will show a slide called *slide_1* when the *some_event* is posted, but it assumes that *slide_1* does not yet exist, and it contains a list of widgets (one text widget and one rectangle widget) which will be added to that slide.

Note that slide names are global in MPF, so if you already had a slide defined called *slide_1* and you redefine it in your slide player like the example below, this new slide will become *slide_1* and the old one will be gone.

```
slide_player:
  some_event:
    slide_1:
      widgets:
        - type: text
          text: I AM A TEXT WIDGET
        - type: rectangle
          width: 200
          height: 100
          color: red
```

You can also mix-and-match defining a slide in the slide player as well as adjusting properties of how the slide is shown. Just add multiple settings, like this:

```
slide_player:
  some_event:
    slide_1:
      widgets:
        - type: text
          text: I AM A TEXT WIDGET
        - type: rectangle
          width: 200
          height: 100
          color: red
      transition: wipe
```

Remember that these slide player settings can also be used in show steps (in a `slides:` section). Any of the examples above apply, you just don't include the event name, like this:

```
##! show: show1
#show_version=5
- time: 0
  slides: slide1
- time: +3
  slides: slide2
- time: +3
  slides:
    slide3:          # newly-defined slide here
      widgets:
      - type: text
        text: I AM SLIDE 3 IN THIS SHOW
```

(continues on next page)

(continued from previous page)

```
        color: lime
- time: +3
  slides:
    slide4:
      transition:
        type: move_out
        duration: 1s
        direction: up
```

Here's a list of all the valid settings for individual slides in the `slide_player:` section of your config file or the `slides:` section of a show. Note that all of these are optional. Any that you do not include will be automatically added with the default values applied.

Usage in config files

In config files, the slide player is used via the `slide_player:` section.

Usage in shows

In shows, the slide player is used via the `slides:` section of a step.

List of settings and options

Refer to the *slide_player* section of the config file reference for a full explanation of how to use the slide player in both config and show files.

Config Options

See *slide_player:* for config details.

Sound Loop player

The *sound loop player* is a *config player* that's used to control sound loop sets (used by sound loop audio tracks). (This player is part of the MPF media controller and only available if you're using MPF-MC for your media controller.)

Examples:

```
sound_loop_player:
  play_basic_beat:
    loops:
      action: play
      sound_loop_set: basic_beat
      timing: loop_end
  add_hi_hats:
    loops:
      action: play_layer
      layer: 1
```

(continues on next page)

(continued from previous page)

```
    timing: loop_end
  stop_hi_hats:
    loops:
      action: stop_looping_layer
      layer: 1
  add_snare:
    loops:
      action: play_layer
      fade_in: 2s
      layer: 2
      timing: now
  add_claps:
    loops:
      action: play_layer
      layer: 3
      timing: loop_end
```

Basic usage:

```
sound_loop_player:
  <triggering_event_name>:
    <sound_loop track name>:
      action: <action name>
      <optional settings>
  <triggering_event_name>:
    <sound_loop track name>:
      action: <action name>
      <optional settings>
```

Usage in config files

In config files, the sound player is used via the `sound_loop_player:` section. Event names that will trigger sound actions are nested sub-headings and sound_loop_set names are either listed as nested sub-headings below that.

Usage in shows

In shows, the sound player is used via the `sounds_loop_sets:` section of a step.

Optional settings

Additional information may be found in the *sound_loop_player* configuration reference documentation.

Sound player

The *sound player* is a *config player* that's used to control sounds. (This player is part of the MPF media controller and only available if you're using MPF-MC for your media controller.)

See *Sounds, Music & Audio* and *How to setup sound for your machine* for details.

Usage in config files

In config files, the sound player is used via the `sound_player:` section. Event names that will trigger sound actions are nested sub-headings and sound names are either listed as nested sub-headings below that.

Usage in shows

In shows, the sound player is used via the `sounds:` section of a step.

Optional settings

Additional information may be found in the *sound_player* configuration reference documentation.

Track player

The *track player* is a *config player* that's used to control audio tracks when MPF events are received. Tracks can be stopped, paused, or played with an optional fade time. The volume of a track can also be changed with an optional fade time. Finally, all sounds currently playing on a track can be stopped (again with an optional fade out time). (This player is part of the MPF media controller and only available if you're using MPF-MC for your media controller.)

Usage in config files

In config files, the track player is used via the `track_player:` section. Event names that will trigger track actions are nested sub-headings and track names are listed as nested sub-headings below that. `__all__` can be used in place of a track name to apply the action to all audio tracks in the sound system.

Example:

```
track_player:
  pause_music_track:
    music:
      action: pause
      fade: 1 sec
  resume_music_track:
    music:
      action: play
  stop_sounds_on_all_tracks:
    __all__:
      action: stop_all_sounds
      fade: 0.5 sec
```

Usage in shows

In shows, the track player is used via the `tracks:` section of a step.

Example:

```
shows:
  my_show_with_sound:
    - time: 0
      tracks:
        music:
          action: set_volume
          volume: 0.3
          fade: 0.25 sec
    - time: 3.5
      tracks:
        music:
          action: set_volume
          volume: 0.5
          fade: 0.25 sec
```

Config Options

Additional information may be found in the *track_player* configuration reference documentation.

Variable player

The *variable player* is a *config player* that's used to set the value of player and machine variables. This is commonly used for scoring in your machine. See *variable_player* for more detailed information.

At the most basic level, you can use this to add to a player's score (which is technically adding value to the player variable called *score*), but in reality you can affect any player or machine variable.

Here's an example:

```
##! mode: mode1
variable_player:
  target_1_hit:
    score: 1000     # adds 1000 to the player's "score" variable
  ramp_1_hit:
    score: 10000    # adds 10,000 to the player's "score" variable
    ramps: 1        # adds 1 to the player's "ramps" variable
  ramp_1_timeout:
    ramps:
      int: 0          # sets the player's "ramps" variable to 0.
      action: set     # means that this event will "set" (or reset) the variable to the value, rather␣
↪than add to it
  ramp_2_hit:
    score:
      int: 25000 * current_player.ramps     # multiplies the value of the current player's "ramps"␣
↪variable by 25,000 and adds the result to the player's "score" variable
      block: true      # "blocks" this event from being passed to variable player sections from lower-
↪priority modes
  counter_treasure_value_complete:
    treasure_name:
      string: RUBY     # Sets the player's "treasure_name" variable to a string called "RUBY"
```

See our *player variables reference* and *machine variables reference* to learn about existing variables. You can also create player variables on the fly if they did not exist. If you want to define defaults for variables you may define them in the *player_vars:* or *machine_vars:* sections.

Usage in config files

In config files, the variable player is used via the `variable_player:` section.

Usage in shows

In shows, the variable player is used via the `variables:` section of a step.

Config Options

See *variable_player:* for config details.

Widget player

The *widget player* is a *config player* that's used to add or remove widgets to existing slides on a display. (This player is part of the MPF media controller and only available if you're using MPF-MC for your media controller.)

Note that the widget player is a *config_player*, so everything mentioned below is valid in the `widget_player:` section of a config file *and* in the `widgets:` section of a *show step*.

Full instructions on how to use the slide_player are included in the *Widgets* section of the documentation. The stuff here in the config reference is for reference later. You can test slides and widgets interactively using *Interactive MC (iMC)*.

Generically-speaking, there are two formats you can use for widget_player entries: "express" and "full" configs. Express configs will look like this:

```
widget_player:
  event1: widget1
  event2: widget2
  event3: widget3
```

Full configs will look like this:

```
widget_player:
   event1:
      widget1:
         <settings>
   event2:
      widget2:
         <settings>
   event3:
      widget3:
         <settings>
```

In both cases, these configurations are saying, "When *event1* is posted, add widget *widget1*. When *event2* is posted, add *widget2*. Etc."

This "express" config is down-and-dirty, with no options, to just add widgets to the current slide on the default display. The full config lets you specify additional options (based on the settings detailed below).

For example, the following config will add *widget_1* when *some_event* is posted, but it will also override the default settings and add widget to the slide called *slide_2*, even if that's not the current slide that's showing.

```
widget_player:
  some_event:
    widget_1:
      slide: slide_2
```

Usage in config files

In config files, the widget player is used via the `widget_player:` section.

Usage in shows

In shows, the widget player is used via the `widgets:` section of a step.

Config Options

See *widget_player:* for config details.

CHAPTER 15

Machine Management

MPF includes many features to help you manage your pinball machine.

(There's a lot to add here. This will include things like the service mode, auditor, remote monitoring and trouble reporting, etc. See: *Help us to write it*)

> **Warning:** If the `service` mode is added to `modes`, the message "coil power off" will appear when the coin door is open. This is only a message: MPF cannot actually turn the coil power off. You must ensure that your power system is wired appropriately to turn HV off when the coin door is open.

Auditor

Todo: *Help us to write it*

The Mission Pinball Framework contains an auditor that can be used to create audit logs of switch events, game events, shots made, and player variables. The exact behavior of what is (and isn't) included in the audit log is controlled in the Auditor section of your machine configuration files. Here's a sample audit file:

```
Events:
 ball_search_begin: 0
 ball_started: 1
 game_ended: 31
 game_started: 41
 machine_init_phase_1: 0
 machine_reset: 29
Player:
```

(continues on next page)

(continued from previous page)

```
 score:
 average: 15634
 top:
 - 71130
 - 59840
 - 50190
 - 47490
 - 39350
 - 33350
 - 25700
 - 24890
 - 21980
 - 21670
 total: 31
Shots:
 AirRaidRamp: 3
 DropTarget: 99
 FullRightOrbit: 5
 Inlane: 54
 LeftOrbit: 13
 LeftRamp: 4
 OrangeStandups: 11
 Outlane: 14
 RightRamp: 7
 Slingshot: 105
 WeakRightOrbit: 6
Switches:
 ShooterLaneL: 20
 alwaysClosed: 0
 buyIn: 0
 captiveBall1: 22
 captiveBall2: 10
 captiveBall3: 2
 centerRampExit: 16
 coin1: 0
 coin2: 0
 coin3: 0
 coin4: 0
 coinDoor: 0
 craneRelease: 0
 down: 0
 dropTargetD: 9
 dropTargetE: 51
 dropTargetG: 45
 dropTargetJ: 38
 dropTargetU: 47
 enter: 98
 esc: 80
 fireL: 0
 fireR: 122
 flipperLwL: 400
 flipperLwL_EOS: 388
 flipperLwR: 440
 flipperLwR_EOS: 434
```

(continues on next page)

(continued from previous page)

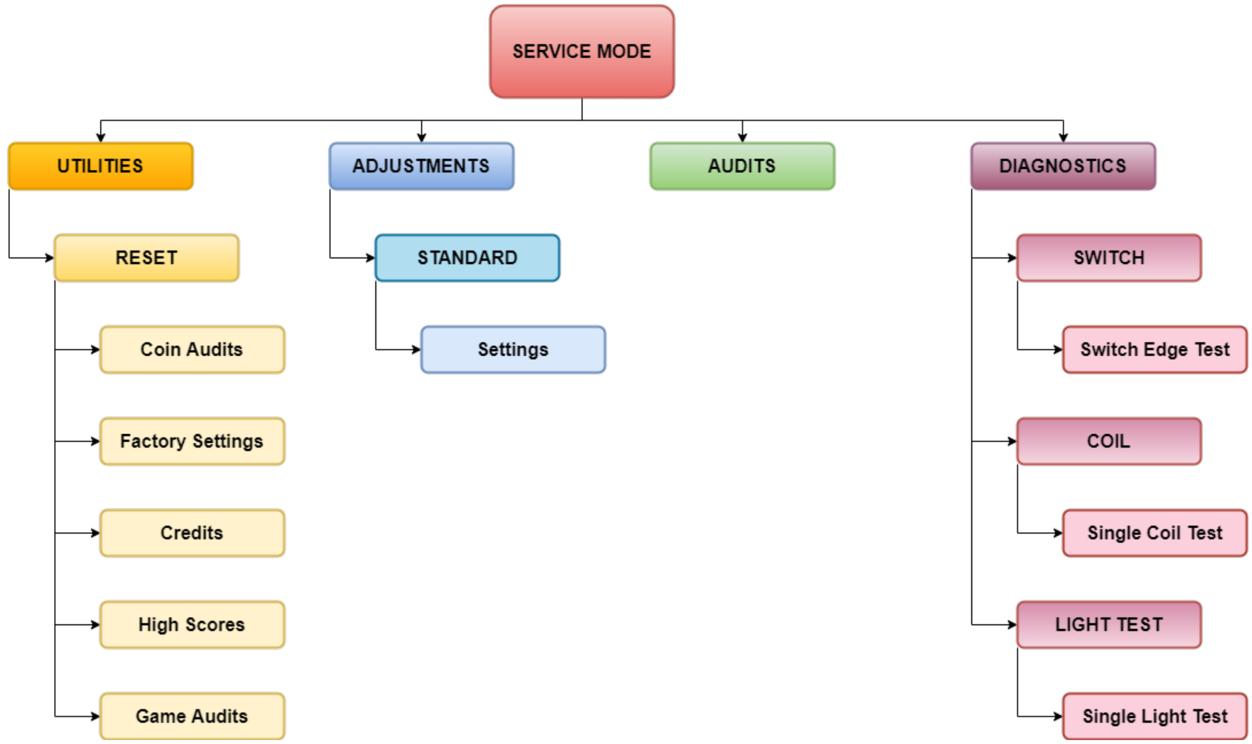
```
flipperUpL: 364
flipperUpL_EOS: 360
flipperUpR: 440
flipperUpR_EOS: 436
globePosition1: 108
globePosition2: 108
inlaneL: 40
inlaneR: 38
leftRampEnter: 24
leftRampExit: 8
leftRampToLock: 4
leftRollover: 136
leftScorePost: 42
magnetOverRing: 0
mystery: 8
outerInlaneR: 30
outlaneL: 22
outlaneR: 6
plumbBob: 0
popperL: 36
popperR: 20
rightRampExit: 14
rightTopPost: 28
shooterR: 106
slamTilt: 0
slingL: 134
slingR: 76
start: 47
subwayEnter1: 16
subwayEnter2: 16
superGame: 0
threeBankTargets: 22
ticketDispenser: 0
topCenterRollover: 24
topRampExit: 6
topRightOpto: 36
trough1: 120
trough2: 96
trough3: 96
trough4: 96
trough5: 96
trough6: 74
troughJam: 76
up: 0
```

Note that in the 'Player' section, the auditor will track the average, the Top 10, and the total numbers of each item. You can configure all this (including how many of each item it records) in the `auditor:` section of the configuration file'.

Service Mode

Service Mode is an important part of a pinball machine that provides an interface that allows the user to perform a number of important operations to their machine. MPF provides a comprehensive base set of service mode features, that can be extended if required.

The structure of the built-in Service Mode is as follows:



Utilities

MPF provides a **Reset** function that allows you to provide a set of standard functions to the user to reset certain elements of the game, such as High Scores, Audits and Earnings. This menu option is available from the `service_menu_selected` event with the label `Utilities Menu`.

Utilities has the following sub menus:

Coin Audits

Resets all counters for earnings data. All counters will be reset to zero in `earnings.yaml` in the `/data` subfolder of your game.

Factory Reset

Resets the value of all of your machine variables in your `machine_vars.yaml` file in the `/data` subfolder of your game to the `initial_value` if the `persist: true` setting is configured for that variable.

Credits

Resets the value of the `credit_units` machine variable in your `machine_vars.yaml` file in the `/data` subfolder of your game to zero.

High Scores (HSTD)

Resets all values for game scores that are being monitored as configured in the `categories:` section of your `high_score.yaml` mode configuration. All scores stored in the `high_scores.yaml` file in the `/data` subfolder of your game will be reset to teh `defaults:` section of your `high_score.yaml` mode configuration.

Game Audits

Resets all counters for game elements that are being audited as configured in the `auditor:` section of your game configuration. All counters will be reset to zero in `audits.yaml` in the `/data` subfolder of your game.

Adjustments

MPF provides an **Adjustments** function that presents all of your configured game variables in the `Settings` section to the user to modify certain elements of the game. This menu option is available from the `service_menu_selected` event with the label `Adjustments Menu`.

Audits

(To be completed)

Diagnostics

MPF provides a **Diagnostics** function that allows the user to test hardware elements of the game such as switches, lights and coils. This menu option is available from the `service_menu_selected` event with the label `Diagnostics Menu`.

Operator Settings

Help us to write it

CHAPTER 16

Tools

There are several tools that have been created to help you build your game in MPF.

MPF Monitor

The MPF Monitor is a graphical utility you can use to interact with a running instance of MPF. See lights change in action, click to control switches, and lay out everything on an image of your playfield.

"Interactive" MC (or "iMC")

The interactive MC lets you create YAML configurations for slides and widgets in realtime and see them on a display. This is great for fine tuning and tweaking your slides.

Service Cli

The MPF service cli is a fast way to debug or troubleshoot your machine during development and operation.

Build Production Configs

A command to prepare production config bundles.

Lightshow Creator

A lightshow generator for MPF.

Language Server in Your IDE

IDE support for your editor to support auto-complete for MPF configs.

MPF format

Reformat your MPF config files.

MPF test

Run single file tests to reproduce problems or verify behaviour.

MPF test

Commands to debug, upgrade or benchmark your hardware.

Machine Fuzzer

Fuzz your machine using afl to find crashes in MPF, your config or your code. Currently not documented. Let us know if you want to use it.

Hardware Debugger

The hardware debugger allows you to scan all your configured hardware platforms. In some cases it also supports firmware updates and configuration settings. See *mpf hardware* for details.

Future Tools

- GUI config builder
- Music builder / looper / manager
- Show builder
- Slide / animation tool
- Auto machine documentation builder
- Device / asset explorer (Why did this sound stop? Why is this LED red? etc)

The MPF Monitor

The MPF monitor is a graphical app that connects to a live running instance of MPF and shows the status of various devices. (LEDs, switches, ball locks, etc.) as well as a running list of recent *MPF events*. You can add a picture of your playfield and drag-and-drop devices to their proper locations so you can interact with your machine when you're not near your physical machine and/or for developing your game. MPF Monitor is also great when you have more than one person working on your MPF code but your physical machine is at one person's house. :)

The MPF Monitor can run on Windows, Mac, and Linux. It uses PyQt6 (Python bindings for Qt6) for its visual framework.

Here's a screen shot of it in action:

Note: The MPF Monitor is *not* a full pinball simulation with physics or moving balls or anything. But it does enough that you can use it to do real work on a machine when that machine is not nearby.


Devices
1
2
autofire
ball_device
bd_drain
idle
bd_lower_...
idle
bd_plunger
ejecting
bd_top_ki...
idle
bd_top_rig...
idle
bd_trough
idle
ball_lock
diverter
drop_target
flipper
led
l_advance...
l_again
l_australia
l_backbox_1
l_backbox...
l_backbox_2
l_backbox_7
Playfield
Events
wait_to_plunge_looped
{}
balldevice_bd_drain_ok_to_receive
{'balls': 1}
balldevice_bd_drain_ball_eject_success
{'balls': 1, 'target': '<ball_device.bd_trough>'}
s_trough_enter_active
{}
balldevice_bd_trough_ball_enter
{'device': '<ball_device.bd_trough>', 'unclaimed_balls': 0, 'new_balls': 0}
sw_drain_inactive
{}
s_drain_inactive
{}
balldevice_bd_drain_ball_left
{'balls': 1, 'num_attempts': 1, 'target': '<ball_device.bd_trough>'}

Video about developing your game without hardware:

https://youtu.be/7XmIIhzEREk

Features

- Connects to a live running instance of MPF.
- Automatically discovers all the pinball mechs and devices in the game.
- Device state is updated in real time in the "Devices" window.
- MPF events and their keyword arguments are posted in real time to "Events" window.
- You can add a photo of your playfield and then drag-and-drop LEDs, lights, and switches from the device tree onto the playfield.
  - LEDs (circle icons) show their color in real time.
  - Lights (circle icons) show their brightness in real time between black and white.
  - Switches (square icons) show their state (green = active, black = inactive).
  - More device types will become "draggable" in the future.
- Left-click on a switch to "tap" it (activate & release). Right-click on a switch to "toggle" it (change its state and hold it).
- Devices added to the playfield image are saved & restored when you restart the monitor.
- Window sizes, positions, and which windows are open are remembered and restored on next use.
- You can start the monitor and leave it running, and it will automatically connect (and disconnect/reconnect) to MPF as MPF starts and stops.

Road Map Features

MPF Monitor is *very* rough at this point. (Really more of a proof-of-concept.) We plan to add more features, including:

- More details for events, including listing registered handlers & making it so you can sort, search, and clear the list.
- Adding all the "game logic" stuff, including modes, shots, shot groups, shot profiles, logic blocks, timers, ball locks, multiballs, achievements, etc.
- Add shows (running shows, step they're on, priority, etc.)
- Add players information (show all player variables and their values)
- A "snapshot" button that can dump the entire current state to a file for debugging later
- Export position (x/y) settings of widgets back to the MPF config
- Connect to MPF-MC to get information about slides, displays, widgets, etc.
- Add color controls to the playfield image to set brightness and color saturation
- Add buttons to enable/disable different types of devices (think of it like "layers" for the playfield image.
- Show additional properties from the selected device (Click a device to see it's full information.)

- Change debug levels of various devices dynamically
- Save the config / layout with a specified file name
- Add multiple playfield views which could each have different devices
- Set colors, shapes, rotation, & sizes of devices (so inserts can be the right shape). Allow configurable "off" colors which can include opacity and "glow" so inserts look like real lights.
- Allow all devices to be added to the playfield image, with custom representation (diverters that animate, flippers that animate, etc.).
- Device state change history that shows what properties changed and when.
- Default (mostly blank) playfield image if no playfield image is specified
- Configurable default options (folder location, playfield image name, etc.)

Next Steps

Installing MPF Monitor

The MPF installer was rewritten for this current release (0.56).

MPF Monitor 0.56 now requires PyQt6. (Prior versions of MPF Monitor required PyQt5.)

```
pip install mpf-monitor
```

Running the MPF Monitor

1. Make sure you installed MPF Monitor first. (You need to actually *run the installer*. You can't just run the monitor from the download folder.)
2. Create a subfolder in your MPF machine folder called `/monitor`
3. Put an image of your playfield in that folder named `playfield.jpg`
4. Run MPF monitor from a command prompt via the command `mpf monitor`. Be sure to run this from your machine folder (the same place where you run `mpf both`).
5. In a new terminal window, Start MPF and MPF-MC. You can start MPF before or after monitor is started, and leave the monitor running while MPF is not.
6. MPF Monitor has multiple windows that can be viewed, though not all may be enabled by default. The "Inspector" window is the main window where you can toggle other windows On and Off. To enable different windows, click on the "Monitor" tab. This will show you a list of all the different windows you can enable and view:
    1. Show device window (this window lists all your switches, shots, targets, etc)
    2. Show event window
    3. Show playfield window (this window shows your playfield picture)
    4. Show mode window
7. MPF Monitor should connect to MPF and populate the devices tree in the device window. You can look through this list to see the states of various devices. The columns in each window are sortable and resizeable.

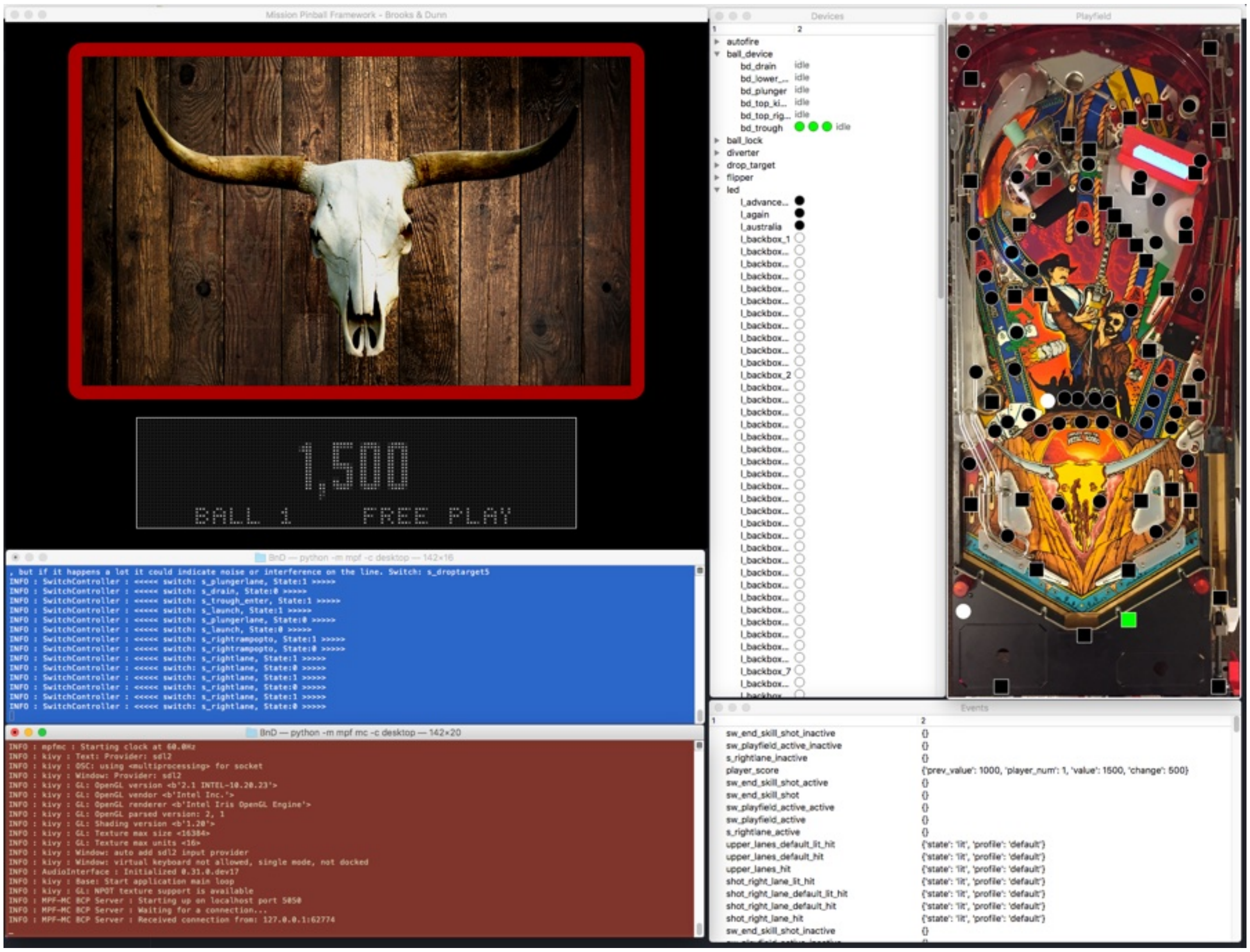
8. You can drag-and-drop switches and LEDs from the Devices window onto the playfield image. When you do this, a config file called `/monitor/monitor.yaml` will be created. If you open that file, you'll see that x/y values of devices are stored in percentages instead of pixels, so they should stay in the right place even if you change your playfield image. The file is updated automatically. You can drag devices that you previously placed on the playfield too (there's a half- second delay so you don't accidentally move something when you're clicking on it).
9. Edit `monitor.yaml` to remove devices from the playfield you don't want anymore.
10. When you resize or reposition one of the monitor windows, the window positioning information will be stored, so the monitor can restore the layout the next time you run it.

Understanding MPF Monitor folders & files

Here's what your machine folder structure will look like when you're using the monitor:

Using the MPF Monitor

We designed MPF Monitor so that all the windows are separate (instead of a main "parent" window), meaning you can resize them all however you want and close the ones you don't need. The idea is that you can keep the monitor running off to the side and still see your MPF display window as well as the terminal windows, like this:



Running with "virtual" hardware

You can use the MPF Monitor with or without a physical machine attached.

If you have a physical machine connected, be careful when toggling switches, since it can really confuse things if a ball is sitting on a switch in your machine and then you use the Monitor to tell MPF that the ball isn't really there. :)

Still though it's nice to be able to "peek inside" the inner workings of MPF even when it's connected to a physical machine, and the Monitor is great for that.

You can also use MPF Monitor with no hardware attached using one of MPF's *virtual platforms*. Specifically the *smart virtual platform* works great if you're using MPF without physical hardware.

Modifying switches and lights on your playfield

More information on the usage of MPF Monitor (0.54+) can be found in *Playfield Devices and Using Device Inspector*.

Playfield Devices

device In the context of MPF Monitor, a **device** refers to a switch, light, or diverter.

Adding devices to playfield

1. Locate the *Devices* window.
2. Locate the light or switch you want to add to the playfield.
3. Drag device to image of playfield.

Note: You can use the search box to filter to the name you are looking for.

Changing the default size of all devices

1. Ensure Device Inspector is disabled.
2. Change the size slider or spinbox.

Note: Any devices that have manually been resized will not be affected by the default size changes. You can reset this for a device by selecting the device and clicking "Reset to Defaults".

Sorting and filtering devices

- To filter devices, type your keyword in the device search box.
- Sorting devices:
  1. Latest received **Default**
     - Should match order of MPF config file
  2. First received
  3. Alphabetical, increasing:
     - Useful when placing ordered targets, ie: *"ltarget1"*, *"ltarget2"*, *"ltarget3"*...
  4. Alphabetical, decreasing

Using Device Inspector

Use Device Inspector to modify your playfield devices without sending switch hits to MPF

Enabling Device Inspector

1. Locate *Inspector* window.
2. Enable Device Inspector by clicking the button labeled "Toggle Device Inspector".
3. Device inspector is enabled. The button will stay "clicked" as an indicator. The *Playfield* window title will change to "Inspector Enabled - Playfield".
4. Changes are saved automatically.
5. Disable Device Inspector by clicking again on "Toggle Device Inspector".

Note: While device is inspector is enabled, clicks on switches will not be sent to MPF.

Viewing the name of a device

1. Enable Device Inspector.
2. On the playfield, select the device you want to view.
3. The name of the device will be shown below the "Toggle Device Inspector" button.

Changing display properties of device

Depending on your image dimensions switches and lights might be a little small or too large. You may also want your device to display as a different shape, or rotated to match an insert. You can change the size, rotation, and shape of a device.

1. Enable Device Inspector.
2. Click on the device you want to change.
3. Change size, shape or rotation by changing options in the inspector.

Note: While device is inspector is enabled, clicks on switches will not be sent to MPF.

Deleting devices from the playlist

1. Enable Device Inspector.
2. Click on the device you want to delete.
3. Click the delete button in inspector.

Resetting a device to its defaults

If you would like to clear your changes to a device's parameters, you can reset all of them by selecting the device and clicking "Reset to Defaults".

1. Enable Device Inspector.

2. Click on the device you want to reset.
3. Click the “Reset to Defaults” button in inspector.

Warning: It is not possible to undo resetting a device to its defaults.

Interactive MC (iMC)

The MPF MC package includes an “interactive” MC which you can use to live-edit YAML configurations for slides and widgets and see the results in realtime in your on-screen window.

Running the iMC does two things:

1. It launches the MC like normal, loading your game’s config files.
2. It launches a second window which has a multi-line editable text box where you can type or paste slide configs.

The idea is you can use the iMC to keep tweaking and fine-tuning your slide and widget settings in a way that’s much easier than starting your game and going through your game to find the slide you’re looking for.

Note: The iMC does not connect to physical hardware, so if you have a physical DMD then you will have to test with an on-screen virtual DMD.

Since the iMC uses the regular MC and the regular config files, you have access to all the named widgets, images, videos, widget styles, fonts, etc. from your machine config.

See *mpf imc command*.

Service Command Line

The MPF service cli is a fast way to debug or troubleshoot your machine during development and operation.

1. Start your game (e.g. using `mpf both`)
2. Start the service cli from within your game folder using `mpf service`.

Your game will go into service mode and you can run diagnostics commands. Once you are done the game will continue and exit service mode. You can use tab to complete commands and arguments.

Commands

list_coils

List all coils in the machine.

coil_pulse <name>

Pulse coil <name>.

coil_enable <name>

Enable coil <name>. This only works if enable is allowed for this coil.

coil_disable <name>

Disable coil <name>.

list_switches

List all switches in the machine.

monitor_switches

Watch for switch changes. Prints any changes until you press Ctrl+c.

list_lights

List all lights in the machine.

light_color <name> <color>

Turn light <name> into color <color>.

light_off <name>

Turn light <name> off.

exit/quit

Exit service cli. Game will reset and start.

See *mpf service command line reference*.

MPF Showcreator

MPF supports playing *light shows* out of files in your config folder. Those are human readable and can be created by hand. But isn't that a bit cumbersome for larger shows? Especially, if you want to swipe over all (or most) of your LEDs this might take days. Luckily, there is a tool for that.

The light show generator for MPF loads your LED positions from the *The MPF Monitor* config and lets you create show for transitions.

Video about showcreator:

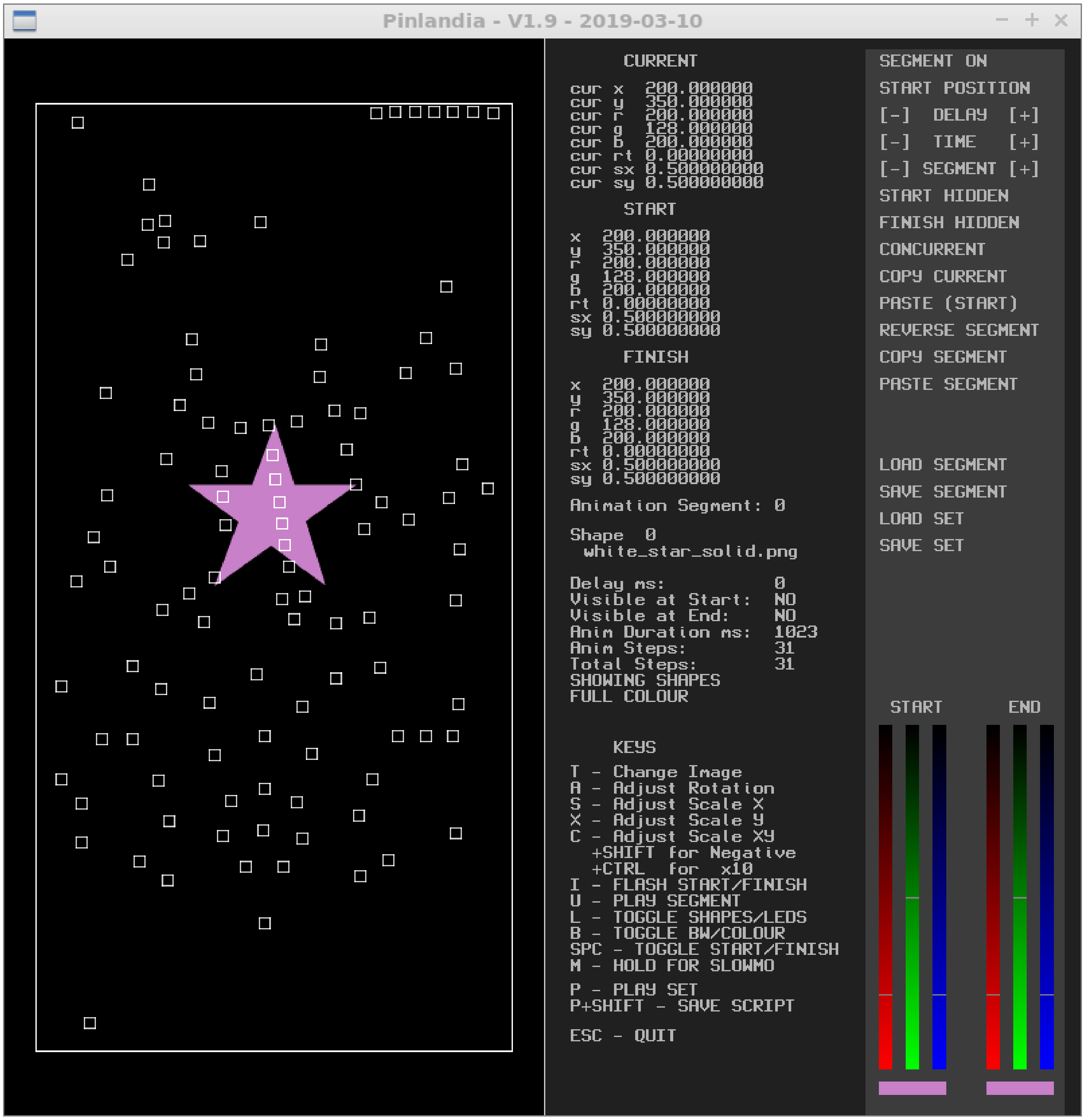
https://youtu.be/bjDWm_pO9_I

Shows in MPF are written in YAML and can be used *universally* to control all kinds of things (such as *lights*, *coils*, *slides*, *widgets*, *sounds* and *more*). Basically, shows are a list of actions combined with a duration after which the next element in the list is played. Here is an example of a light show with three lights which sequentially turn blue over one second:

```
##! show: my_show
#show_version=5
- duration: .25
  lights:
    l_arrow_1: off
    l_arrow_2: off
    l_arrow_3: off
- duration: .25
  lights:
    l_arrow_1: blue
    l_arrow_2: off
    l_arrow_3: off
- duration: .25
  lights:
    l_arrow_1: blue
    l_arrow_2: blue
    l_arrow_3: off
- duration: .25
  lights:
    l_arrow_1: blue
    l_arrow_2: blue
    l_arrow_3: blue
```

In this simple example it totally makes sense to create the show by hand. You could also throw in *tokens* for the lights and reuse the show all over the machine for different light triples.

However, imagine you want to swipe over all lights in your machine. That would be a lot of text and also hard to get right manually. Luckily, Mark, the maker of the Nightmare before Christmas custom pinball machine, created this awesome MPF Lightshow generator.

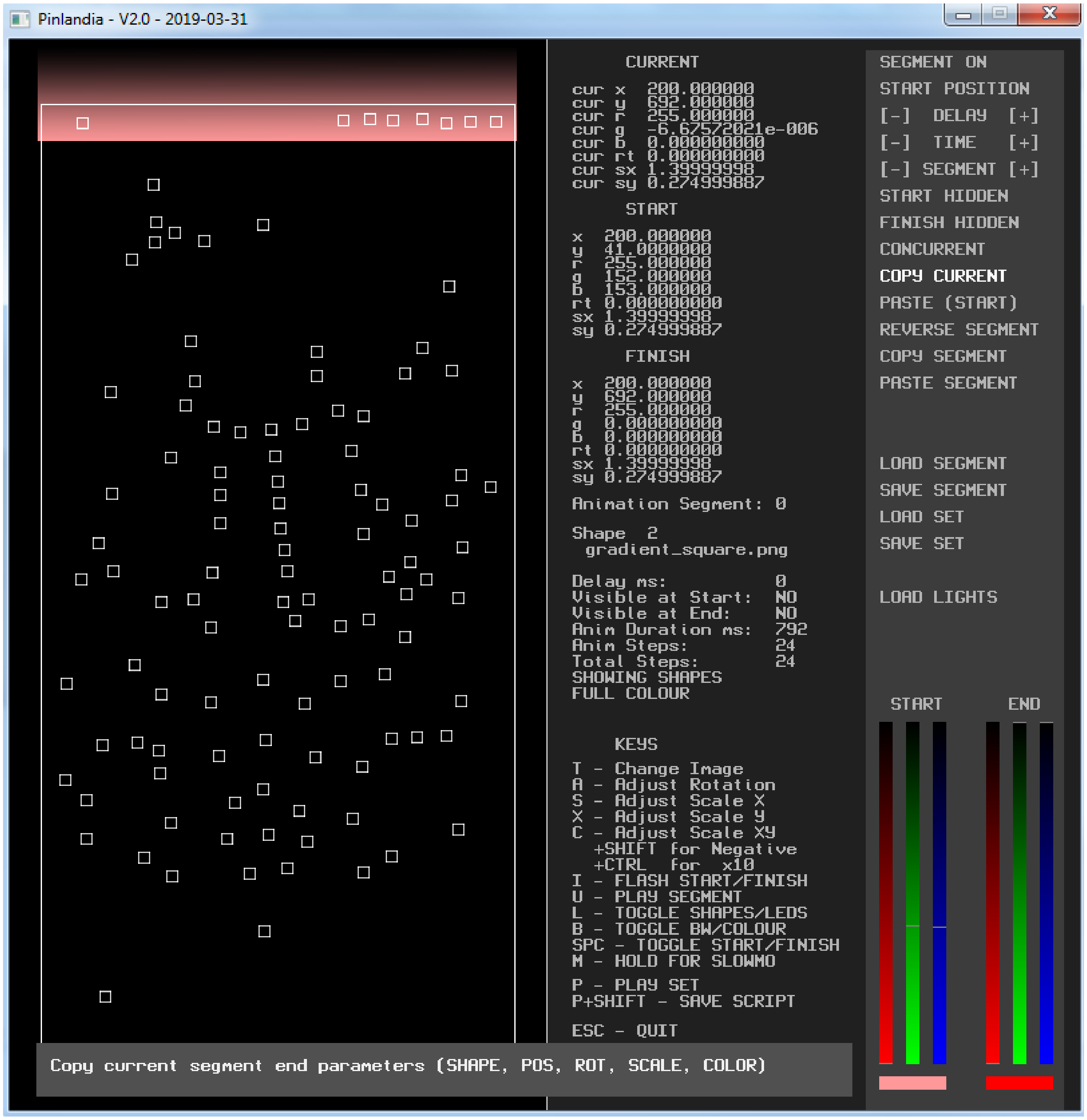


The tool allows you to set a shape (i.e. a star in the example), choose a start and an end position and color. Based on that it will create a light show for you which contains one section per step (at a defined frame rate). Neat right? You might ask: How does it know where my lights are located on the playfield?

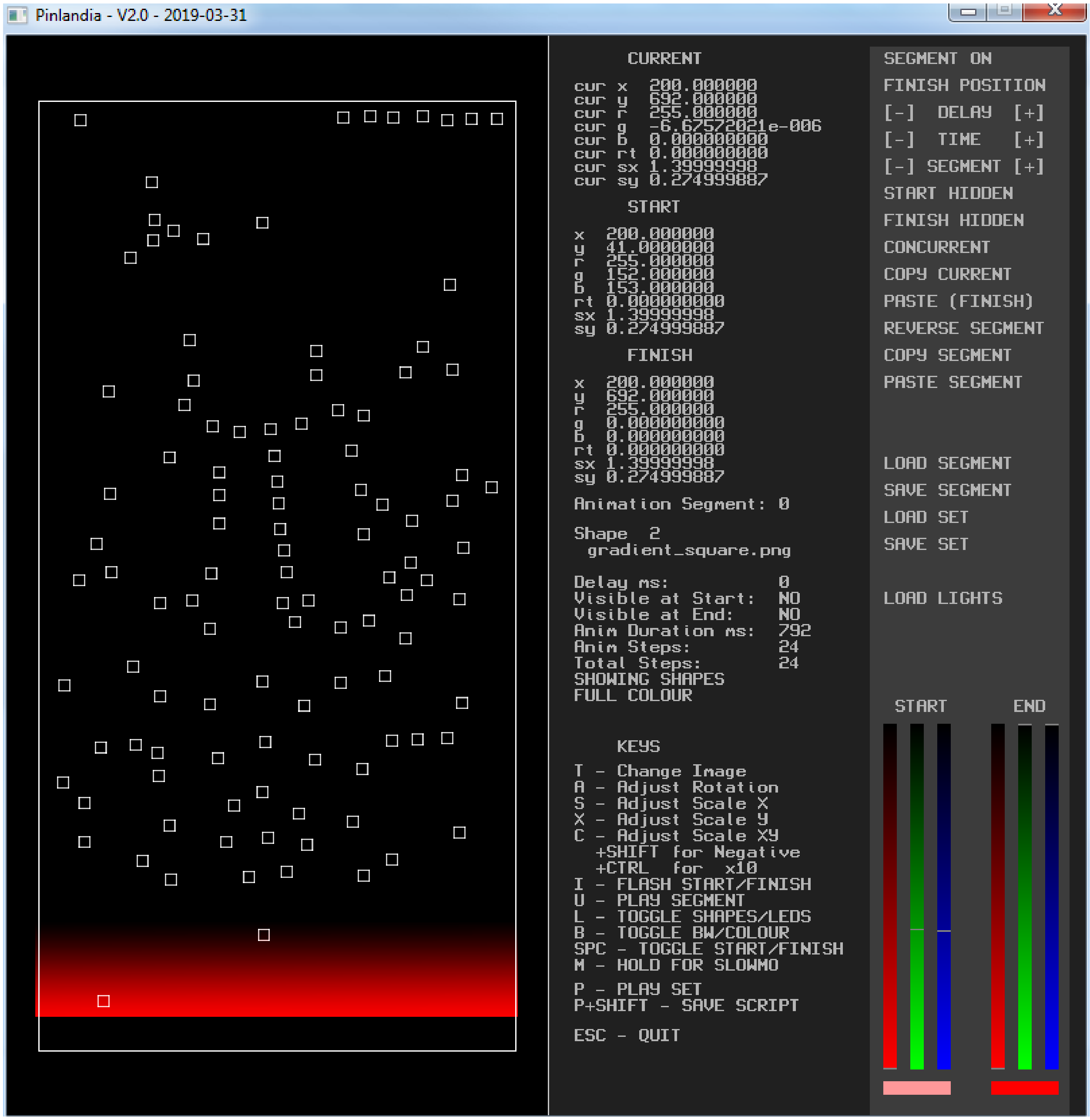
Luckily, you probably already have them set if you used the *MPF Monitor*. It allows you to use drag and drop to position all your switches and lights on a playfield image. Those positions are then saved to the `monitor/monitor.yaml` file in your machine folder. All you have to do is point the light show creator to the `monitor/monitor.yaml` file on startup.

You set the start and end positions, rotations, scales and colors of that shape anywhere you want over the playfield.

Here we start with a gradient bar at the top of the playfield in a pink color.



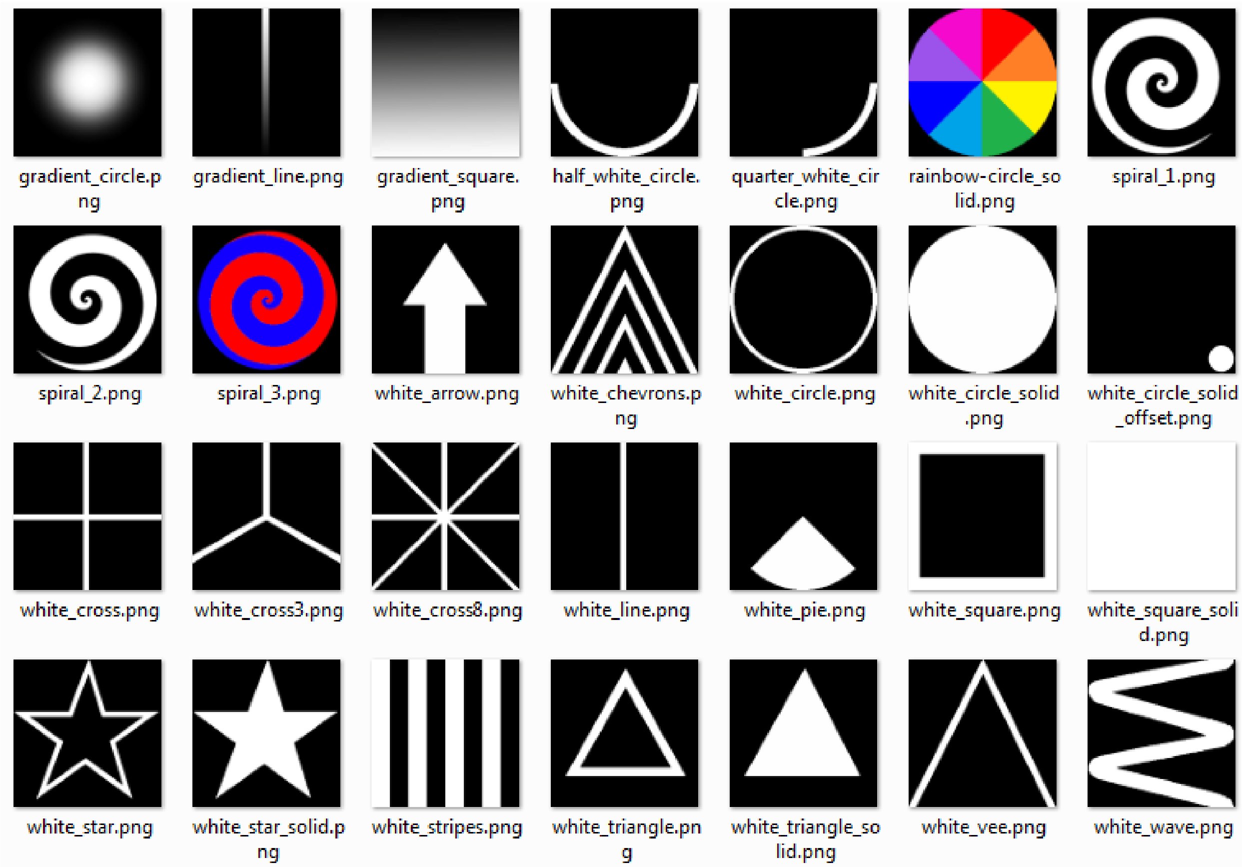
We want the final position to be here at the bottom, in a darker red shade.



You can then adjust the length of the animation in milliseconds and hence the number of steps in the final show. In this example, the shape will be moved from the start to finish in 24 steps.

Based on these settings, it will create a light show for you which contains all needed commands per step for each of the lights the shape passes over. Lightshow playback speed can be adjusted in MPF.

You're not restricted to just the included shapes. You can make your own shapes and drop them in the shapes folder.



Once you get the hang of animating a single shape, you can go further by adding in more shapes. You can add a total of 256 shapes in animation segments. Each segment can be set to `concurrent` (start and end same time as the previous segment) or `follow` (start after previous segment) This allows for more interesting multipart shows. For example you could have several color swipes coming from different directions one after the other or effects like multiple spotlights moving across the playfield like a hollywood premiere.

Running the showcreator on Windows

1. Checkout or download the showcreator repository.
2. Double click on led.exe

Compiling and running the showcreator on Ubuntu

Inside a new install folder:

```
# inside a new install folder
apt install linux-libc-dev:i386 libxft2:i386 g++-multilib gcc-multilib libxpm-dev:i386 libxxf86vm-
↪dev:i386 libgl1-mesa-dev:i386 libglu1-mesa-dev:i386
git clone https://github.com/blitz-research/blitzmax.git
cd blitzmax
cd _src_/linux
./install.bat  # yes its .bat
```

(continues on next page)

(continued from previous page)

```
cd ../../../   # back to your src folder

git clone https://github.com/missionpinball/showcreator.git
cd showcreator
../blitzmax/bin/bmk makeapp led.bmx

# run it
./led
```

Afterwards you can run the showcreator using (from within your install folder):

```
./showcreator/led
```

Key bindings

- A - adjust rotation
- S - adjust x scale
- X - adjust y scale
- C - adjust both x and y scales
- HOLD SHIFT to reverse above functions
- HOLD CTRL to increase functions by 10X
- I - flash between START and FINISH end points
- L - toggle between viewing SHAPES or affected LEDs
- B - toggle between BW and full colour output
- B+SHIFT - change the B/W Threshold (16-240)/256
- SPC - toggle between START and FINISH end points
- U - play segment
- P - play complete set
- M - HOLD for slow motion during segment/set play
- P+SHIFT - play set and create script file
- ESC - quit - Y/N confirm quit
- Left Mouse Button Down over playfield adjusts position of current end (START or FINISH) +SHIFT adjusts both START AND FINISH positions

Dynamic Shows

The tool is handy to render static shows which will not change during runtime. If you want to render shows dynamically (using your GPU) you can also use *your lights as display in MC* but that will cost much more resources during runtime than offline generated shows.

Using the MPF Language Server in Your IDE to Edit Configs

The MPF language service implements the language server protocol (LSP) to bring syntax highlighting, auto completion, diagnostics and more to numerous IDEs (and not just to one of them). Your IDE most likely supports LSP either directly or via a plugin. Even some text editors (such as Sublime) support LSP.

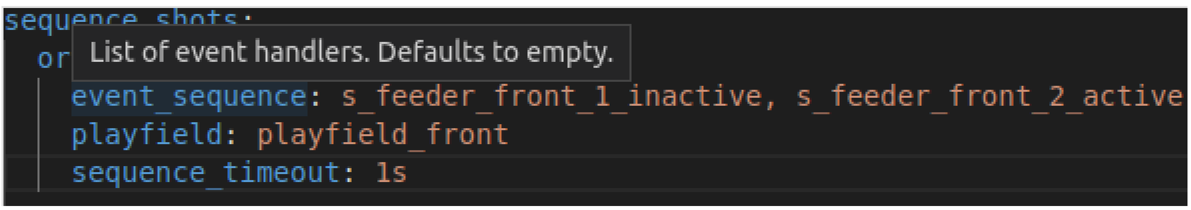
You can also follow our video about the perfect IDE setup:

https://youtu.be/QdDHEe2aEJo

Features

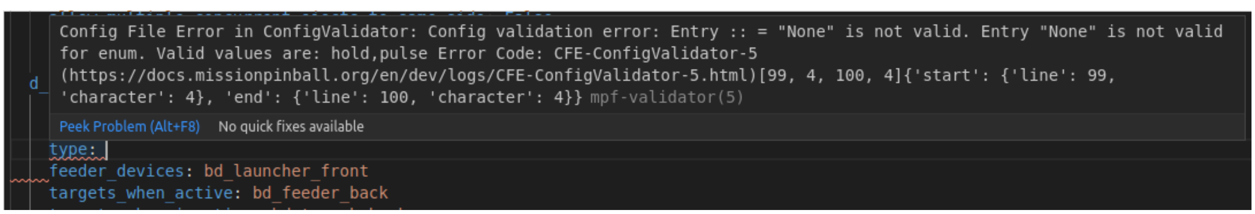
The MPF language server helps you to efficiently write MPF config. In the following you find a selection of the features.

Context Help

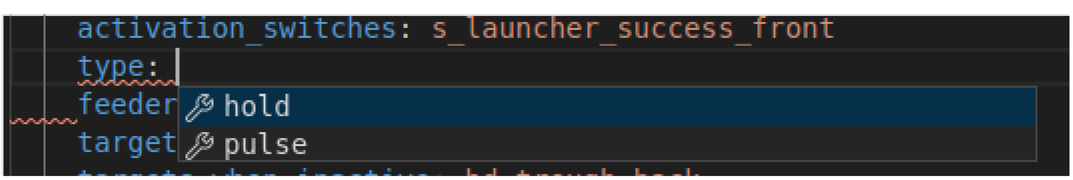


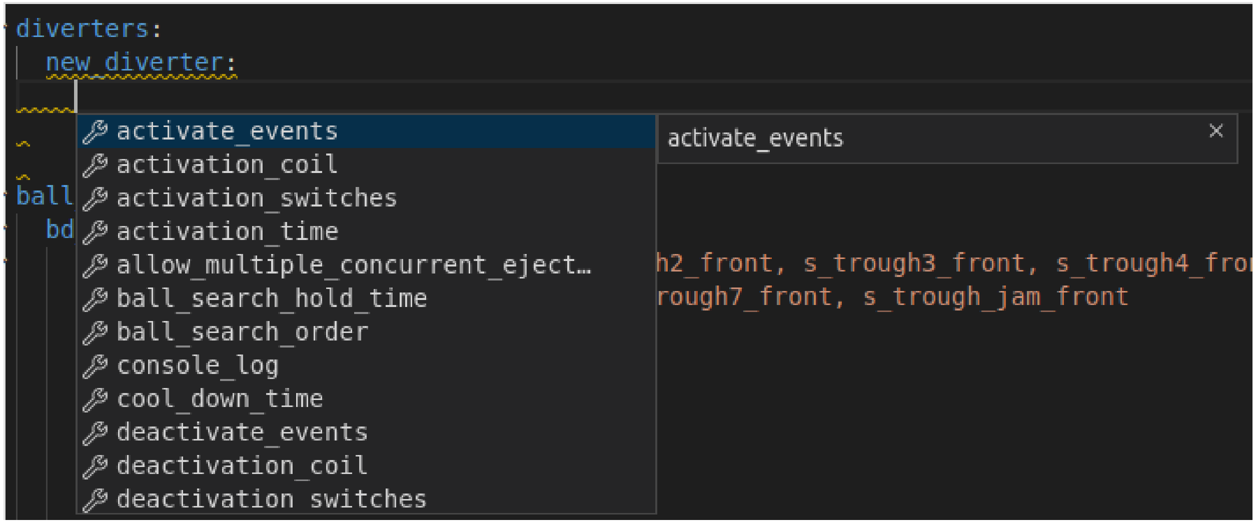
Hover over a setting and the LSP will give you context about the type. In the future this will also show you the documentation entry about this setting.

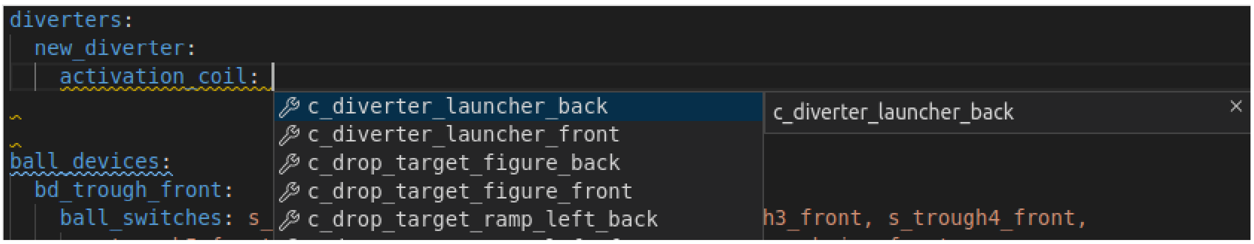
Error Highlighting



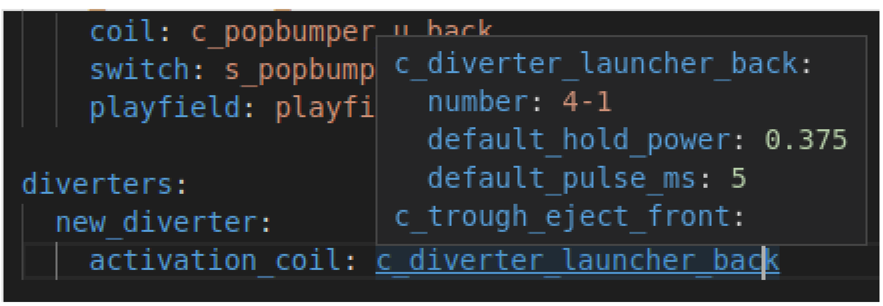
Auto Completion







Go To Definition



Installation

See the Language Server Documentation for now.

Build Command Line

The build command line `mpf build` can compile configs for production.

Commands

production_bundle

Call this inside your machine folder. It will create `mpf_config.bundle` and `mpf_mc_config.bundle` inside your machine folder. Those two files contain the complete configuration including all shows for your machine. If you change any configs, modes or shows rerun this command. Make sure that your final machine runs exactly the same version of MPF or bad things will happen. Regenerate those files when upgrading MPF (even when not changing configs).

See *Tuning Software for Production* for details about production machines.

Format And Lint Config Files

The command line `mpf format` can reformat your MPF configs.

Run it using `mpf format path/to/your_file.yaml`. It will show you a preview of the changes it your make:

```
$ mpf format config/config.yaml
Parsing single test config/config.yaml.
Config is not linted.
---
+++
@@ -1,13 +1,13 @@
 #config_version=5

 config:
-- shots.yaml
-- switches.yaml
-- coils.yaml
-- devices.yaml
-- leds.yaml
-- slides.yaml
-- sound.yaml
+  - shots.yaml
+  - switches.yaml
+  - coils.yaml
+  - devices.yaml
+  - leds.yaml
+  - slides.yaml
+  - sound.yaml

 mpf:
   device_modules:
@@ -203,7 +203,7 @@
       0.54: servo_pos2
     ball_search_min: 0.35
     ball_search_max: 0.55
-    debug: True
+    debug: true
   servo_figure_back:
     number: servo_back-64-0
     reset_events: machine_reset_phase_3
@@ -217,8 +217,7 @@
       0.31: servo_pos2
```

(continues on next page)

(continued from previous page)

```
    ball_search_min: 0.1
    ball_search_max: 0.3
-    debug: True
+    debug: true

Not writing back changes. Use --yes to do this.
```

You can add --yes to the commandline to apply the changes.

Run Single File Tests

The command line `mpf test` can run single file tests or doc pages.

You can create a text file which contains the main config, shows and modes of your test machine. Then at the bottom you can create some test assertions.

The structure look like this:

```
# your machine-wide config here. That is what is normally in config/config.yaml.

# you can have a few modes
##! mode: some_mode
# mode config here

# you can have a few modes
##! mode: another_mode
# mode config here

# additionally you can have separate shows
##! show: some_show
# show here

# now you can add a test
##! test
#! start_game
# run the machine for 1 virtual second
#! advance_time_and_run 1
# post an event
#! post some_event
```

All test assertions are defined in MpfDocTestCase. Just remove the `command_` prefix and you are good to go.

MPF Hardware Command

MPF allows has a few command line actions to check and update your hardware:

mpf hardware scan

You can run *mpf hardware scan* to get an overview over the connected hardware. MPF will try to enumerate all connected boards and tell you what it know about your hardware. The output varies per hardware platform from almost nothing to a lot.

mpf hardware firmware_update

MPF will try to upgrade the firmware of your hardware if this is supported for your hardware. There will probably be specific configuration in your hardware platform section to enable this.

mpf hardware benchmark

Overview video about mpf hardware benchmark:

https://youtu.be/uRT--368J6A

MPF will benchmark latency and jitter of inputs, outputs and rules for your hardware setup (i.e. your controller with your OS and hardware). This needs to be configured:

```
switches:
  s_test1:
    number:
  s_test2:
    number:

coils:
  c_coil1:
    number:
  c_coil2:
    number:
    allow_enable: true

flippers:
  f_flipper:
    activation_switch: s_test1
    main_coil: c_coil2

hardware_benchmark:
  coil1: c_coil1
  coil2: c_coil2
  switch1: s_test1
  switch2: s_test2
  flipper: f_flipper
```

Disconnect or disable high voltage. Then connect *s_test1* to *c_coil1* and *s_test2* to *c_coil2*. MPF will enable the flipper *f_flipper* which will create a hardware rule on *s_test1* to pulse *c_coil2*. Afterwards, MPF will pulse *c_coil1* which should then activate *s_test1*. In turn the hardware rule should pulse *c_coil2* which then activates *s_test2*. Hardware benchmark will measure the timings of the two switches. It will repeat this procedure a few times and run some statistics on the results.

CHAPTER 17

Testing your machine

This section moved to the *developer documentation*.

CHAPTER 18

Finalizing your machine

This section will discuss all the "final" steps you need to take to get your machine ready to run without you.

Most of this is unfinished: See *Help us to write it*. Also let us know in the forum if you have any questions about this.

Tuning Software for Production

Run MPF in production mode

YAML is quite slow to parse and reading configs dominates the startup time of MPF and MPF-MC. This is mostly fine during development and we can partially mitigate the costs by caching. However, things are different when running a production machine as caching will not work on a cold boot with a typical read-only setup. Usually production machine setups use less beefy computers with slower disks which makes thinks even worse.

Starting with version 0.54 MPF has a production mode which will use pre-compiled config bundles for much faster start-up times. Additionally, this will disable some expensive config and runtime validations to increase performance. Furthermore this will reduce the amount of debug output.

First run `mpf build production_bundle` which will create `mpf_config.bundle` and `mpf_mc_config.bundle`. You have to recreate those files after every config, mode or show change. Those bundles include all yaml files but not any other assets (such as videos or sounds). Second, add the `-P` flag to the commandline to run MPF in production mode.

MPF will also try to keep running in some cases instead of exiting the game. This will not be helpful to find bug but a when you ship machines you won't see the log anyway. Finally, MPF will try to initialize for `30s` and then exit in case something went wrong. You can use that to run MPF in a while loop or to reboot your PC in case initialization went wrong.

Run MPF without text UI

Text UI costs some performance so disable it in production or on less powerful hardware in general. You can do this by adding the `-t` flag to the MPF commandline.

Install the latest Python version

For instance, MPF runs significantly faster on Python 3.6 than on 3.5. Similarly, 3.5 is faster than 3.4. We expect the same for the next releases. You might not need this if you are using PyPy.

Install uvloop

When running MPF on linux install `uvloop` will reduce latency and increase throughput for I/O operations. This will keep your game responsive:

```
pip3 install uvloop
```

MPF will use uvloop once it is available. Requires at least Python 3.7 for the latest version.

Run MPF with PyPy

PyPy is a replacement for standard Python. It uses a Just-in-time compiler that makes it facter and often use less memory. PyPy does not support all Python code, and currently works for MPF, but not for MPF-MC because kivy is not yet compatible with PyPy. Performance and latency improvements are around 10x in our benchmarks so this might be essential on low-end hardware. Download PyPy and install it. Since PyPy is a separate Python environment you need to install pip and reinstall all pip packages for PyPy.

```
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
pypy get-pip.py
pypy -m pip install mpf
```

Afterward, you can start MPF within your game folder using:

```
pypy -m mpf game
```

Instead of the `mpf` command just use `pypy -m mpf`. For `pip` use `pypy -m pip`.

You still need to run MPF-MC using normal Python. This might change in the future.

Some random hints

- Optimize assets for your hardware. Audio should have the same sampling rate as your hardware is using. Images and videos should be at native resolution to prevent scaling up or down.
- Re-encode your videos in a codec which can be efficiently decoded on your target hardware.
- Let us know if we missed something here.

Choosing a computer to run MPF

Please make sure you read the *Choosing a PC for MPF* section first.

In this section we talk about a potential production setup. Thus, this is mostly about compromises. What is the minimal (e.g. most cost effective) hardware? You probably want to tune your game first.

Help us to write it

Single-board versus "real" computers?

Picking an OS

The checklist

Now that you've read about all the background information that goes into picking a host computer, let's break it down into the questions you need to answer to pick the one that makes sense for you.

What OS are you familiar with?

More and more commercial machines are running Linux. But if you're comfortable with Windows and you've never used Linux, then by all means do not put a Linux computer in your pinball machine. It's just not worth the headache. Sure, this might mean that you have to buy a $150 motherboard/SSD/RAM/PSU combination versus a $50 single board computer, but meh, that 100 bucks will be worth it in terms of future pain avoided. And besides, pinball machines cost thousands of dollars to build. What's another 100 bucks to make your life easier?

Do you have anything you can use now?

The best host computer is the one that you already have. :) Seriously, if you have something laying around, just start using it. You can always change it out later. BTW, we've received a few questions from people wanting to use Mac Minis.

Is this a one-off machine, or are you taking something into production?

What are your graphics and display requirements?

The Bottom Line

Remember that MPF and Python work identically regardless of whether they're running on Windows, Mac, or Linux. So even if you pick the "wrong" host computer now, you can always change it out later without having to change any of your code or configuration files. So if you have an old laptop sitting around then go ahead and use it for MPF. You can always swap it out with a small single-board computer down the road.

Choosing an OS for your final machine

Help us to write it

Talk about "Freezing" it, lock down, recovery, auto booting, etc.

Controlling your machine & computer power on / power off

Unlike computers pinball machines are not expected to have a shutdown procedure. Users tend to just turn off the power which might cause problems with your operating system and filesystems.

Two general approaches exist here:

Computer Start-up and Shutdown Controller

Scott Danesi sells a board called Computer Start-up and Shutdown Controller (CSSC) which will trigger a shutdown of your PC when the main power supply of your machine is turned of (part number: #600-0322-00). However, you need to make sure that the PC still has power until shutdown is complete. You can either use a separate outlet (and make sure not to disconnect it early) or add an Uninterruptible power supply (UPS) to your machine.

This solution is very useful during development and early prototypes. Especially if you are using (older) Windows which very much dislikes unclean shutdowns. However, with modern operating systems and journalling filesystems (such as ext4 or ReFS) this became less of an issue.

Make it work on Linux

If you use a Linux distribution with systemd set `HandlePowerKey=ignore` in `/etc/systemd/logind.conf`.

To handle power button events install `acpid`. Add `/etc/acpi/events/powerbtn` with the following content (or change it if if already exists):

```
event=button[ /]power
action=/sbin/poweroff
```

Restart `acpid` (or your computer) and you should be good to go.

Read-only Filsystems

When you finished your machine you will usually run in on Linux on an embedded PC. Multiple solutions exist here such as OpenEmbedded/Yocto. At that point you will usually have a build process which builds an image which is then deployed to your target PC (via a SD card or flash process). This image will be mounted read-only and cannot get damaged by a crash.

Furthermore, you often add one partition to store audits/highscores and sometimes logs. It is recommended to use a journaling filesystem for this partition and expect it to break. Usually, there is some kind of reset mechanism to wipe this partition in case it gets corrupted (sometimes automatic in case it can no longer be mounted).

We are happy to discuss those topics in our forum (and extend this section as a result of that).

Using MPF to Shutdown a Computer

While the above two methods are the best ways to power your computer on or off, there may be times when you want to use MPF to shutdown your computer.

For example, if you're developing a DMD-based game and don't have a computer monitor attached, you can use MPF to safely shutdown your computer.

Create a mode called `shutdown_computer` and create a `/code` subfolder and a `/config` subfolder.

Create a `shutdown_computer.py` file in the `/code` folder with the following code:

```
from mpf.core.mode import Mode
import os
import platform
class shutdown_computer(Mode):
    def mode_init(self):
        self.log.info('shutdown_computer mode_init')
        self.OS_type = platform.system().lower()
    def mode_start(self, **kwargs):
        self.log.info('shutdown_computer mode_start')
        self.add_mode_event_handler('shutdown_host_computer', self.shutdown_host)
    def shutdown_host(self, **kwargs):
        #shutdown the mpf game if it's running
        #shutdown the computer
        if self.OS_type == 'linux':
            shutdown_str = 'shutdown -t 0'
        elif self.OS_type == 'windows':
            shutdown_str == 'shutdown -s -t 0'
        else:
            self.log.warning(f'Sorry this feature is not available in {self.os_type}')
            return
        os.system(shutdown_str)
    def mode_stop(self, **kwargs):
        self.machine.events.post('shutdown_computer mode_ended')
        self.log.info('shutdown_computer mode_stop')
```

Create a `shutdown_computer.yaml` file in the `/config` folder with the following code:

```
##! mode: shutdown_computer
#config_version=5
mode:
  start_events: mode_base_started
  stop_events: shutdown_mode_cancel
  priority: 400
  code: shutdown_computer.shutdown_computer

combo_switches:
  shutdown_hold:
    switches_1: s_left_flipper
    switches_2: s_start
    hold_time: 5s
    events_when_both: shutdown_host_computer
```

Enable the mode in your machine config file.

The above config is an example on how you could shutdown the computer. This example requires you to hold down the left flipper and start button together for five seconds, then the computer will shutdown.

You can change this and use the shutdown_host_computer event to shutdown your computer as you like.

Enabling & fine-tuning ball search

Help us to write it

Fine-tuning ball device timing

Related Config File Sections
ball_devices:

The default timeouts in *ball_devices* are very conservative and usually too long. You might have noticed delays after the eject of the second ball when starting a multiball. This is caused by the default `eject_timeouts` setting which will cause the ball device to wait `10s` until the ball is confirmed to be on the playfield. Only after that the next ball will be ejected because before that timeout the ball may return back into the device (e.g. roll back in the plunger lane).

To minimize delays during ejects to the playfield you need to measure the maximum time the ball may take to return after an eject. Set `eject_timeouts` to that value but not lower. If you set it lower the ball may become confirmed and then you end up with two simultanious ball inside the plunger lane. In case that time is still too long you might be able to use `confirm_eject_switch` (but that might require a hardware change).

Also, please note that this only applies to devices ejecting to a playfield. If you are ejecting into another device (e.g. trough to plunger lane) the timeout does not really matter because the ball will be confirmed once it hits the target device.

Fine-tuning switches

Help us to write it

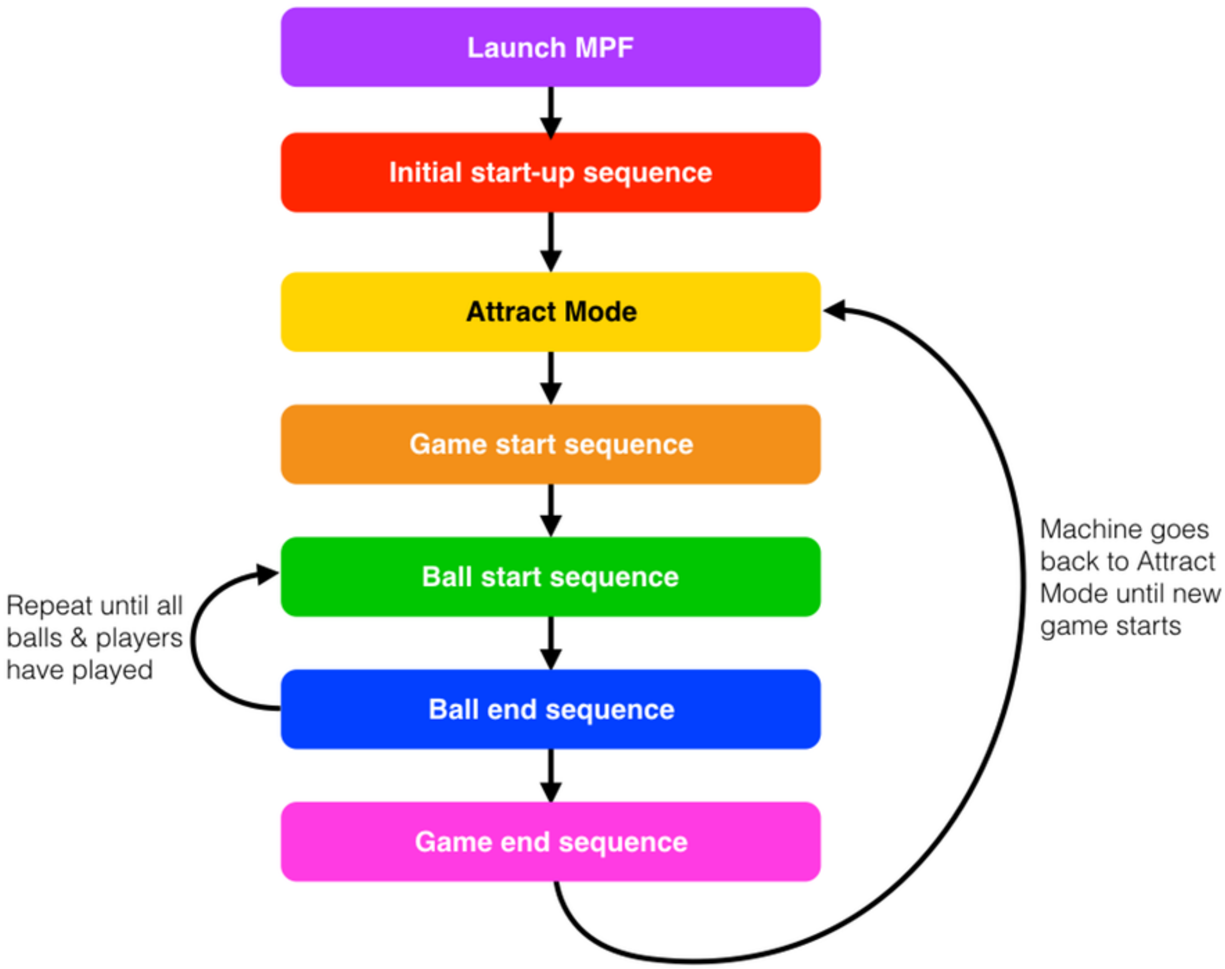
Talk about debounce, also broken switch detection, alternative workarounds, etc.

See the *switch debouncing section*.

CHAPTER 19

Flowcharts

The software that runs a pinball machine is really complex. Even though MPF hides a lot of that complexity from you, it's still helpful to know exactly what's going on under the hood. This diagram shows the high level flow. Read on to see the details of each step.



MPF Boot Up / Start Up Sequence

The first phase of operation of MPF is the start up sequence which is basically everything that takes from from the time you run `mpf` until the time your machine is up and running in attract mode. We're not going to list every single detail here—to see that just look at a log file generated in verbose mode—but this should give you a pretty high level gist:

1. Loads the configuration from file: `<your MPF project root>/mpf/mpfconfig.yaml`
2. Loads the machine config file you specified in the command line. Note that this config file may load other config files.
3. Sets the default hardware platform. (FAST, P-ROC, OPP, SPIKE, virtual, etc.)
4. Loads the system modules. The exact order is specified in `mpfconfig.yaml`. Currently it's:
    1. config_processor
    2. timing
    3. event manager
    4. mode controller

    5. Device manager
        1. Device modules are loaded
        2. Machine-wide devices are created
    6. switch controller
    7. ball controller
    8. light controller
    9. bcp
    10. logic blocks
    11. variable player/scoring
    12. shot profile manager

5. System events are registered (for things like shutdown, quit, etc.)
6. Posts the event *init_phase_1* .
    1. The event player is initialized
7. Posts the event *init_phase_2* .
    1. The ball controller configures eject targets
    2. The playfield configures eject targets
    3. Score reels configure their switches
    4. BCP sets up connections
    5. The switch controller sets up switch events
    6. The device manager registers all the control_events for machine- wide devices
8. Plugins are loaded
9. Posts the event *init_phase_3* .
    1. The ball lock devices initialize
    2. Diverters register for switches
    3. The shot profile manager registers shot profiles
10. Scriptlets are loaded
11. Posts the event *init_phase_4* .
    1. Drop targets update their states from their switches
    2. The auditor initializes
    3. OSC starts
    4. The asset managers start loading machine-wide assets
    5. The mode controller processes and loads all the modes
12. Posts the event *init_phase_5* .
    1. The light controller processes machine-wide light scripts and light player entries
13. The machine controller's `reset()` method is called.

14. Reset posts the event *machine_reset_phase_1*.
    1. Ball devices initialize their switches
    2. BCP sends the reset command to any attached media controllers
15. Reset posts the event *machine_reset_phase_2*.
    1. The ball controller updates its count of known balls
    2. Ball devices configure their eject targets
16. Reset posts the event *machine_reset_phase_3*.
    1. Ball locks are reset
    2. Drop targets are reset
    3. Drop target banks are reset
    4. GI is enabled
    5. Multiball devices are reset
    6. The attract mode starts as its a registered handler for *machine_reset_phase_3* .

Game Start Sequence

This sequence document starts with the attract mode running and ends with the running.

1. The player pushes a button tagged with "start". The time is noted.
2. The player releases that button. (This is important because in MPF it's possible to do different things based on a so-called "long press" of the start button. For example, you might start the machine in tournament mode, or allow players to select a player profile. So the game start process doesn't actually begin until the start button is released.)
3. The Attract mode posts the boolean event *request_to_start_game*. See the section below about the "How the *request_to_start_game* event works."
    1. The ball controller makes sure there are enough balls and that they are all gathered.
    2. Other modules make sure they are ready for the game to start and deny it if not.
4. The attract mode's `result_of_start_request` is the callback for the request event. If the result is True, this process continues.
5. The attract mode posts an event *game_start*.
6. The game mode is registered as a handler for the *game_start* event, so it starts.
7. The game mode posts a queue event called *game_starting*.
    1. The score reels reset themselves
    2. The auditor enables itself
    3. Info lights reset
8. The game mode's `game_start()` method is the callback for that queue event which is called when that event is finished.
9. The game mode calls its `_player_add()` method.

   1. The first player is created
   2. The number of players is updated

10. The game mode posts the event *game_started* .
11. The game mode calls its `player_turn_start()` method.

At this point we have a running game!

How the "request_to_start_game" event works

When a player pushes (and releases) the start button during attract mode, the Attract Mode code posts an MPF event called *request_to_start_game*.This event is not a normal event that is just posted and forgotten, rather, it's a special type of event called a "boolean event." When a system component posts a boolean event, it actually watches for responses from every other component that is watching for that event. If this event is posted and nothing speaks up to stop it, then the module that posted that event will continue. But if anything "kills" that event, that will cause whatever module that posted it to *not* proceed. This can be a bit confusing, so let's go through this in plain English:

1. When a player pushes and releases the start button, the attract mode says, "Hey! I'd like to start a game now. Does anyone have a problem with that?
2. This gives other components a chance to pipe up and say, "Yeah! I have a problem with that. You're not starting a game!"
3. If no one speaks up, the attract mode will say, "Ok, I'm posting a follow up event to kick off the game start process."
4. But if any component denies the start, then the attract mode will do nothing, and the game doesn't start.

So what types of components might register to watch for and/or interrupt the game start request? Lots of them.

The ball controller watches for this event and will make sure that the game has the minimum number of balls installed, and that those balls are all in their "home" positions. If everything is ok when the game start request comes in, then the ball controller will do nothing, allowing the start to proceed. But if the start request comes in an the ball controller doesn't have enough balls, it will "kill" the start request, and the game won't start. (When something kills an event like this, it's up to that component to make it obvious to the player what's going on. For example, the ball controller might put a message on the DMD which says something about balls being missing.)

Another component that might care about this game start request is the credits module. If the machine is *not* set to free play, then when the *request_to_start_game* event is posted, the credits module will make sure there's at least one credit on the machine. If not, then it will kill the event and not allow the game to start.

At this point you might be wondering what the point of all this is? Why have these start request events? Isn't this overly complicated? Why not just have MPF check all these things on its own?

The beauty of these types of events is that it makes it easy to customize and add features and components to MPF without the core MPF software knowing (or caring) what's installed and what might be starting an event. The MPF core doesn't know about credits or free play or any of that. It just says, "Hey, I want to start a game. Is that cool?" If you don't have a credits module, or if the credits module isn't active because the machine is on free play, then the credits module isn't there to deny the start and MPF can start the game no problem. But if then if you add or enable the credits module,then this start request process is what gives that random module a "hook" into the game starting process.

The real power of this comes with future flexibility. You might want to create some other type of component that we never thought of. (Maybe you don't want any new games to start after 11pm or something?) Thanks to this request event, you can write your own module as a simple snap- in which "hooks" this game start event, and MPF doesn't need to know about the details, and you don't have to resort to a "hack" of the MPF core to hook in whatever future crazy module you have. It's very cool!

Ball Start Sequence

This sequence shows everything that happens when a new ball starts in MPF. There are actually a few different ways we can end up here: If this the first ball of the first player in a new game:

1. After the game mode posts the *game_started* event, it will call its `player_turn_start()` method.
2. The `player_turn_start()` method does a few things:
   1. If there's not an active player (because this it the start of a new game), it called the game mode's `player_rotate()` method which maps the game's *player* attribute to the current player.
   2. Posts an event called *player_turn_started*.
   3. The game mode's `_player_turn_started()` method is a callback for that event, which is called next.
3. The `_player_turn_started()` method:
   1. Increments the ball count for the player
   2. Calls the game mode's `ball_starting()` method.
4. The `ball_starting()` method:
   1. Posts player, ball, and score information to the debug log
   2. Posts the *ball_starting* event. Like the *game_starting* event from the last step, this is also a queue event, meaning any component can hook in to do whatever it needs to do before releasing control. (This could be per-player animations and cut scenes, maybe the tilt wants to wait a few seconds for the plumb bob to stop rocking, etc.)
5. The game's `ball_started()` method is the callback for the ball_starting event.
   1. Event handlers for ball_drain are added.
   2. balls_in_play is set to 1.
   3. The *ball_started* event is posted.
6. Many things are configured to respond to the *ball_started* event, including:
   1. Shots are enabled
   2. Autofire devices are enabled
   3. Flippers are enabled
   4. Ball lock devices are enabled
   5. Multiball devices are enabled

7. The playfield's `add_ball()` method is called.
   1. The ball controller looks for a ball device in the `default_source_device` setting of your playfield, and it changes that device's desired ball count to 1. (In this example lets assume that you have a plunger lane and a trough.)
   2. The trough sees that one of its eject targets (the plunger lane) wants a ball, so it ejects one.
   3. The plunger lane receives and confirms that it now has a ball.
   4. If this machine has a launch button and a coil-fired plunger, the player hits a button tagged with `player_controlled_eject_tag`.
   5. The ball controller receives a request to add a live ball.
   6. The ball device in the `default_source_device` of your playfield ejects its ball.
   7. If the machine is configured with a player_controller_eject_tag, that tag is passed as the trigger event that will launch the ball.

The ball is now in play.

Mode Start Sequence

Here's what happens when a mode starts:

1. One of the events in the mode's `start_events:` is posted.
2. The mode's `start()` method responds since it's registered as a handler for those events.
   1. If the mode is currently active, this process ends.
   2. If a *callback* kwarg is included in the event, it's saved for later use.
   3. Any *kwargs* that were attached to the event which started the mode are saved for later use.
3. Any devices that are configured in this mode's config that are not already created are created now.
4. Any events listed in the mode's `stop_events:` setting are registered and will call the mode's `stop()` method if they're posted.
   1. These events are registered with the priority of the mode +1, so they are called first.
5. Any registered mode *start_methods* are called one-by-one. These are called with the mode, the mode's config, and the mode's priority as *kwargs*.
6. Any device control_events from the mode config are registered
7. A queue event is posted called *mode_<mode_name>_starting* .
8. The mode's `_started()` method is the callback for the starting queue event and is called when that event is complete.
9. Mode timers are started.
10. An event *mode_<mode_name>_started* is posted.
11. The mode's `_mode_started_callback()` method is the callback for the started event, so it's called once that event is complete.
12. The mode's `mode_start()` method is called. (This is the method that can be subclassed to run custom mode code.)

    1. Any *kwargs* that were passed along with the event that started the mode are passed to the `mode_start()` method.

13. If a start *callback* was passed with the event that started the mode, it's called now.

Mode Stop Sequence

Here's what happens behind-the-scenes when a mode stops.

1. An event listed in the mode's `stop_events:` setting is posted.
2. This is handled by the mode's `stop()` method.
    1. If the mode is not active, this process ends.
    2. If a *callback* argument was passed, it's saved now for later use
    3. Other *kwargs* are saved for later use
3. Switch handlers registered by that mode are removed.
4. Timers set in that mode are stopped and removed.
5. Delays set in that mode are cleared.
6. An queue event is posted: *mode_<mode_name>_stopping*.
7. Once that queue is clear, the mode's `_stopped()` method is called.
8. Any mode *stop_methods* registered for that mode are called one-by- one. (mode stop_methods are based on anything that gets returned from the call to the mode's start_methods when the mode starts).
9. An event *mode_<mode_name>_stopped* is posted.
10. Once any handlers for that event have finished, the mode's `_mode_stopped_callback()` method is called.
11. Mode event handlers are removed.
12. Devices that were created as part of this mode are removed.
13. The mode's `mode_stop()` method is called. (This is the method that can be subclassed in custom mode code for things you want to run when the mode stops.)
    1. If *kwargs* were passed as part of the event in Step 1, they're included in the call to `mode_stop()`.
14. If a *callback* was saved in Step 2, it's called now.

Ball End Sequence

This sequence starts with a ball live and in play and ends when the ball drains and the ball is over.

1. The ball enters a ball device device tagged with drain.
2. The ball controller's `_ball_drained_handler()` method responds to the ball having entered a device tagged with `drain`.
3. It posts a relay event called *ball_drain*, along with the number of balls that just drained.

   1. Various modules can hook event this to "remove" a ball from the *ball_drain* event so it doesn't count as a drain. (For example, ball save.)

4. The game mode's `ball_drained()` method is registered as a handler for the *ball_drain* event.
5. It subtracts the number of balls that just drained from its *balls_in_play* count.
6. If the balls_in_play count was a positive number and goes to zero, the game mode's ball_ending() method is called.
7. The game mode posts the queue event *ball_ending*.
8. Once that event is done, the game mode's `_ball_ending_done()` method is called.
9. The event *ball_ended* is posted.
10. The game mode's `ball_ended()` method is called.
    1. If the player has any extra balls, the game mode's `shoot_again()` method is called.
    2. If the player is the last player, and the ball is the last ball, the game mode's `game_ending()` method is called.
11. Otherwise the game mode's `player_rotate()` method is called.
12. The game mode's `player_turn_start()` method is called.

CHAPTER 20

Troubleshooting

Your machine is not starting, behaving weird or crashing? We are sorry to hear that. This chapter tries to help you to help yourself. Please try to find the root of your problem. Maybe the solution will be obvious then. If not we will help you in the forum.

Please remember that this is a two step process: First, try to diagnose the problem and collect as much information as possible. Second, report the issue if you cannot find a solution yourself. If you skip the first step we will likely send you to this page.

Step 1: Diagnosing Your Issue

Do you already know *how to turn on debugging and increase log verbosity*?

What kind of issue are you having?

Debugging Memory Leaks

Sometimes you might experience out of memory conditions. This might be due to bugs in MPF, custom code or certain config features. We found that most leaks are caused either by dangling event handlers or slide/widget which never get unloaded. For that reason, we added a feature to MPF and MPF-MC to dump all of those. To trigger the debug dump start MPF and MPF-MC without the production flag and post the `debug_dump_stats` events. For example, you can add a keyboard key `d` to do that:

```
keyboard:
  d:
    event: debug_dump_stats
```

The MPF log will contain something like this:

```
2018-12-10 21:35:55,682 : INFO : EventManager : Event: ======'debug_dump_stats'====== Args={'_from_bcp
↪': True}
2018-12-10 21:35:55,683 : INFO : EventManager : --- DEBUG DUMP EVENTS ---
2018-12-10 21:35:55,683 : INFO : EventManager : Total registered_handlers: 265. Total event_queue: 0.
↪Total callback_queue: 0. Total _queue_tasks: 0
2018-12-10 21:35:55,683 : INFO : EventManager : Registered Handlers:
2018-12-10 21:35:55,683 : INFO : EventManager :   Total handlers: 24 (for ball_starting)
[...]
2018-12-10 21:35:55,689 : INFO : EventManager :   Total handlers: 1 (for balldevice_bd_scoop_front_ball_
↪eject_failed)
2018-12-10 21:35:55,689 : INFO : EventManager : Queue events:
2018-12-10 21:35:55,689 : INFO : EventManager : --- DEBUG DUMP EVENTS END ---
```

MPF-MC will contain even more information:

```
2018-12-10 21:35:55,682 : EventManager : Event: ======'debug_dump_stats'====== Args={}
2018-12-10 21:35:55,702 : EventManager : --- DEBUG DUMP EVENTS ---
2018-12-10 21:35:55,703 : EventManager : Total registered_handlers: 42. Total event_queue: 0. Total
↪callback_queue: 0. Total _queue_tasks: 0
2018-12-10 21:35:55,703 : EventManager : Registered Handlers:
2018-12-10 21:35:55,703 : EventManager :   Total handlers: 2 (for service_power_off)
2018-12-10 21:35:55,703 : EventManager :   Total handlers: 2 (for debug_dump_stats)
2018-12-10 21:35:55,703 : EventManager :   Total handlers: 1 (for service_menu_show)
2018-12-10 21:35:55,703 : EventManager :   Total handlers: 1 (for assets_to_load)
2018-12-10 21:35:55,703 : EventManager :   Total handlers: 1 (for sound_loop_sets_clear)
2018-12-10 21:35:55,703 : EventManager :   Total handlers: 1 (for master_volume_decrease)
2018-12-10 21:35:55,703 : EventManager :   Total handlers: 1 (for service_menu_selected_switch)
[...]
2018-12-10 21:35:55,705 : EventManager :   Total handlers: 1 (for service_coil_test_start)
2018-12-10 21:35:55,705 : EventManager :   Total handlers: 1 (for service_door_opened)
2018-12-10 21:35:55,705 : EventManager : Queue events:
2018-12-10 21:35:55,705 : EventManager : --- DEBUG DUMP EVENTS END ---
2018-12-10 21:35:55,705 : mpfmc : --- DEBUG DUMP DISPLAYS ---
2018-12-10 21:35:55,705 : mpfmc : Active slides: {'playfield_blank': <Slide name=playfield_blank,
↪priority=0, id=1>, 'transparent_playfield': <Slide name=transparent_playfield, priority=0, id=5>,
↪'dmd_back_blank': <Slide name=dmd_back_blank, priority=0, id=2>, 'window_slide_1': <Slide name=window_
↪slide_1, priority=0, id=6>, 'dmd_front': <Slide name=dmd_front, priority=10, id=8>, 'dmd_front_blank
↪': <Slide name=dmd_front_blank, priority=0, id=4>, 'window_blank': <Slide name=window_blank,
↪priority=0, id=3>, 'dmd_back': <Slide name=dmd_back, priority=10, id=7>} (Count: 8). Displays: {'dmd_
↪front': <Display name=dmd_front[128, 32], current slide=dmd_front, total slides=2>, 'dmd_back':
↪<Display name=dmd_back[128, 32], current slide=dmd_back, total slides=2>, 'window': <Display
↪name=window[600, 700], current slide=window_slide_1, total slides=2>, 'playfield': <Display
↪name=playfield[225, 250], current slide=transparent_playfield, total slides=2>} (Count: 4)
2018-12-10 21:35:55,705 : mpfmc : Listing children for display: <Display name=dmd_front[128, 32],
↪current slide=dmd_front, total slides=2>
2018-12-10 21:35:55,705 : mpfmc : <Display name=dmd_front[128, 32], current slide=dmd_front, total
↪slides=2>
2018-12-10 21:35:55,705 : mpfmc : <Slide name=dmd_front, priority=10, id=8>
2018-12-10 21:35:55,706 : mpfmc : <WidgetContainer id=None z=0 key=None>
2018-12-10 21:35:55,706 : mpfmc : <Image name=drache, size=[128, 32], pos=[64, 16]>
2018-12-10 21:35:55,706 : mpfmc : Total children: 4
[...]
2018-12-10 21:35:55,708 : mpfmc : <Slide name=transparent_playfield, priority=0, id=5>
2018-12-10 21:35:55,708 : mpfmc : <WidgetContainer id=None z=2 key=None>
2018-12-10 21:35:55,708 : mpfmc : <Image name=nyannyan, size=[110, 281], pos=[172.0, 155.
↪70284725004058]>
```

(continues on next page)

(continued from previous page)

```
2018-12-10 21:35:55,708 : mpfmc : <WidgetContainer id=None z=2 key=None>
2018-12-10 21:35:55,708 : mpfmc : <Image name=nyannyan, size=[110, 281], pos=[60.0, 100.44406174999192]>
2018-12-10 21:35:55,708 : mpfmc : Total children: 6
2018-12-10 21:35:55,708 : mpfmc : --- DEBUG DUMP DISPLAYS END ---
2018-12-10 21:35:55,732 : mpfmc : --- DEBUG DUMP OBJECTS ---
2018-12-10 21:35:55,732 : mpfmc : Elements in list (may be dead): 152
2018-12-10 21:35:55,732 : mpfmc : <Display name=playfield[225, 250], current slide=transparent_
↪playfield, total slides=2>
2018-12-10 21:35:55,732 : mpfmc : <Display name=dmd_back[128, 32], current slide=dmd_back, total␣
↪slides=2>
2018-12-10 21:35:55,732 : mpfmc : <Display name=window[600, 700], current slide=window_slide_1, total␣
↪slides=2>
2018-12-10 21:35:55,732 : mpfmc : <Display name=dmd_front[128, 32], current slide=dmd_front, total␣
↪slides=2>
2018-12-10 21:35:55,732 : mpfmc : <DisplayWidget size=[600, 700], pos=[0, 0], source=window>
2018-12-10 21:35:55,733 : mpfmc : <Slide name=playfield_blank, priority=0, id=1>
[...]
2018-12-10 21:35:55,737 : mpfmc : <Image name=drache, size=[128, 32], pos=[64, 16]>
2018-12-10 21:35:55,737 : mpfmc : <Image name=nyannyan, size=[110, 281], pos=[60.0, 100.44406174999192]>
2018-12-10 21:35:55,737 : mpfmc : <Image name=nyannyan, size=[110, 281], pos=[172.0, 155.
↪70284725004058]>
2018-12-10 21:35:55,737 : mpfmc : --- DEBUG DUMP OBJECTS END ---
2018-12-10 21:35:55,737 : mpfmc : --- DEBUG DUMP CLOCK ---
2018-12-10 21:35:55,737 : mpfmc : <ClockEvent (1.0) callback=<function Cache._purge_by_timeout at␣
↪0x7fe7c73eeae8>>
2018-12-10 21:35:55,737 : mpfmc : <ClockEvent (0.0) callback=<bound method SoundSystem.tick of <mpfmc.
↪core.audio.SoundSystem object at 0x7fe7b89c0080>>>
2018-12-10 21:35:55,737 : mpfmc : <ClockEvent (0.0) callback=<bound method BcpProcessor._get_from_queue␣
↪of <mpfmc.core.bcp_processor.BcpProcessor object at 0x7fe7b89457f0>>>
2018-12-10 21:35:55,737 : mpfmc : <ClockEvent (1.0) callback=<bound method MpfMc._check_crash_queue of
↪<mpfmc.core.mc.MpfMc object at 0x7fe7c8ab91e8>>>
2018-12-10 21:35:55,737 : mpfmc : <ClockEvent (0.0) callback=<bound method MpfMc.tick of <mpfmc.core.mc.
↪MpfMc object at 0x7fe7c8ab91e8>>>
2018-12-10 21:35:55,737 : mpfmc : <ClockEvent (0.2) callback=<bound method WindowSDL._check_keyboard_
↪shown of <kivy.core.window.window_sdl2.WindowSDL object at 0x7fe7b833a180>>>
2018-12-10 21:35:55,738 : mpfmc : <ClockEvent (0.0) callback=<bound method EffectWidget._update_glsl of
↪<kivy.uix.effectwidget.EffectWidget object at 0x7fe7b79c1e80>>>
2018-12-10 21:35:55,738 : mpfmc : <ClockEvent (0.0) callback=<bound method EffectWidget._update_glsl of
↪<kivy.uix.effectwidget.EffectWidget object at 0x7fe7b79f0180>>>
2018-12-10 21:35:55,738 : mpfmc : <ClockEvent (0.0) callback=<bound method EffectWidget._update_glsl of
↪<kivy.uix.effectwidget.EffectWidget object at 0x7fe79edc9c78>>>
2018-12-10 21:35:55,738 : mpfmc : <ClockEvent (0.0) callback=<bound method DmdBase.tick of <mpfmc.core.
↪dmd.RgbDmd object at 0x7fe7b89ee198>>>
2018-12-10 21:35:55,738 : mpfmc : <ClockEvent (0.0) callback=<bound method EffectWidget._update_glsl of
↪<kivy.uix.effectwidget.EffectWidget object at 0x7fe79edc9f50>>>
2018-12-10 21:35:55,738 : mpfmc : <ClockEvent (0.0) callback=<bound method DmdBase.tick of <mpfmc.core.
↪dmd.RgbDmd object at 0x7fe7b88b36a0>>>
2018-12-10 21:35:55,738 : mpfmc : <ClockEvent (0.1) callback=<bound method Image._anim of <kivy.core.
↪image.Image object at 0x7fe7b7c0a800>>>
2018-12-10 21:35:55,738 : mpfmc : <ClockEvent (0.0) callback=<bound method McDisplayLightPlayer._tick␣
↪of BcpConfigPlayer.display_lights>>
2018-12-10 21:35:55,738 : mpfmc : <ClockEvent (9.0) callback=<bound method Widget.remove of <Image␣
↪name=nyannyan, size=[110, 281], pos=[60.0, 100.44406174999192]>>>
2018-12-10 21:35:55,738 : mpfmc : <ClockEvent (0.0) callback=<bound method Animation._update of <mpfmc.
↪uix.relative_animation.RelativeAnimation object at 0x7fe79d1f18d0>>>
```

(continues on next page)

(continued from previous page)

```
2018-12-10 21:35:55,738 : mpfmc : <ClockEvent (9.0) callback=<bound method Widget.remove of <Image␣
↪name=nyannyan, size=[110, 281], pos=[172.0, 155.70284725004058]>>>
2018-12-10 21:35:55,739 : mpfmc : <ClockEvent (0.0) callback=<bound method Animation._update of <mpfmc.
↪uix.relative_animation.RelativeAnimation object at 0x7fe79d1f1e80>>>
2018-12-10 21:35:55,739 : mpfmc : --- DEBUG DUMP CLOCK END ---
```

Leaks usually occur over time so dump all objects on start of your machine. Leave it running for a few minutes and dump all objects again. Then compare the output of those two. Look for events with a very high number of handlers (or a number which is constantly increasing). Check for widgets or slides which are existing more than once. If you got questions ask in the forum.

Debugging YAML Parse Errors

In case something goes wrong and you get errors like this:

```
ValueError: YAML error found in file config/config.yaml. Line 22,Position 10: mapping values are not␣
↪allowed here
  in "config/config.yaml", line 22, column 10
```

This means that the error might be at line 22, just before it or shortly after it. Sometimes it is tricky to tell whats wrong when one space is off. A good editor might help but it might be still hard to spot the exact point.

Install an IDE

We recommend the *MPF language server* with a supported IDE for that.

Install the extension

If you are struggling to find the problem you can reformat your file using `ruamel.yaml`. To do that you first need to install the `ruamel.yaml.cmd` extension:

```
pip3 install ruamel.yaml.cmd==0.2
```

Make a backup

Before you continue: Make a backup of your machine config. Seriously, do it! Even better, use git and commit right now!

Reformatting YAML files

After that you can reformat single files using the `round-trip` command. For example if you want to reformat `your_file.yaml` first check the changes it would make:

```
yaml round-trip your_file.yaml
```

If that looks alright perform them by adding the `--save` flag:

```
yaml round-trip --save your_file.yaml
```

This will keep comments but reformat all your indents to two spaces per level. It should be easier now to spot the problem.

Reformat Your Config Using MPF format

Run `mpf format` on your config. See *Format And Lint Config Files* for details.

What if it did not help?

If this did not help you can ask in the mpf-users Google group. Please post the full error message, your log file and the relevant config file.

Debugging Segfaults

If you experience a crash/segfault or hang (especially in MC) you can run gdb on python to find the crash or hang. You can attach a debugger to the running mc process like this:

```
$ ps aux | grep mpf
jan       9678 12.4  0.3 1082068 127304 pts/2  SNl+ 23:17   0:06 /usr/bin/python3 /usr/local/bin/mpf mc
jan       9760 37.0  0.1 571368 56660 pts/3    Sl+  23:17   0:01 /usr/bin/python3 /usr/local/bin/mpf␣
↪game -X
```

In this example `9678` is the pid of MC and `9760` is the pid of MPF. You can then attach gdb:

```
$ sudo gdb python3 9678
[...]
(gdb) thread apply all bt
[...]
(gdb) thread apply all py-bt
[...]
```

Please send us the complete output of gdb. That will help us to figure out the problem.

Debugging MPF installation problems

If you suspect a problem with MPF itself you can try to run the demo_man game. Make sure that you select the same version as your MPF version (i.e. demo_man 0.33.x for MPF 0.33.10).

Additionally, you can run the MPF and MPF-MC unit tests (the number of tests may be different).

```
$ python3 -m unittest discover -s mpf.tests
[...]
----------------------------------------------------------------------
Ran 622 tests in 20.818s

OK
```

Similarly, you can run MPF-MC unit tests (they will take a bit longer and might show some deprecation warnings from kivy):

```
$ python3 -m unittest discover -s mpfmc.tests
[...]
Ran 182 tests in 193.610s

OK
```

If you coils are not firing, switches are not working or hardware is behaving weirdly in general read our *hardware troubleshooting guide*.

Step 2: Prepare a Report and Ask in the Forum

Please include the following information if available and relevant:

Output of MPF diagnosis

If your game won't run, let's make sure MPF is ok. This will also tell use which MPF and MPF-MC version you are using. Run `mpf diagnosis` from within your machine folder to see if your installation is fine:

```
$ mpf diagnosis



MPF version: MPF v0.50.0-dev.11
MPF install location: /data/home/jan/cloud/flipper/src/mpf/mpf
Machine folder detected: /data/home/jan/cloud/flipper/src/good_vs_evil
MPF-MC version: MPF-MC v0.50.0-dev.5 (config_version=5, BCP v1.1, Requires MPF v0.50.0-dev.10)

Serial ports found:
/dev/ttyUSB3
    desc: Quad RS232-HS
    hwid: USB VID:PID=0403:6011 LOCATION=1-12
/dev/ttyUSB2
    desc: Quad RS232-HS
    hwid: USB VID:PID=0403:6011 LOCATION=1-12
/dev/ttyUSB1
    desc: Quad RS232-HS
    hwid: USB VID:PID=0403:6011 LOCATION=1-12
/dev/ttyUSB0
    desc: Quad RS232-HS
    hwid: USB VID:PID=0403:6011 LOCATION=1-12
```

Relevant Configuration

Please provide the relevant configuration snippets. Leave out anything which is not related. For instance if you got problems with lights on your P-Roc or FAST platform provide the configuration for the relevant lights, the `p_roc` or `fast` section and any light_players or shows which are used when the problem occurs.

Attach a Log with debug and verbose logging

Please attach the log with verbose logging from MPF or MPF-MC (depending where your problem occured). Make sure you enabled `debug` on the relevant devices and/or platforms. See *how to turn on debugging and increase log verbosity* for details.

A link to your machine config also help. Ideally this would be some git repository which can be checked out and browsed online.

Prepare the Error Message

Your error message likely is inside your log. However, please include it inside your message as well. See *Reading MPF Errors* for how to read the error.

Please check the relevant device or platform documentation for any mentions of that error. Often we already documented how to solve it.

Tell Us How to Reproduce Your Problem

It might be hard for us to help you if we cannot reproduce your issue. Is there a way you can provide a minimal config which shows your problem? Try to remove everything unrelated to your problem and bring it to its bare minimum. Sometimes you will find the root of the issue while doing this. You would be surprised how often issues are caused by seemingly unrelated devices or configs.

Ideally you can provide a *single file test* which fails or shows your issue in its log. This allows us to verify the issue quickly and provide a quick fix. But don't worry if this not possible. Just a minimal machine config is also fine. In that case please tell us how to run your machine to experience the issue.

Ask In the Forum

With all this information ask in our support forum. Please keep in mind that MPF is an open source project and we are doing this for fun in our spare time. Be kind and patient. If you provide more relevant information it is likely that somebody can help you. More is not always better if it is not relevant to your problem. But missing information will just delay the overall process.

1. If you got a problem with a device (e.g. a ball_lock) or a platform (e.g. P-ROC or FAST) add `debug: True` to the relevant config section to enable extra debug output.
2. Add a log of your game. Therefore, run your game with `mpf both -v -V` and grab the latest MPF and MC log from the log folder in your machine.
3. Describe how to reproduce your problem.
4. Provide relevant config snippets or, if possible, a link to download/checkout your machine config so we can reproduce the issue.

Consider Improving the Documentation

Did you solve your issue but found that some relevant information in the documentation is missing or should be linked/located elsewhere? Either tell us in the forum or consider *improving the documentation* yourself to save future users some troubles the same way others saved you some troubles by writing this documentation.

More Howtos

How to Turn On Debug and How to Increase Log Verbosity?

You got some kind of issue in MPF? A crash, weird behaviour or it won't start? Then this guide is for you. You will learn how to turn up logging and how to selectively enable debugging.

1. Run MPF without text ui

The text ui which is shown by default may hide some errors and make troubleshooting more difficult. To disable text ui run mpf using:

```
$ mpf both -t
```

This will just show the log on the console. If some crashes occur this might reveal them as the text ui sometimes hides them.

2. Start MPF and MPF-MC separately

If MPF and MPF-MC logs mix up too much you can start them separately:

```
$ mpf game -t
```

And in another console:

```
$ mpf mc
```

This might help you to find out where a crash or error is originating from.

3. Increase Log Verbosity

If you experience problems you should increase verbosity:

```
$ mpf game -t -v -V
```

Start MPF-MC in a separate console:

```
$ mpf mc -v -V
```

(This will also work with `mpf both -t -v -V`).

Scroll up in the console to find the error which was emitted.

This will increase the size of your logs and slow down MPF a bit. It should not be used in production but it should be fine to always use otherwise especially during development.

4. Checkout the Log Folder

MPF will generate separate logs for MPF and MPF-MC in the `logs` folder in your machine. Those will also contain a bit more information than the console. Find the issue in your log. Keep this ready for later in case you want to report an issue.

5. Enable Debugging

If you are having an issue with a specific device or platform you should try to enable debugging. Almost all devices and platforms in MPF support a `debug` option. If in doubt check the *config reference*. For instance if you suspect an issue with a switch add `debug: true` to it's config:

```
switches:
  my_switch:
    number: 42
    debug: true
```

Same works with all devices. It will generate more log lines but should not affect performance much.

Most platforms support the same. For instance with a P-Roc:

```
p_roc:
  debug: true
```

For most platforms this will generate a lot of log lines and might also affect performance a lot. We recommend to disable it after you finished debugging. See *Troubleshooting Hardware Platforms* for details.

After enable debug check the log again to understand what your device or platform is actually doing at the time of your issue.

Reading MPF Errors

MPF errors might be chained. This means that a more general error is caused by a more specific one. In general, you need to read those error from the bottom to the top. On the bottom there will be the most general error and all errors above will be more specific.

For instance a Switch might be unable to initialize because your hardware platform cannot connect to the relevant node board:

```
INFO : EventManager : Event: ======'shutdown'====== Args={}
Shutdown because of an exception:
ERROR : Machine : Runtime Exception
Traceback (most recent call last):
  File "/mpf/mpf/devices/switch.py", line 135, in _initialize
    self.config['number'], config, self.config['platform_settings'])
  File "/mpf/mpf/platforms/virtual.py", line 94, in configure_switch
    raise AssertionError("Cannot find board for switch {}".format(number))
AssertionError: Cannot find board for switch 0-7
```

(continues on next page)

(continued from previous page)

```
The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "/mpf/mpf/core/machine.py", line 741, in _run_loop
    raise self._exception['exception']
  File "uvloop/cbhandles.pyx", line 70, in uvloop.loop.Handle._run
  File "/mpf/mpf/core/events.py", line 114, in _async_handler_done
    future.result()
  File "/mpf/mpf/core/device_manager.py", line 103, in _load_device_modules
    await self.initialize_devices()
  File "/mpf/mpf/core/device_manager.py", line 199, in initialize_devices
    await collection[device_name].device_added_system_wide()
  File "/mpf/mpf/core/system_wide_device.py", line 15, in device_added_system_wide
    await self._initialize()
  File "/mpf/mpf/devices/switch.py", line 137, in _initialize
    raise AssertionError("Failed to configure switch {} in platform. See error above".format(self.
↪name)) from e
AssertionError: Failed to configure switch s_door_back in platform. See error above
```

So in this case the door switch could not be configured because the node board was missing at the hardware.

Attaching A Debugger to MPF

Video about how to attach a debugger using pycharm:

https://youtu.be/LPSfUHKIpYk

CHAPTER 21

How To Build Physical Pinball Machines

This chapter is about building physical pinball machines. If you plan to build a homebrew machine read this section and you might be able to skip some mistakes which have been made by others before. We try to cover most of the building part here. A few things are elsewhere, notably *Pinball Controll Systems* and *Pinball Mechs*.

Where should you start? If you want to create a custom layout read our *guide on layout considerations*.

What Should You Consider When Planning a Playfield Layout?

In general, it is a good idea if you really played pinball before. You need to get a feel for what is rewarding as a player and then you need to understand how that works. For instance, where are the switches to play the first sound when you hit the entrance of a shot? When and how does the machine count the shot as successful?

Otherwise, you might end up with switches that when hit do absolutely nothing. Not even a blinking light, score increment or a basic sound effect because it does not make sense at that point. Anyone who had played even a little pinball would've known what people were expecting. As you observe and play more pinball, you get a knack for what types of things will really immerse the player, and can then incorporate them into your own games.

Some notes from people in our community (*please tell* us if you have more):

Pop Bumpers

Pop bumpers are great for randomizing the action of the ball. They can also offer the player a bit of pause to consider where the ball is coming from. When designing your pop bumper area, it's generally good advice to have rubber surrounding the area to keep the ball action high.

Surrounding pop bumpers with steel (like ball guides, for instance) is a quick way to kill the action (and the fun).

Below is a picture of pop bumpers surrounded by steel, with large gaps to the lower playfield. Action here will be very limited:

Below is a picture of pop bumpers surrounded by rubber, with a defined exit to the lower playfield. Action here will be very high:

Upper Flippers

Upper flippers are a great way to add a second dimension to the shot flow of a game. For example, if you have an upper right flipper, you can incorporate shots on the middle/upper left of the playfield behind other objects. These are generally harder shots on their own. With that said, you should at least have a feed to the upper flipper (from a ramp, an orbit, or some other shot). Generally its a good idea to make the access to the flipper easier, because the shots off of the flipper will be more difficult.

Inserts

Make sure each shot has dedicated inserts so that you can indicate what you want the player to shoot. RGB arrow inserts are great for denoting shots to the player (you can color code them) and you don't have to put text over the arrow. Lower inserts will generally be "mode specific". You can look at the ramp/orbit shots on Demolition Man to get an idea for the insert layout. The arrows are generic, everything else is specific.


JACKPOT
EXPLODE
EXPLODE
CAR CHASE
CLAW READY
FREEZE

Ball Guides and Posts

The ball should never ever ever hit metal directly unless its a ball guide. Even then, the end of the ball guide should have a rubber post (positioned such that the edge of the guide is covered, but a ball rolling down the guide wont hit it and have its trajectory altered). (I knew this and still managed to screw it up on Wizard Blocks as several shots immediately hit the lane guides or metal ramps. It causes the whole game to play like a big clunky metal piece of garbage.)

Also, when positioning ball guides for an orbit shot, it is generally a pleasing experience to the player to have the ball come off the orbit right toward the flipper. Some designs have placed orbits such that they hit the tip of the slingshot and bounce the ball out of control. This makes for a more difficult game, and if orbit shots are key to certain modes, this could be rather frustrating for a player who's trying to control the ball.

Shot Lines

It goes without saying that any shot you place on a playfield should be makeable from one or multiple flippers. Fan layouts are a common occurence in playfield design (think of No Fear or Monster Bash), typically arranged into a fan of 7 or 8 shots.

If you find yourself doing more asymmetric playfield designs, you should pay special attention to make sure that the shots are makeable. You can draw shot lines from the flippers to measure this.

Every shot from the flipper generally leaves in a straight line. Take your playfield CAD/Drawing/etc and see if you can draw a straight line up the playfield to the shot you're testing. Be sure to account for half the diameter of the ball to make sure a collision with another object won't throw the ball off course.

Shot lines are also a great tool to see which shots are able to be back handed off the flipper on the same side.

Here's an example of shot lines on Demolition Man:

Anything missing?

Do you have more advice? Did you make a stupid mistake (in retrospective) and want to tell other about it? Please *contribute a section to this guide* or tell us in the MPF Users Google Group.

Planning Layout with CAD

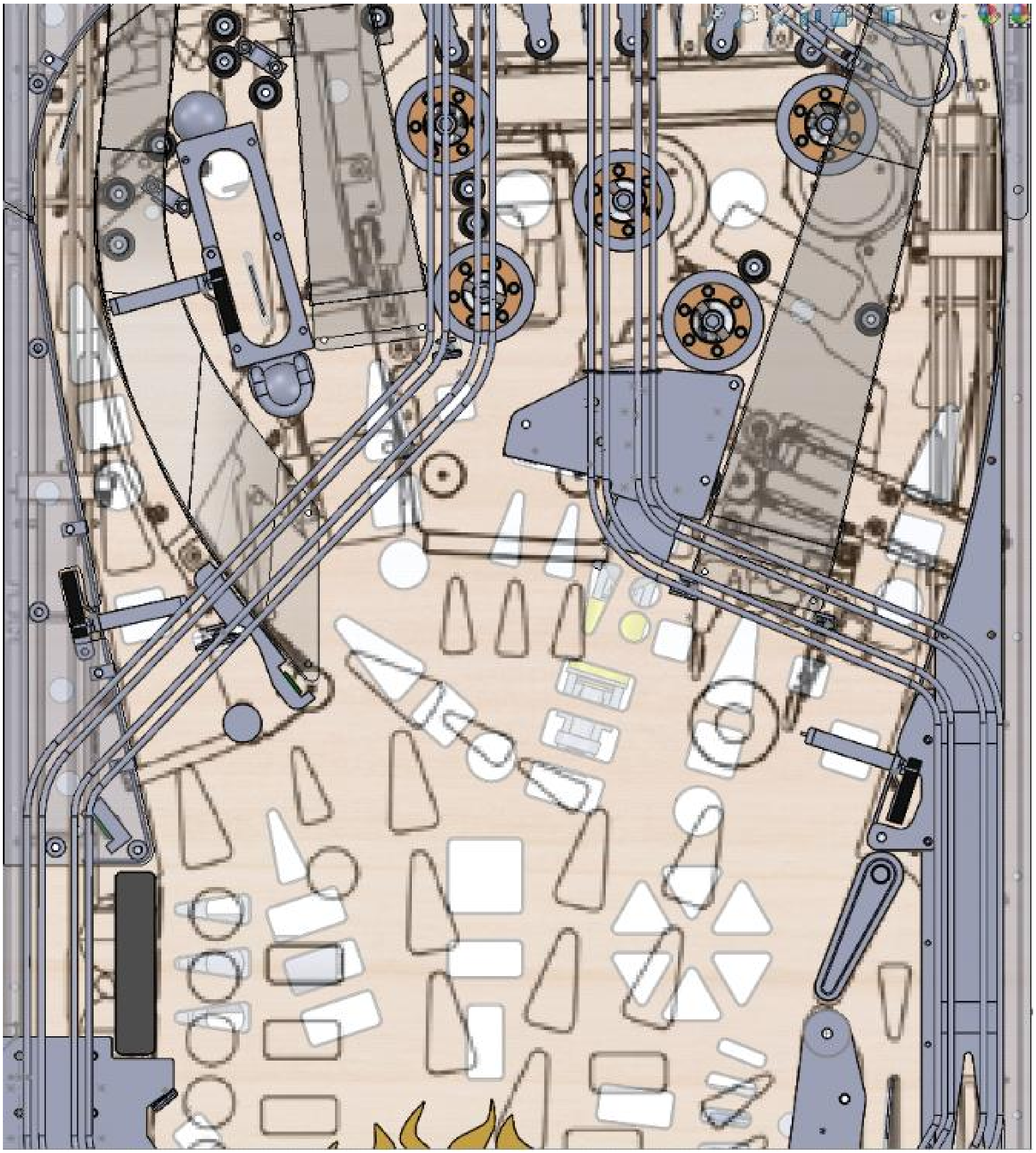
Borrowing Shots With CAD

If you have planned shot in an area that matches an existing machine, you can borrow/take their geometry. This way you know the geometry will be good without any revisions. PDF's of instructions manuals for games are a good source for the flat overhead pic you will need. IPDB.org is one good source for these.

Most CAD programs have a function to overlay an image file directly onto your model. Search youtube for "How to overlay image" + your CAD system.

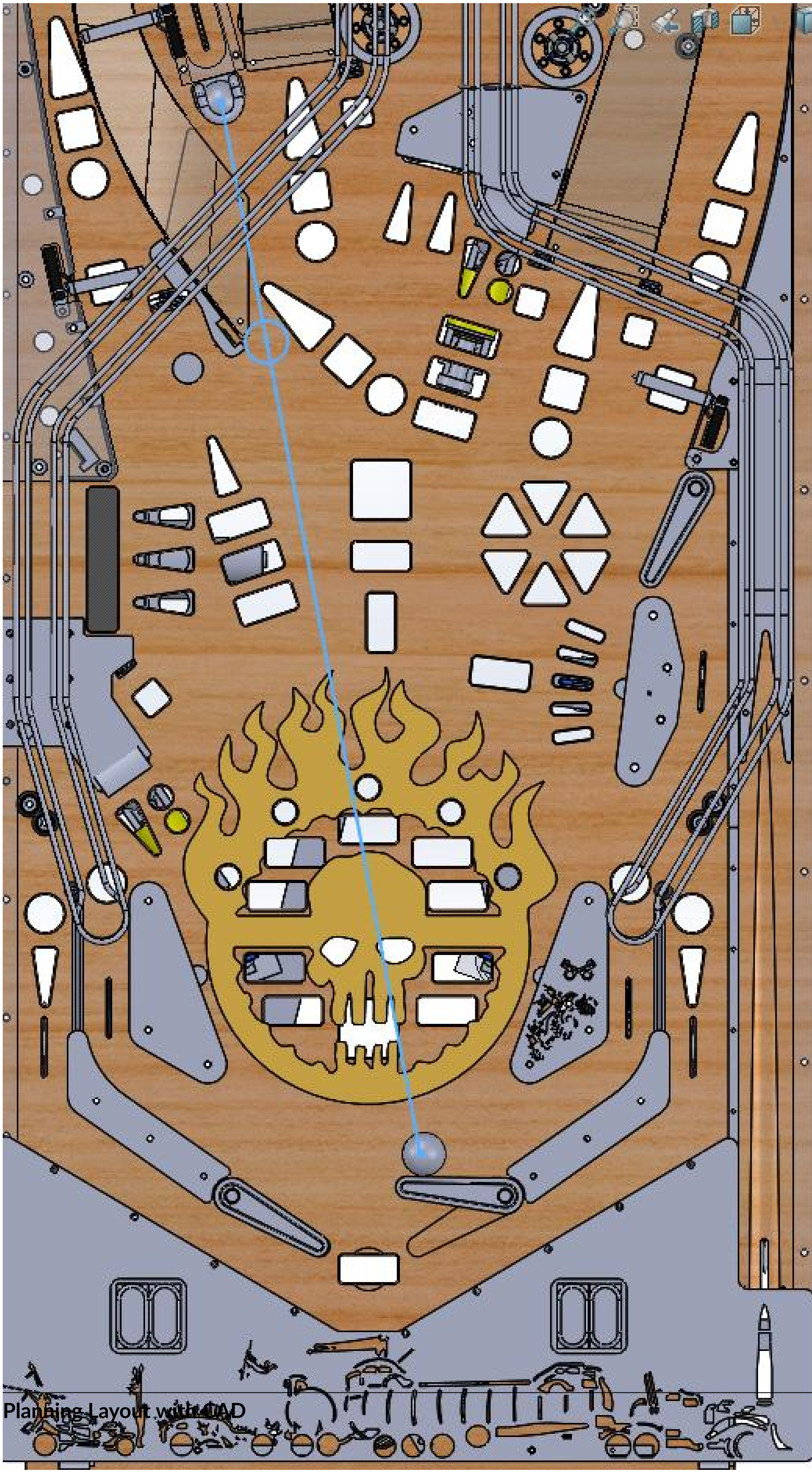
You will have to move and skew the image until the flippers and size line up with your drawing. Once this is done you can take a shot with confidence.

Here's an example of a Spiderman pinball overlay onto a homebrew pinball machine to get the geometry of the Venom ramp shot:

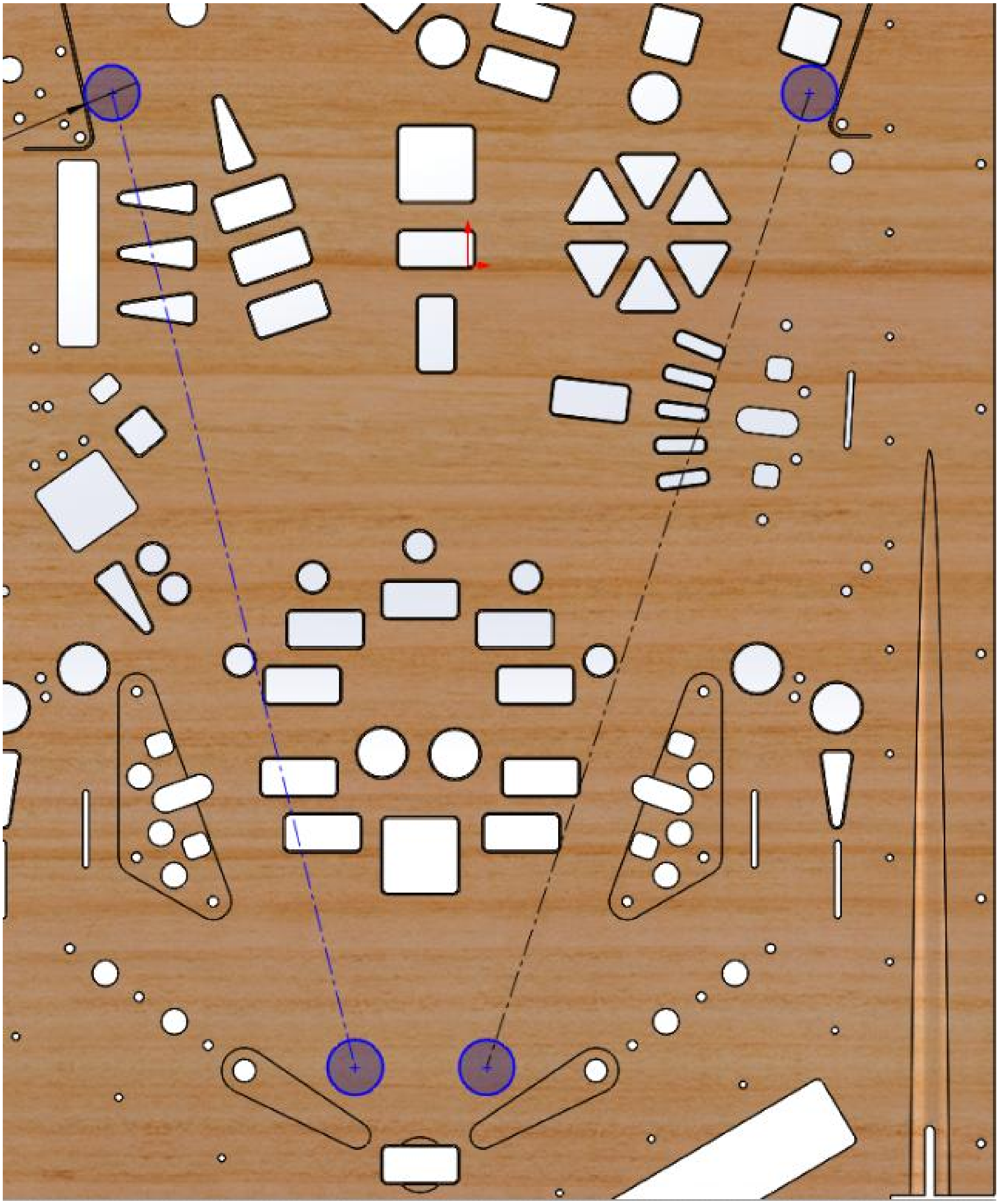


Using to CAD to Test/Plan Shots

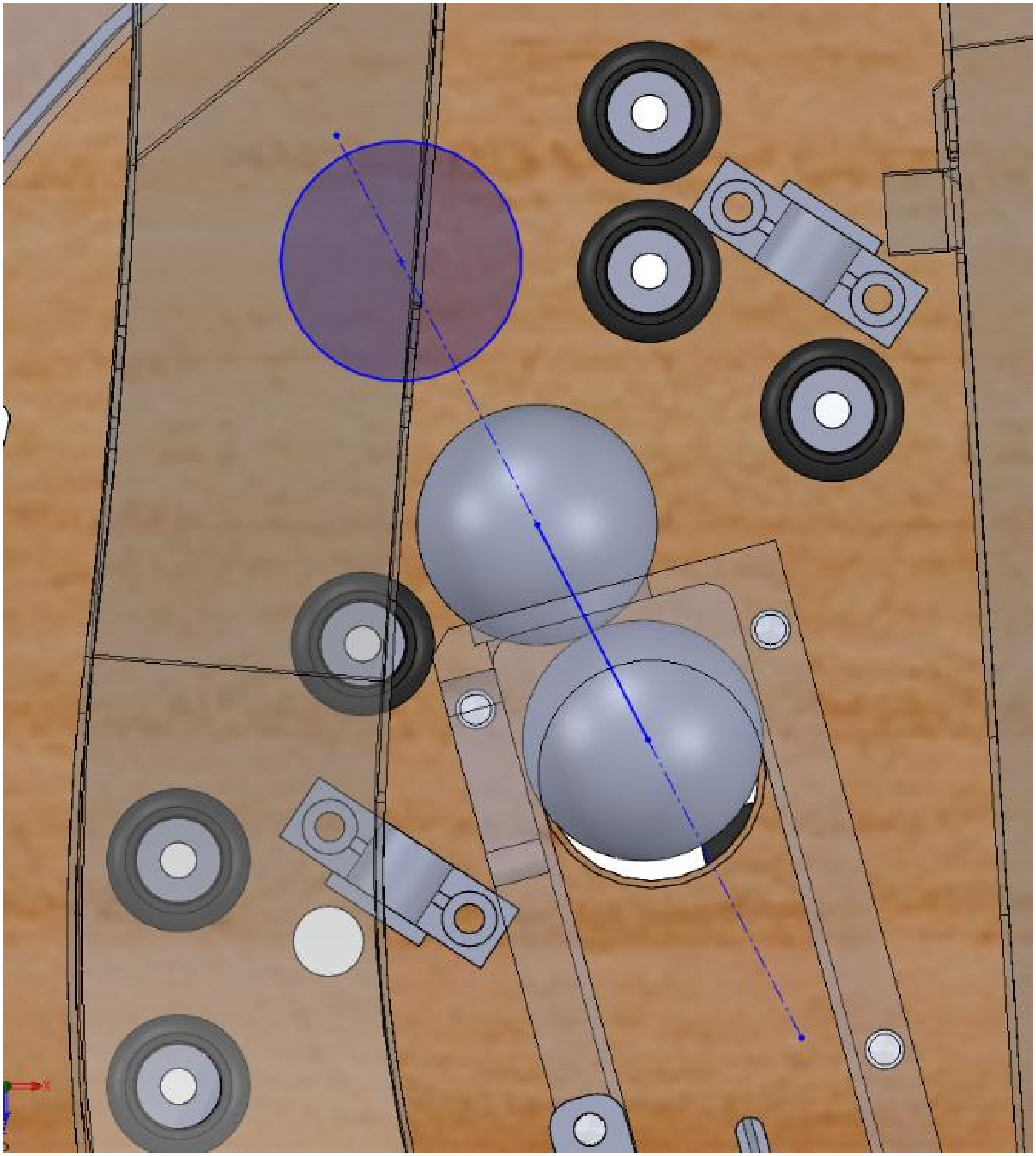
You can draw a shot in CAD to see if it is makeable. Here is an example of testing if a newton ball shot can be made from the right flipper:



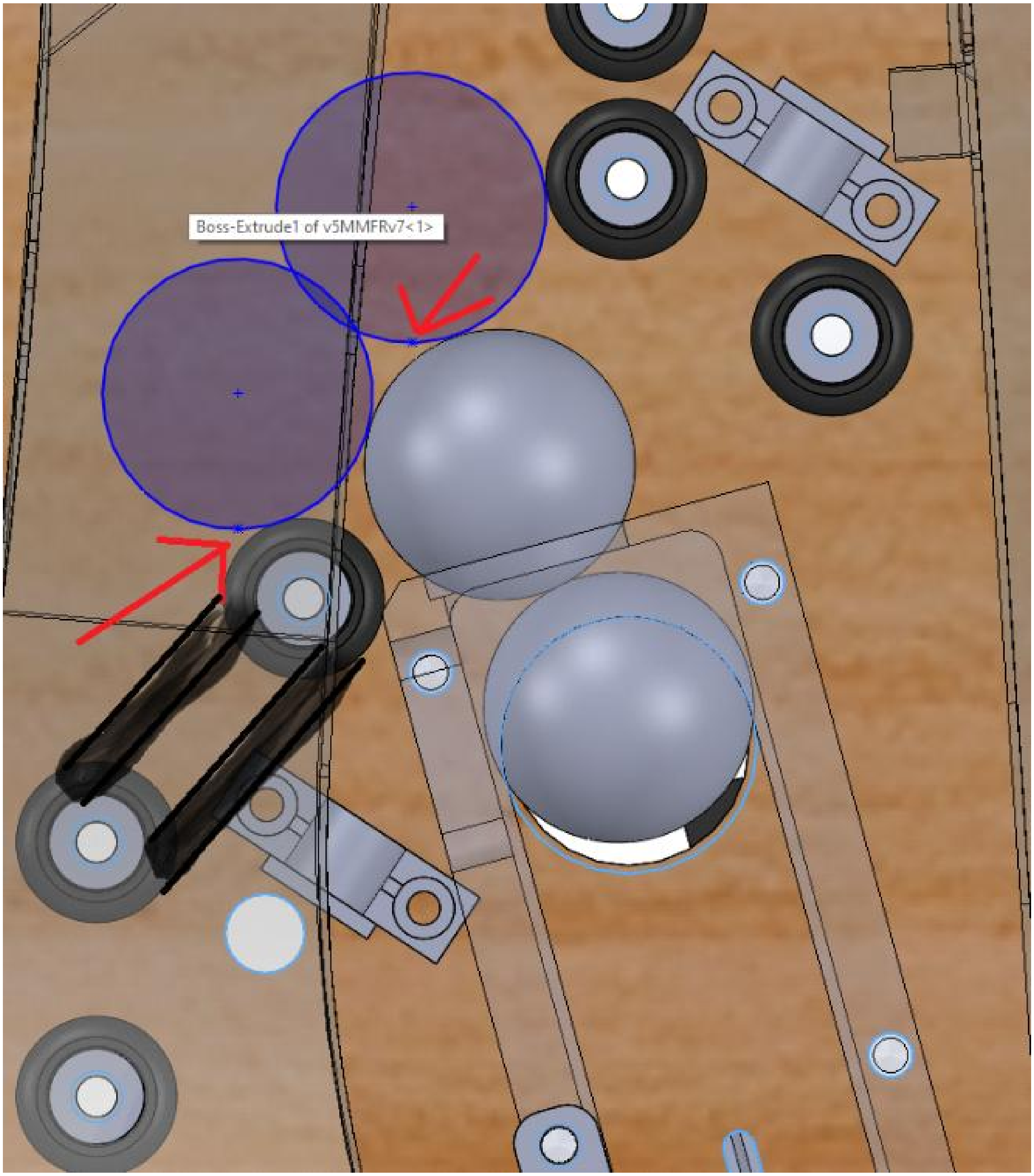
Here we test where the balls will go coming of the orbit shots. (We left a small straight line at the end/beginning of the orbit when designing, then made the dotted lines parallel):



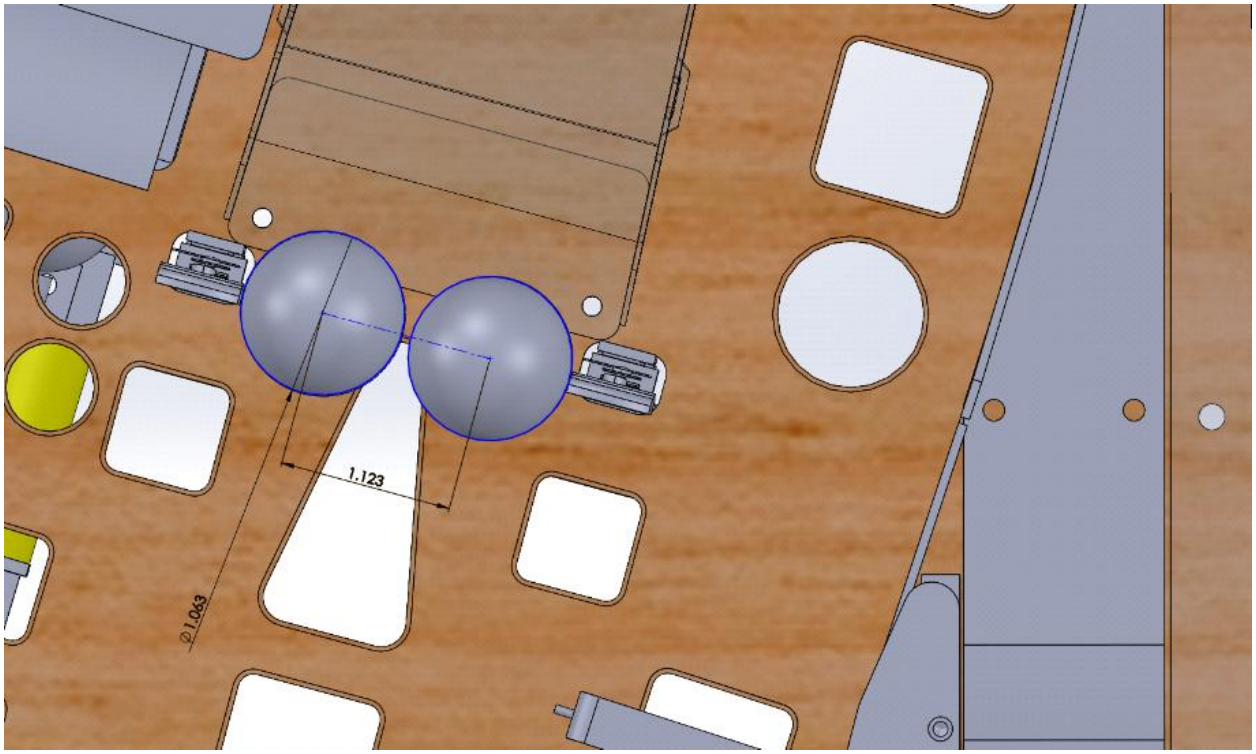
Here we use CAD to see how a ball lock will exit when hit with a trapped newton ball:



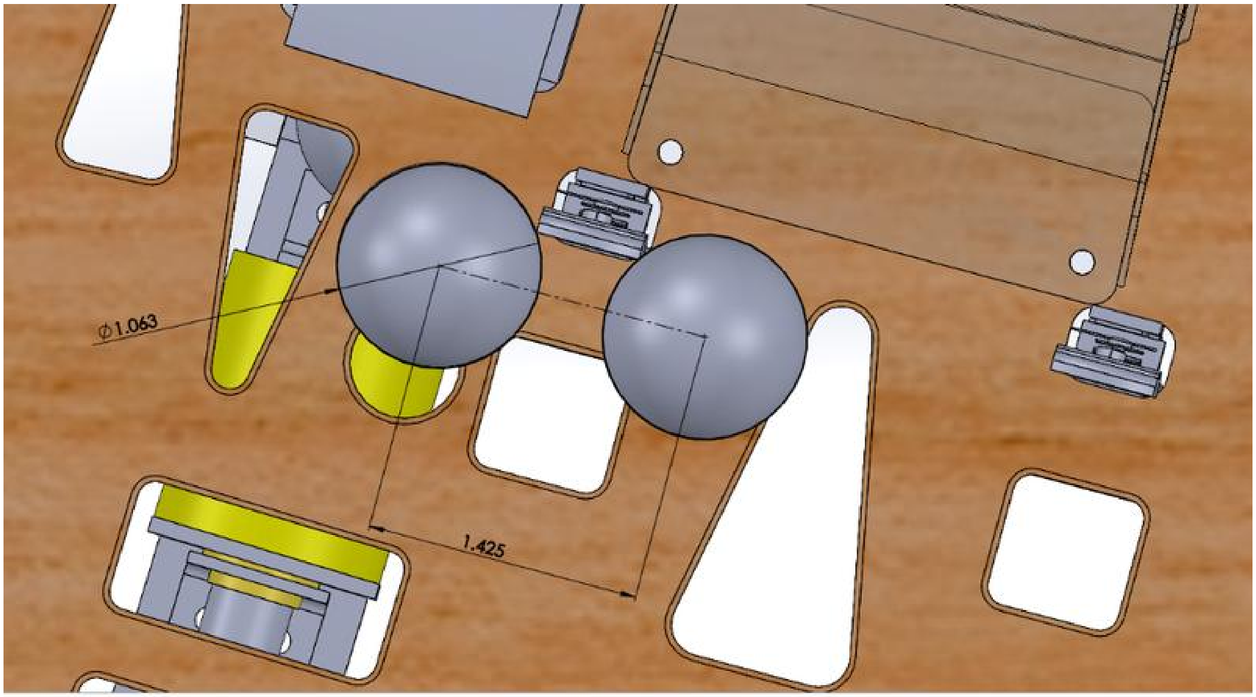
Here we use CAD to see if balls wil get trapped after the ball lock:



Subtract the ball diameter (1.0625”) from ramp and lane shots to see their actual width.



Here you can see a .500” wide mini target is easier to hit than an over 2” wide ramp.



CHAPTER 22

Example Configuration Files

MPF is very complex with lots of modules and options. In order to make sure that everything works, we have over 700 automated tests that run every time we add or change something in MPF in order to make sure we didn't break something.

All of these automated tests include config files (machine configs, mode configs, and show files). In many ways, these config files are the "ultimate truth" when it comes to what configs actually work with MPF.

All of the links below show the actual config files (pulled from the MPF and MPF-MC packages) that are used to test MPF. They're also a valuable resource for people creating games with MPF since they show many different options and configurations that are known to work.

You can click on any of the links below to see the actual config files for each topic. Each link may have multiple separate machine configs, mode configs, and/or show configs.

accelerometer (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 1: your_machine_folder/accelerometer/config/config.yaml

```
#config_version=5

accelerometers:
    test_accelerometer:
        number:
        level_x: 0
```

(continues on next page)

(continued from previous page)

```
        level_y: 0
        level_z: 1
        hit_limits:
            0.5: event_hit1
            1.5: event_hit2
        level_limits:
            2: event_level1
            5: event_level2
```

achievement (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 2: your_machine_folder/achievement/config/config.yaml

```
#config_version=5

switches:
    test:
        number:

lights:
  led1:
    number:
  led2:
    number:
  led4:
    number:
  led5:
    number:
  led6:
    number:

modes:
  - base
  - mode1
  - auto_select

shows:
  achievement1_enabled:
    - time: 1
  achievement1_started:
    - time: 1
  achievement1_completed:
    - time: 1
  achievement1_disabled:
    - time: 1
  achievement1_stopped:
```

(continues on next page)

(continued from previous page)

```
    - time: 1

  achievement2_disabled:
    - time: 1
      lights:
        (led): off
  achievement2_enabled:
    - time: 1
      lights:
        (led): yellow
  achievement2_started:
    - time: 1
      lights:
        (led): green
  achievement2_stopped:
    - time: 1
      lights:
        (led): red
  achievement2_completed:
    - time: 1
      lights:
        (led): blue
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 3: your_machine_folder/achievement/modes/base/config/base.yaml

```
#config_version=5
mode:
    start_events: ball_starting
    priority: 100

achievements:
  achievement1:
    start_events: achievement1_start
    stop_events: achievement1_stop
    enable_events: achievement1_enable
    disable_events: achievement1_disable
    complete_events: achievement1_complete
    reset_events: achievement1_reset
    show_when_disabled: achievement1_disabled
    show_when_enabled: achievement1_enabled
    show_when_started: achievement1_started
    show_when_stopped: achievement1_stopped
    show_when_completed: achievement1_completed
    restart_on_next_ball_when_started: True

  achievement2:
```

(continues on next page)

(continued from previous page)

```
    start_events: achievement2_start
    stop_events: achievement2_stop
    enable_events: achievement2_enable
    disable_events: achievement2_disable
    complete_events: achievement2_complete
    reset_events: achievement2_reset
    events_when_started: test_event, test_event2
    show_when_enabled: achievement2_enabled
    show_when_started: achievement2_started
    show_when_completed: achievement2_completed
    restart_after_stop_possible: False
    enable_on_next_ball_when_enabled: False
    show_tokens:
      led: led1

  achievement3:
    start_events: achievement3_start
    stop_events: achievement3_stop
    enable_events: achievement3_enable
    disable_events: achievement3_disable
    complete_events: achievement3_complete
    reset_events: achievement3_reset
    events_when_started: test_event, test_event3
    show_when_disabled: achievement_disabled
    show_when_enabled: achievement_enabled
    show_when_started: achievement_started
    show_when_stopped: achievement_stopped
    show_when_completed: achievement_completed
    restart_after_stop_possible: False

  achievement4:
    start_events: achievement4_start
    stop_events: achievement4_stop
    enable_events: achievement4_enable
    disable_events: achievement4_disable
    complete_events: achievement4_complete
    reset_events: achievement4_reset
    show_when_disabled: achievement_disabled
    show_when_enabled: achievement_enabled
    show_when_started: achievement_started
    show_when_stopped: achievement_stopped
    show_when_completed: achievement_completed
    show_when_selected: achievement_selected
    show_tokens:
          led: led4

  achievement5:
    start_events: achievement5_start
    stop_events: achievement5_stop
    enable_events: achievement5_enable
    disable_events: achievement5_disable
    complete_events: achievement5_complete
    reset_events: achievement5_reset
    events_when_started: test_event, test_event5
```

(continues on next page)

(continued from previous page)

```
    show_when_disabled: achievement_disabled
    show_when_enabled: achievement_enabled
    show_when_started: achievement_started
    show_when_stopped: achievement_stopped
    show_when_completed: achievement_completed
    show_when_selected: achievement_selected
    show_tokens:
          led: led5

  achievement6:
    start_events: achievement6_start
    stop_events: achievement6_stop
    enable_events: achievement6_enable
    disable_events: achievement6_disable
    complete_events: achievement6_complete
    reset_events: achievement6_reset
    events_when_started: test_event, test_event6
    show_when_disabled: achievement_disabled
    show_when_enabled: achievement_enabled
    show_when_started: achievement_started
    show_when_stopped: achievement_stopped
    show_when_completed: achievement_completed
    show_when_selected: achievement_selected
    show_tokens:
          led: led6

  achievement7: {}

  achievement8: {}

  achievement9: {}

  achievement10:
    debug: True

  achievement11:
    debug: True

  achievement12:
    enable_events: enable_achievements

  achievement13:
    enable_events: enable_achievements

  achievement14: {}

  achievement15: {}

  achievement16: {}

  achievement17: {}

  mode1_a1:
    enable_events: enable_all
```

(continues on next page)

(continued from previous page)

```
    start_events: start_all
    stop_events: stop_all
    complete_events: complete_all

  mode1_a2:
    enable_events: enable_all
    start_events: start_all
    stop_events: stop_all
    complete_events: complete_all

achievement_groups:
    group1:
        achievements: achievement7, achievement8, achievement9
        auto_select: true

    group2:
        achievements: achievement4, achievement5, achievement6
        enable_events: group2_enable
        disable_events: group2_disable
        start_selected_events: group2_start
        select_random_achievement_events: group2_random
        rotate_right_events: group2_rotate_right
        rotate_left_events: group2_rotate_left

        disable_while_achievement_started: False
        enable_while_no_achievement_started: False

        events_when_all_completed: group2_complete
        events_when_no_more_enabled: group2_no_more
        events_when_enabled: group2_enabled

        show_when_enabled: group2_show
        show_tokens:
          led: led2

    group3:
        achievements:
          - achievement10
          - achievement11
          - achievement12
          - achievement13
        auto_select: yes
        debug: True

    group4:
        debug: True
        achievements: achievement14, achievement15, achievement16
        enable_events: group4_enable
        disable_events: group4_disable
        start_selected_events: group4_start
        select_random_achievement_events: group4_random
        rotate_right_events: group4_rotate_right
        rotate_left_events: group4_rotate_left
        events_when_all_completed: group4_complete
```

(continues on next page)

(continued from previous page)

```yaml
        events_when_no_more_enabled: group4_no_more
        events_when_enabled: group4_enabled
        allow_selection_change_while_disabled: True
        auto_select: True

shows:
  group2_show:
    - duration: .1
      lights:
        (led): red
    - duration: .1
      lights:
        (led): blue
  achievement_enabled:
    - duration: 1
      lights:
        (led): yellow
  achievement_disabled:
    - duration: 1
      lights:
        (led): off
  achievement_completed:
    - duration: 1
      lights:
        (led): blue
  achievement_started:
    - duration: 1
      lights:
        (led): green
  achievement_stopped:
    - duration: 1
      lights:
        (led): red
  achievement_selected:
    - duration: 1
      lights:
        (led): orange
```

Listing 4: your_machine_folder/achievement/modes/mode1/config/mode1.yaml

```yaml
#config_version=5
mode:
    start_events: start_mode1
    stop_events: stop_mode1
    priority: 100

achievement_groups:
    mode1_ag1:
        achievements: mode1_a1, mode1_a2
        events_when_no_more_enabled: enable_all
        auto_select: yes
        debug: True
```

Listing 5: your_machine_folder/achievement/modes/auto_select/config/auto_select.yaml

```
#config_version=5
mode:
    start_events: start_mode2
    stop_events: stop_mode2
    priority: 100

achievements:
  spinTasticAward:
    complete_events: mode_spinTasticAward_stopped
    reset_events: reset_bonusAwards
    debug: True

  tagTeamAward:
    complete_events: mode_tagTeamAward_stopped
    reset_events: reset_bonusAwards
    debug: True

  doubleChanceAward:
    complete_events: mode_doubleChanceAward_stopped
    reset_events: reset_bonusAwards
    debug: True

  extraBallAward:
    complete_events: extraBallAwardIntro_complete
    reset_events: reset_bonusAwards
    debug: True

  prodigiousPopsAward:
    complete_events: mode_prodigiousPopsAward_stopped
    reset_events: reset_bonusAwards
    debug: True

achievement_groups:
  bonus_awards:
    achievements:
      - doubleChanceAward
      - extraBallAward
      - prodigiousPopsAward
      - tagTeamAward
      - spinTasticAward
    auto_select: true
    enable_while_no_achievement_started: false
    rotate_right_events: sw_pops, s_spotTarget_active
    allow_selection_change_while_disabled: true
    disable_while_achievement_started: false
    start_selected_events: start_event
    disable_events: disable_bonus
    enable_events: enable_group
    debug: True
```

animated_images (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 6: your_machine_folder/animated_images/config/test_animated_images.yaml

```
#config_version=5

displays:
  default:
    width: 400
    height: 300

images:
  stick-figures-skipframes:
    file: reel.gif
    frame_skips:
      - from: 3
        to: 8

slides:
  slide1:
    - type: image
      image: ball
      y: 250
      fps: 30
    - image: busy-stick-figures-animated
      type: image
      y: 100
      x: 250
    - type: text
      text: ZIP FILE OF PNGs
      y: 260
    - type: text
      text: ANIMATED GIF
      x: 10
      y: 100
      anchor_x: left
    - type: text
      text: (ALSO TESTING STOPPING
      x: 10
      y: 80
      font_size: 10
      anchor_x: left
    - type: text
      text: SKIPPING, & STARTING)
      font_size: 10
      x: 14
      y: 68
      anchor_x: left
  slide2:
```

(continues on next page)

(continued from previous page)

```
    - image: busy-stick-figures-animated
      type: image
      y: 100
      x: 250
  slide3:
    - image: busy-stick-figures-animated
      type: image
      auto_play: false
      start_frame: 4
  slide4:
    - image: stick-figures-skipframes
      type: image
      auto_play: false
      animations:
          advance_frames:
            - property: end_frame
              value: 10
              duration: 0
slide_player:
  slide1: slide1
  slide1_remove:
    slide1: remove
  slide2:
    slide2:
      priority: 200
  slide3: slide3
  slide4: slide4
```

animation (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 7: your_machine_folder/animation/config/test_animation.yaml

```
#config_version=5
displays:
  default:
    width: 400
    height: 300

slides:
  slide1:
    type: text
    text: text
    x: 0
    animations:
      show_slide:
        - property: x  # x, y, height, width, opacity, rotation?
```

(continues on next page)

(continued from previous page)

```
        value: 101
        duration: 1s
        repeat: False
      - property: x  # x, y, height, width, opacity, rotation?
        value: 100
        duration: 1s
        timing: with_previous  # or after prev
        repeat: True
  reset_animations_events: pre_show_slide

slide2:
  type: text
  text: ANIMATION TEST
  color: ff00ff
  font_size: 100
  x: 400
  y: 300
  animations:
    entrance2:
      property: x, y
      value: 0, 0
      duration: 1s
      timing: with_previous  # or after prev

slide3:
  type: text
  text: text3
  color: green
  opacity: 0
  animations:
    entrance3: fade_in, multi
    fade_in: fade_in
    advance_x: advance_x_50
    advance_y: advance_y_50
    advance_xy: advance_xy_50

slide4:
  type: text
  text: text4
  animations:
    entrance4: fade_in, multi
    some_event4: multi

slide5:
  type: text
  text: text5
  animations:
    entrance5: fade_in, multi
    event5:
      property: x # x, y, height, width, opacity, rotation?
      value: 98
      duration: 1s
      timing: with_previous  # or after prev
      repeat: True
```

(continues on next page)

(continued from previous page)

```
slide6:
  type: text
  text: text6

slide7:
  type: text
  text: TEST ANIMATION ON show_slide
  x: 100
  color: ffaa00
  font_size: 50
  animations:
    show_slide:
      property: x
      value: 500
      duration: 500ms

slide8:
  type: text
  text: TEST ANIMATION FROM OFF SCREEN
  y: 75%

base_slide:
  background_color: blue
  widgets:
    type: text
    text: WIDGET ANIMATION TESTS

slide9:
  type: text
  text: ANIMATION pre_show_slide
  x: 100
  color: ffaa00
  font_size: 50
  animations:
    pre_show_slide:
      property: x
      value: 500
      duration: 500ms

slide10:
  type: text
  text: ANIMATION show_slide
  x: 100
  color: ffaa00
  font_size: 50
  animations:
    show_slide:
      property: x
      value: 500
      duration: 500ms

slide11:
  type: text
```

(continues on next page)

(continued from previous page)

```
    text: ANIMATION pre_slide_leave
    color: ffaa00
    font_size: 50
    animations:
      pre_slide_leave:
        property: x
        value: -400
        duration: 500ms

  slide12:
    type: text
    text: ANIMATION slide_leave
    color: ffaa00
    font_size: 50
    animations:
      slide_leave:
        property: x
        value: 0
        duration: 500ms

  slide13:
    type: text
    text: RESET POSITION pre_show_slide
    x: 100
    animations:
      show_slide:
        - property: x
          value: 200
          duration: 1s
    reset_animations_events: pre_show_slide

  slide14:
    type: text
    text: RESET POSITION slide_play
    x: 100
    animations:
      show_slide:
        - property: x
          value: 200
          duration: 1s
    reset_animations_events: slide_play

  slide15:
    type: text
    text: RESET POSITION standard event
    x: 100
    animations:
      show_slide:
        - property: x
          value: 200
          duration: 1s
    reset_animations_events: event1

slide_player:
```

(continues on next page)

(continued from previous page)

```
  show_slide1: slide1
  show_slide7: slide7
  show_slide2: slide2
  show_slide3: slide3
  show_slide8: slide8
  show_slide9:
    slide9:
      transition:
        type: fade
        duration: 1s
  show_slide10:
    slide10:
      transition:
        type: fade
        duration: 1s
  show_slide11: slide11
  show_slide12: slide12
  show_base_slide: base_slide
  show_base_slide_with_transition:
    base_slide:
      transition:
        type: fade
        duration: 1s
  show_slide13: slide13
  show_slide14: slide14
  show_slide15: slide15

widgets:
  widget1:
     type: text
     text: WIDGET 1
     color: red
     x: -100
     animations:
       move_on_slide:
        - property: x
          value: 100
          duration: 500ms
          timing: after_previous
       move_off_slide:
        - property: x
          value: -100
          duration: 500ms
          timing: after_previous
     expire: 2s

  widget2:
    type: text
    text: widget2
    color: red
    opacity: 0
    animations:
      animate_widget2: fade_in, multi
      pulse_widget2: pulse, pulse, pulse, pulse
```

(continues on next page)

(continued from previous page)

```
widget_player:
  show_widget1: widget1
  show_widget2: widget2

animations:
  fade_in:
    property: opacity
    value: 1
    duration: 1s
    timing: with_previous
    repeat: True

  multi:
  - property: y
    value: 0
    duration: 1s
  - property: x
    value: 0%
    duration: 1s
    timing: with_previous
    repeat: False

  pulse:
  - property: opacity
    value: 0
    duration: 100ms
  - property: opacity
    value: 1
    duration: 100ms
    timing: after_previous

  advance_x_50:
    property: x
    value: 50
    relative: True
    duration: 1s

  advance_y_50:
    property: y
    value: 50
    relative: True
    duration: 1s

  advance_xy_50:
    property: x, y
    value: 50, 50
    relative: True
    duration: 1s
```

apc (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 8: your_machine_folder/apc/config/config.yaml

```
#config_version=5

hardware:
    platform: lisy

lisy:
    connection: serial
    port: com1
    baud: 115200

switches:
    s_test00:
        number: 00
    s_flipper:
        number: 1
    s_flipper_eos:
        number: 2
    s_slingshot:
        number: 3
    s_test37:
        number: 37
    s_test77_nc:
        number: 77
        type: 'NC'

coils:
    c_test:
        number: 0
    c_test_allow_enable:
        number: 1
        default_hold_power: 1.0
    c_trough_eject:
        number: 103
        default_pulse_ms: 3s
    c_flipper_main:
        number: 5
        default_pulse_ms: 30
    c_flipper_hold:
        number: 6
        allow_enable: True
    c_slingshot:
        number: 7

digital_outputs:
    game_over_relay:
        number: 1
        type: light
        enable_events: ball_started
```

(continues on next page)

(continued from previous page)

```
        disable_events: ball_will_end

flippers:
    f_test_hold_eos:
        debug: true
        main_coil: c_flipper_main
        hold_coil: c_flipper_hold
        activation_switch: s_flipper
        eos_switch: s_flipper_eos
        use_eos: true

autofire_coils:
    ac_slingshot:
        coil: c_slingshot
        switch: s_slingshot

lights:
  test_light:
    number: 3

segment_displays:
  info_display:
    number: 0
    size: 16
  player1_display:
    number: 1
    size: 5
  player2_display:
    number: 2
    size: 7
  player3_display:
    number: 3
    size: 3
  player4_display:
    number: 4
    size: 16

hardware_sound_systems:
    default:
        label: APC

hardware_sound_player:
    test2:
        2:
            action: play
    test4:
        5:
            track: 2
            action: play
    play_file:
        "some_file": play_file
    play_file_loop:
        "some_file":
          action: play_file
```

(continues on next page)

(continued from previous page)

```
        platform_options:
          loop: True
          no_cache: False
  play_text:
      text:
        action: text_to_speech
        value: "Hello MPF"
        platform_options:
          loop: False
          no_cache: True
  volume_05:
      set_volume:
        action: set_volume
        value: 0.5
  increase_volume:
      0.1: increase_volume
  decrease_volume:
      decrease_volume:
        action: decrease_volume
        value: 0.01
  test3:
      3: play
  test_stop: stop
```

asset_manager (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 9: your_machine_folder/asset_manager/config/test_asset_loading.yaml

```
#config_version=5

mpf:
  default_light_hw_update_hz: 1

lights:
    led_01:
        number: 0
    led_02:
        number: 1
    light_01:
        number: 0
        subtype: matrix
        label: Test 0
    light_02:
        number: 1
        subtype: matrix
        label: Test 1
```

(continues on next page)

(continued from previous page)

```
    gi_01:
        subtype: gi
        number: 0
    flasher_01:
        platform: coils
        number: flasher_01

coils:
    coil_01:
        number: 1
        default_pulse_ms: 30
    flasher_01:
        number: 2
        label: Test flasher
        default_pulse_ms: 40

modes:
  - mode1

show_pools:
  group1:
    load: preload
    shows:
      - show1
      - show2
      - show3
    type: random
  group2:
    load: preload
    shows:
      - show1
      - show2
      - show3|2
    type: random
  group3:
    shows:
      - show1
      - show2
      - show3
    type: sequence
  group4:
    shows:
      - show1|4
      - show2|2
      - show3
    type: sequence
  group5:
    shows:
      - show1|1
      - show2|5
      - show3|1
    type: random_force_next
  group6:
    shows:
```

(continues on next page)

(continued from previous page)

```
      - show1
      - show2
      - show3
    type: random_force_all
  group7:
    shows:
      - show1
      - show2{mode.mode1.active}
      - show3{mode.mode1.stopping}
    type: random
  group8:
    shows:
      - show1{mode.mode1.active}
      - show2
      - show3{mode.mode1.stopping}
    type: sequence
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 10: your_machine_folder/asset_manager/modes/mode1/shows/show6.yaml

```
#show_version=5
- time: 0
  lights:
    led_01: 006400
    led_02: CCCCCC
    light_01: CC
    light_02: 78
    gi_01: FF
- time: 1
  lights:
    led_01: DarkGreen
    led_02: Black
- time: 2
  lights:
    led_01: DarkSlateGray
    led_02: Tomato
    light_01: FF
    light_02: 33
    gi_01: 99
- time: +1
  lights:
    led_01: MidnightBlue-f500 ms
    led_02: DarkOrange-f0.5 s
    gi_01: 33
- time: 4
  lights:
    led_01: Off-f800
```

(continues on next page)

(continued from previous page)

```
    led_02: Off-f800
    light_01: 00-f800
    light_02: 00-f800
    gi_01: 00
- time: 6
```

Listing 11: your_machine_folder/asset_manager/modes/mode1/shows/on_demand/show10.yaml

```
#show_version=5
- time: 0
  lights:
    led_01: 006400
    led_02: CCCCCC
    light_01: CC
    light_02: 78
    gi_01: FF
- time: 1
  lights:
    led_01: DarkGreen
    led_02: Black
- time: 2
  lights:
    led_01: DarkSlateGray
    led_02: Tomato
    light_01: FF
    light_02: 33
    gi_01: 99
- time: +1
  lights:
    led_01: MidnightBlue-f500 ms
    led_02: DarkOrange-f0.5 s
    gi_01: 33
- time: 4
  lights:
    led_01: Off-f800
    led_02: Off-f800
    light_01: 00-f800
    light_02: 00-f800
    gi_01: 00
- time: 6
```

Listing 12: your_machine_folder/asset_manager/modes/mode1/shows/preload/show7.yaml

```
#show_version=5
- time: 0
  lights:
    led_01: 006400
    led_02: CCCCCC
    light_01: CC
    light_02: 78
    gi_01: FF
- time: 1
  lights:
```

(continues on next page)

(continued from previous page)

```
    led_01: DarkGreen
    led_02: Black
- time: 2
  lights:
    led_01: DarkSlateGray
    led_02: Tomato
    light_01: FF
    light_02: 33
    gi_01: 99
- time: +1
  lights:
    led_01: MidnightBlue-f500 ms
    led_02: DarkOrange-f0.5 s
    gi_01: 33
- time: 4
  lights:
    led_01: Off-f800
    led_02: Off-f800
    light_01: 00-f800
    light_02: 00-f800
    gi_01: 00
- time: 6
```

Listing 13: your_machine_folder/asset_manager/modes/mode1/shows/custom1/show8.yaml

```
#show_version=5
- time: 0
  lights:
    led_01: 006400
    led_02: CCCCCC
    light_01: CC
    light_02: 78
    gi_01: FF
- time: 1
  lights:
    led_01: DarkGreen
    led_02: Black
- time: 2
  lights:
    led_01: DarkSlateGray
    led_02: Tomato
    light_01: FF
    light_02: 33
    gi_01: 99
- time: +1
  lights:
    led_01: MidnightBlue-f500 ms
    led_02: DarkOrange-f0.5 s
    gi_01: 33
- time: 4
  lights:
    led_01: Off-f800
    led_02: Off-f800
    light_01: 00-f800
```

(continues on next page)

(continued from previous page)

```
    light_02: 00-f800
    gi_01: 00
- time: 6
```

Listing 14: your_machine_folder/asset_manager/modes/mode1/shows/mode_start/show9.yaml

```
#show_version=5
- time: 0
  lights:
    led_01: 006400
    led_02: CCCCCC
    light_01: CC
    light_02: 78
    gi_01: FF
- time: 1
  lights:
    led_01: DarkGreen
    led_02: Black
- time: 2
  lights:
    led_01: DarkSlateGray
    led_02: Tomato
    light_01: FF
    light_02: 33
    gi_01: 99
- time: +1
  lights:
    led_01: MidnightBlue-f500 ms
    led_02: DarkOrange-f0.5 s
    gi_01: 33
- time: 4
  lights:
    led_01: Off-f800
    led_02: Off-f800
    light_01: 00-f800
    light_02: 00-f800
    gi_01: 00
- time: 6
```

Listing 15: your_machine_folder/asset_manager/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  priority: 300
  game_mode: False
```

Show file examples

Here are some example show files that go along with the above config(s).

Note that there are multiple shows here.

Listing 16: your_machine_folder/asset_manager/shows/show1.yaml

```
#show_version=5
- time: 0
  lights:
    led_01: 006400
    led_02: CCCCCC
    light_01: CC
    light_02: 78
    gi_01: FF
- time: 1
  lights:
    led_01: DarkGreen
    led_02: Black
- time: 2
  lights:
    led_01: DarkSlateGray
    led_02: Tomato
    light_01: FF
    light_02: 33
    gi_01: 99
- time: +1
  lights:
    led_01: MidnightBlue-f500 ms
    led_02: DarkOrange-f0.5 s
    gi_01: 33
- time: 4
  lights:
    led_01: Off-f800
    led_02: Off-f800
    light_01: 00-f800
    light_02: 00-f800
    gi_01: 00
- time: 6
```

Listing 17: your_machine_folder/asset_manager/shows/show12.yaml

```
#show_version=5
- time: 0
  lights:
    led_01: 006400
    led_02: CCCCCC
    light_01: CC
    light_02: 78
    gi_01: FF
- time: 1
  lights:
    led_01: DarkGreen
    led_02: Black
- time: 2
  lights:
    led_01: DarkSlateGray
    led_02: Tomato
    light_01: FF
```

(continues on next page)

(continued from previous page)

```
    light_02: 33
    gi_01: 99
- time: +1
  lights:
    led_01: MidnightBlue-f500 ms
    led_02: DarkOrange-f0.5 s
    gi_01: 33
- time: 4
  lights:
    led_01: Off-f800
    led_02: Off-f800
    light_01: 00-f800
    light_02: 00-f800
    gi_01: 00
- time: 6
```

Listing 18: your_machine_folder/asset_manager/shows/show2.yaml

```
#show_version=5
- time: 0
  lights:
    led_01: 006400
    led_02: CCCCCC
    light_01: CC
    light_02: 78
    gi_01: FF
- time: 1
  lights:
    led_01: DarkGreen
    led_02: Black
- time: 2
  lights:
    led_01: DarkSlateGray
    led_02: Tomato
    light_01: FF
    light_02: 33
    gi_01: 99
- time: +1
  lights:
    led_01: MidnightBlue-f500 ms
    led_02: DarkOrange-f0.5 s
    gi_01: 33
- time: 4
  lights:
    led_01: Off-f800
    led_02: Off-f800
    light_01: 00-f800
    light_02: 00-f800
    gi_01: 00
- time: 6
```

Listing 19: your_machine_folder/asset_manager/shows/show3.yaml

```
#show_version=5
- time: 0
  lights:
    led_01: 006400
    led_02: CCCCCC
    light_01: CC
    light_02: 78
    gi_01: FF
- time: 1
  lights:
    led_01: DarkGreen
    led_02: Black
- time: 2
  lights:
    led_01: DarkSlateGray
    led_02: Tomato
    light_01: FF
    light_02: 33
    gi_01: 99
- time: +1
  lights:
    led_01: MidnightBlue-f500 ms
    led_02: DarkOrange-f0.5 s
    gi_01: 33
- time: 4
  lights:
    led_01: Off-f800
    led_02: Off-f800
    light_01: 00-f800
    light_02: 00-f800
    gi_01: 00
- time: 6
```

Listing 20: your_machine_folder/asset_manager/shows/on_demand/show5.yaml

```
#show_version=5
- time: 0
  lights:
    led_01: 006400
    led_02: CCCCCC
    light_01: CC
    light_02: 78
    gi_01: FF
- time: 1
  lights:
    led_01: DarkGreen
    led_02: Black
- time: 2
  lights:
    led_01: DarkSlateGray
    led_02: Tomato
    light_01: FF
```

(continues on next page)

(continued from previous page)

```
    light_02: 33
    gi_01: 99
- time: +1
  lights:
    led_01: MidnightBlue-f500 ms
    led_02: DarkOrange-f0.5 s
    gi_01: 33
- time: 4
  lights:
    led_01: Off-f800
    led_02: Off-f800
    light_01: 00-f800
    light_02: 00-f800
    gi_01: 00
- time: 6
```

Listing 21: your_machine_folder/asset_manager/shows/preload/show4.yaml

```
#show_version=5
- time: 0
  lights:
    led_01: 006400
    led_02: CCCCCC
    light_01: CC
    light_02: 78
    gi_01: FF
- time: 1
  lights:
    led_01: DarkGreen
    led_02: Black
- time: 2
  lights:
    led_01: DarkSlateGray
    led_02: Tomato
    light_01: FF
    light_02: 33
    gi_01: 99
- time: +1
  lights:
    led_01: MidnightBlue-f500 ms
    led_02: DarkOrange-f0.5 s
    gi_01: 33
- time: 4
  lights:
    led_01: Off-f800
    led_02: Off-f800
    light_01: 00-f800
    light_02: 00-f800
    gi_01: 00
- time: 6
```

Listing 22: your_machine_folder/asset_manager/shows/preload/subfolder/show4b.yaml

```
#show_version=5
- time: 0
  lights:
    led_01: 006400
    led_02: CCCCCC
    light_01: CC
    light_02: 78
    gi_01: FF
- time: 1
  lights:
    led_01: DarkGreen
    led_02: Black
- time: 2
  lights:
    led_01: DarkSlateGray
    led_02: Tomato
    light_01: FF
    light_02: 33
    gi_01: 99
- time: +1
  lights:
    led_01: MidnightBlue-f500 ms
    led_02: DarkOrange-f0.5 s
    gi_01: 33
- time: 4
  lights:
    led_01: Off-f800
    led_02: Off-f800
    light_01: 00-f800
    light_02: 00-f800
    gi_01: 00
- time: 6
```

Listing 23: your_machine_folder/asset_manager/shows/custom1/show13.yaml

```
#show_version=5
- time: 0
  lights:
    led_01: 006400
    led_02: CCCCCC
    light_01: CC
    light_02: 78
    gi_01: FF
- time: 1
  lights:
    led_01: DarkGreen
    led_02: Black
- time: 2
  lights:
    led_01: DarkSlateGray
    led_02: Tomato
    light_01: FF
```

(continues on next page)

(continued from previous page)

```
    light_02: 33
    gi_01: 99
- time: +1
  lights:
    led_01: MidnightBlue-f500 ms
    led_02: DarkOrange-f0.5 s
    gi_01: 33
- time: 4
  lights:
    led_01: Off-f800
    led_02: Off-f800
    light_01: 00-f800
    light_02: 00-f800
    gi_01: 00
- time: 6
```

Listing 24: your_machine_folder/asset_manager/shows/custom1/show11.yaml

```
#show_version=5
- time: 0
  lights:
    led_01: 006400
    led_02: CCCCCC
    light_01: CC
    light_02: 78
    gi_01: FF
- time: 1
  lights:
    led_01: DarkGreen
    led_02: Black
- time: 2
  lights:
    led_01: DarkSlateGray
    led_02: Tomato
    light_01: FF
    light_02: 33
    gi_01: 99
- time: +1
  lights:
    led_01: MidnightBlue-f500 ms
    led_02: DarkOrange-f0.5 s
    gi_01: 33
- time: 4
  lights:
    led_01: Off-f800
    led_02: Off-f800
    light_01: 00-f800
    light_02: 00-f800
    gi_01: 00
- time: 6
```

assets_and_image (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 25: your_machine_folder/assets_and_image/config/test_asset_loading.yaml

```
#config_version=5

modes:
  - mode1

assets:
    images:
        default:
            load: preload
        preload:
            load: preload
            test_key: test_value
        on_demand:
            load: on_demand
        mode_start:
            load: mode_start

images:
  image_12_new_name:
    file: image12.png
    test_key: test_value_override12
  image_13_new_name:
    file: image13.png
  image3:
    test_key: test_value_override3

image_pools:
  group1:
    load: preload
    images:
      - image1
      - image2
      - image3
    type: random
  group2:
    load: preload
    images:
      - image1
      - image2
      - image3|2
    type: random
  group3:
```

(continues on next page)

(continued from previous page)

```
    images:
      - image1
      - image2
      - image3
    type: sequence
  group4:
    images:
      - image1|4
      - image2|2
      - image3
    type: sequence
  group5:
    images:
      - image1|1
      - image2|5
      - image3|1
    type: random_force_next
  group6:
    images:
      - image1
      - image2
      - image3
    type: random_force_all
```

Listing 26: your_machine_folder/assets_and_image/config/test_image.yaml

```
#config_version=5

modes:
  - mode1

displays:
  default:
    width: 400
    height: 300

slides:
  random_image_test:
    - type: image
      image: random_image
      x: 50
  image_test:
    - type: image
      image: image1
      x: 50
      animations:
        show_slide:
          - property: rotation
            value: 360
            duration: 2s
    - type: image
      image: image2
      rotation: 25
      x: 80
```

(continues on next page)

(continued from previous page)

```
    - type: image
      image: image3
      scale: 1.5
      x: 110
    - type: image
      image: image4
      rotation: -45
      x: 140
    - type: image
      image: image5
      x: 170
      animations:
        show_slide:
          - property: scale
            value: 3.0
            duration: 1s
          - property: scale
            value: 0.1
            duration: 1s
          - property: scale
            value: 3.0
            duration: 1s
          - property: scale
            value: 1.0
            duration: 1s
    - type: image
      image: image6
      x: 200
    - type: image
      image: image7
      x: 230
    - type: image
      image: image8
      x: 260
    - type: image
      image: image9
      x: 290
    - type: image
      image: image10
      x: 320
    - type: image
      image: image11
      x: 350
    - type: image
      image: image12
      x: 380

image_pools:
  random_image:
      images:
         - image1
         - image2
         - image3
```

(continues on next page)

(continued from previous page)

```
widgets:
  random_image_widget:
    - type: image
      image: random_image
      rotation: 25
      x: 80

slide_player:
  show_slide1: image_test
  show_random_slide: random_image_test

widget_player:
  add_random_image:
    random_image_widget:
      slide: random_image_test
  remove_random_image:
    random_image_widget:
      action: remove

assets:
    images:
        default:
            load: preload
        preload:
            load: preload
            test_key: test_value
        on_demand:
            load: on_demand
        mode_start:
            load: mode_start
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 27: your_machine_folder/assets_and_image/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  priority: 300

images:
  image6:
    file: image6.png
    load: mode_start
```

audio (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 28: your_machine_folder/audio/config/test_audio_sound_loop.yaml

```
#config_version=5
sound_system:
  buffer: 2048
  frequency: 44100
  channels: 2
  tracks:
    loops:
      type: sound_loop
      volume: 0.6
    music:
      volume: 0.5
      simultaneous_sounds: 1
    sfx:
      volume: 0.4
      simultaneous_sounds: 8
      preload: yes
    voice:
      volume: 0.6
      simultaneous_sounds: 1
      preload: yes

assets:
  sounds:
    default:
      load: preload
    loops:
      load: preload
      track: loops
    voice:
      load: preload
      track: voice
    sfx:
      load: preload
      track: sfx
    music:
      load: on_demand
      track: music
    playlist:
      load: on_demand
      track: sfx

sounds:
  kick:
    loops: -1
    markers:
```

(continues on next page)

(continued from previous page)

```
      - time: 1.0s
        events: kick_marker_1
  hihat:
    loops: -1
    markers:
      - time: 0.5s
        events: hihat_marker_1
      - time: 1.5s
        events: hihat_marker_2

sound_loop_sets:
  hi_hat:
    sound: hihat
    volume: 0.7
    tempo: 130
    events_when_played: hi_hat_played
    events_when_looping: hi_hat_looping
    events_when_stopped: hi_hat_stopped

  basic_beat:
    sound: kick
    volume: 0.5
    tempo: 130
    events_when_played: basic_beat_played
    events_when_looping: basic_beat_looping
    events_when_stopped: basic_beat_stopped

  basic_beat_layers:
    sound: kick
    volume: 0.5
    tempo: 130
    layers:
      - sound: hihat
        volume: 0.7
        initial_state: stop
      - sound: snare
        volume: 0.6
        initial_state: stop
      - sound: clap
        volume: 0.45
        initial_state: stop
    events_when_played: basic_beat_layers_played
    events_when_looping: basic_beat_layers_looping
    events_when_stopped: basic_beat_layers_stopped, sound_loop_set_stopped

  basic_beat2:
    sound: kick2
    volume: 0.5
    tempo: 130
    events_when_played: basic_beat2_played
    events_when_looping: basic_beat2_looping
    events_when_stopped: basic_beat2_stopped

  basic_beat_layers2:
```

(continues on next page)

(continued from previous page)

```
    sound: kick2
    volume: 0.5
    tempo: 130
    layers:
      - sound: hihat
        volume: 0.7
      - sound: snare
        volume: 0.6
      - sound: clap
        volume: 0.45
        initial_state: stop
      - sound: bass_synth
        volume: 0.5
        initial_state: play
    events_when_played: basic_beat_layers2_played

sound_loop_player:
  play_hi_hat:
    loops:
      action: play
      sound_loop_set: hi_hat

  play_basic_beat:
    loops:
      action: play
      sound_loop_set: basic_beat

  play_basic_beat_layers:
    loops:
      action: play
      sound_loop_set: basic_beat_layers
  add_hi_hats:
    loops:
      action: play_layer
      layer: 1
  stop_hi_hats:
    loops:
      action: stop_looping_layer
      layer: 1
  add_snare:
    loops:
      action: play_layer
      fade_in: 2s
      layer: 2
  add_claps:
    loops:
      action: play_layer
      layer: 3

  play_basic_beat2:
    loops:
      action: play
      sound_loop_set: basic_beat2
```

(continues on next page)

(continued from previous page)

```
  play_basic_beat_layers2:
    loops:
      action: play
      sound_loop_set: basic_beat_layers2
      timing: next_beat_interval
      interval: 2
  add_bass_synth:
    loops:
      action: play_layer
      layer: 4
      fade_in: 3s
  fade_out_bass_synth:
    loops:
      action: stop_layer
      layer: 4
      fade_out: 4s

  stop_looping_current_loop:
    loops:
      action: stop_looping
  stop_current_loop:
    loops:
      action: stop
      fade_out: 1.5s
  reset_current_loop:
    loops:
      action: jump_to
      time: 0s
  jump_to_middle_of_loop:
    loops:
      action: jump_to
      time: 0.923s

sound_player:
    play_sound_synthping: 210871_synthping
    basic_beat_layers2_played: 210871_synthping
```

Listing 29: your_machine_folder/audio/config/test_audio_disabled.yaml

```
#config_version=5
sound_system:
  enabled: False

modes:
 - mode1
```

Listing 30: your_machine_folder/audio/config/test_audio_gstreamer.yaml

```
#config_version=5
displays:
  default:
    width: 400
    height: 300
```

(continues on next page)

(continued from previous page)

```
sound_system:
  buffer: 2048
  frequency: 44100
  channels: 2
  tracks:
    music:
      volume: 0.5
      simultaneous_sounds: 1
    sfx:
      volume: 0.3
      simultaneous_sounds: 8
    voice:
      volume: 0.6
      simultaneous_sounds: 1

assets:
  sounds:
    default:
      load: preload
    voice:
      load: preload
      track: voice
    sfx:
      load: on_demand
      track: sfx
    music:
      load: on_demand
      track: music
    loops:
      load: preload
      track: sfx
    playlist:
      load: on_demand
      track: sfx
  videos:
      default:
          load: preload
      preload:
          load: preload
      on_demand:
          load: on_demand
      mode_start:
          load: mode_start

sounds:
  264828_text:
    volume: 0.1
    events_when_played: text_sound_played
    events_when_looping: text_sound_looping
    events_when_stopped: text_sound_stopped
    loops: 6
    simultaneous_limit: 3
    stealing_method: skip
```

(continues on next page)

(continued from previous page)

```
  210871_synthping:
    simultaneous_limit: 3
    stealing_method: oldest
    events_when_played: synthping_played

  198361_sfx-028:
    volume: 0.25

  263774_music:
    volume: 0.4

  city_loop:
    file: 223093__qubodup__seamless-city-loop.flac
    streaming: True
    volume: 0.15
    fade_in: 2.0 sec

sound_player:
  play_sound_text: 264828_text
  play_sound_synthping: 210871_synthping
  play_sound_sfx_028: 198361_sfx-028
  play_city_loop: city_loop
  stop_city_loop:
    city_loop:
      action: stop
      fade_out: 0.1s

slides:
  video_test:
    - type: video
      video: mpf_video_small_test
    - type: text
      text: Sound and Video Test
      y: bottom+20%
    - type: text
      text: ""
      y: bottom+10%

slide_player:
  show_slide1: video_test

videos:
  mpf_video_small_test:
    width: 100
    height: 70
    events_when_played: test_video_played
    events_when_stopped: test_video_stopped
```

Listing 31: your_machine_folder/audio/config/test_audio_playlist.yaml

```
#config_version=5
sound_system:
  buffer: 2048
```

(continues on next page)

(continued from previous page)

```
  frequency: 44100
  channels: 2
  tracks:
    playlist:
      type: playlist
      volume: 0.6
      crossfade_time: 2s

assets:
  sounds:
    default:
      load: preload
    playlist:
      load: preload
      track: playlist

sounds:
  drumbeat_7:
    file: 144554__kxtells__drumbeat-7.ogg
    events_when_played: drumbeat_7_played
    events_when_stopped: drumbeat_7_stopped
  hippie_ahead:
    file: 214473__diboz__hippeahead.ogg
    events_when_played: hippie_ahead_played
    events_when_stopped: hippie_ahead_stopped
  rainbow_disco_bears:
    file: 322071__edemson86__rainbowdiscobears.ogg
    events_when_played: rainbow_disco_bears_played
    events_when_stopped: rainbow_disco_bears_stopped
  dirty_grinding_beat_loop:
    file: 385984__blockh34d__dirty-grinding-beat-loop.ogg
    events_when_played: dirty_grinding_beat_loop_played
    events_when_stopped: dirty_grinding_beat_loop_stopped

playlists:
  attract_music:
    sounds: drumbeat_7, rainbow_disco_bears, dirty_grinding_beat_loop, hippie_ahead
    shuffle: False
    repeat: False
    events_when_played: attract_music_played
    events_when_stopped: attract_music_stopped
    events_when_looping: attract_music_looping
    events_when_sound_changed: attract_music_sound_changed
    events_when_sound_stopped: attract_music_sound_stopped
  other_playlist:
    sounds: hippie_ahead, rainbow_disco_bears
    events_when_played: other_playlist_played
    events_when_stopped: other_playlist_stopped
  third_playlist:
    sounds: dirty_grinding_beat_loop, drumbeat_7
    events_when_played: third_playlist_played
    events_when_stopped: third_playlist_stopped

playlist_player:
```

(continues on next page)

(continued from previous page)

```
  play_attract_music:
    playlist:
      playlist: attract_music
      action: play

  advance_playlist:
    playlist:
      action: advance

  stop_playlist:
    playlist:
      action: stop
```

Listing 32: your_machine_folder/audio/config/test_audio_default_settings.yaml

```
#config_version=5

# No sound_system section, default settings should be used

modes:
 - mode1

assets:
    sounds:
        default:
            load: preload
        voice:
            load: preload
            track: default
        sfx:
            load: on_demand
            track: default
            ducking:
                target: default
                delay: 0
                attack: 0.3 sec
                attenuation: 0.45
                release_point: 0.5 sec
                release: 1.0 sec
        music:
            load: on_demand
            track: default
        loops:
            load: preload
            track: default
        playlist:
            load: on_demand
            track: default
```

Listing 33: your_machine_folder/audio/config/test_audio_bad_buffer_setting.yaml

```
#config_version=5
sound_system:
```

(continues on next page)

(continued from previous page)

```
  buffer: 1000 # Not a power or two as required
  tracks:
    voice:
      volume: 0.6
      simultaneous_sounds: 1
      preload: yes
    sfx:
      volume: 0.4
      simultaneous_sounds: 8
      preload: yes
    music:
      volume: 0.5
      simultaneous_sounds: 1

modes:
 - mode1

assets:
    sounds:
        default:
            load: preload
        voice:
            load: preload
            track: voice
        sfx:
            load: on_demand
            track: sfx
        music:
            load: on_demand
            track: music
        loops:
            load: preload
            track: sfx
        playlist:
            load: on_demand
            track: sfx
```

Listing 34: your_machine_folder/audio/config/test_audio.yaml

```
#config_version=5
sound_system:
  buffer: 2048
  frequency: 44100
  channels: 2
  tracks:
    music:
      volume: 0.5
      simultaneous_sounds: 1
      events_when_stopped: music_track_stopped
      events_when_played: music_track_played, keep_going
      events_when_paused: music_track_paused
    sfx:
      volume: 0.4
      simultaneous_sounds: 8
```

(continues on next page)

(continued from previous page)

```
      preload: yes
    voice:
      volume: 0.6
      simultaneous_sounds: 1
      preload: yes


modes:
 - mode1
 - mode2

assets:
    sounds:
        default:
            load: preload
        voice:
            load: preload
            track: voice
        sfx:
            load: preload
            track: sfx
        music:
            load: on_demand
            track: music
        loops:
            load: preload
            track: sfx
        playlist:
            load: on_demand
            track: sfx

sounds:
    264828_text:
        events_when_played: text_sound_played
        events_when_looping: text_sound_looping
        events_when_stopped: text_sound_stopped
        loops: 7
        simultaneous_limit: 3
        stealing_method: skip
    104457_moron_test:
        streaming: False
        events_when_played: moron_test_played
        events_when_stopped: moron_test_stopped
        events_when_about_to_finish: moron_test_about_to_finish
        volume: 0.6
        about_to_finish_time: 2s
        ducking:
            target: music
            delay: 0
            attack: 1.0sec
            attenuation: -18db
            release_point: 3sec
            release: 2.25sec
        markers:
```

(continues on next page)

(continued from previous page)

```
            - time: 2.5sec
              events: moron_marker
            - time: 3.5sec
              name: verse_1
              events: moron_next_marker, last_marker
            - time: 5.39sec
              name: about_to_finish
              events: moron_about_to_finish_marker
    210871_synthping:
        priority: 1
        simultaneous_limit: 3
        stealing_method: oldest
        events_when_played: synthping_played
        max_queue_time: 2s
    198361_sfx-028:
        simultaneous_limit: 3
        stealing_method: newest
    263774_music:
        streaming: False
    looptest:
        loop_start_at: 1.8461538s
        loop_end_at: 3.6923077s
        loops: 3
        streaming: False
        events_when_played: looptest_played
        events_when_looping: looptest_looping
        events_when_stopped: looptest_stopped

sound_pools:
    drum_group:
        load: preload
        type: sequence
        simultaneous_limit: 3
        stealing_method: skip
        track: sfx
        sounds:
            - 4832__zajo__drum07
            - 84480__zgump__drum-fx-4
            - 100184__menegass__rick-drum-bd-hard

sound_player:
    load_music:
        263774_music:
            action: load
    unload_music:
        263774_music:
            action: unload
    play_sound_synthping: 210871_synthping
    play_sound_text:
        264828_text:
            loops: -1
            priority: 100
    stop_sound_looping_text:
        264828_text:
```

(continues on next page)

(continued from previous page)

```
            action: stop_looping
    play_sound_moron_test: 104457_moron_test
    stop_sound_moron_test:
        104457_moron_test:
            action: stop
    play_sound_test:
        113690_test:
            volume: 0.25
    play_sound_music:
        263774_music:
            volume: 0.5
    stop_sound_music:
        263774_music:
            action: stop
    play_sound_drum_group: drum_group
    play_sound_text_default_params: 264828_text
    play_sound_text_param_set_1:
        264828_text:
            volume: 0.67
            loops: 2
            priority: 1000
            start_at: 0.05s
            fade_in: 0.25s
            fade_out: 0.1s
            max_queue_time: 0.15s
            events_when_played: text_sound_played_param_set_1
            events_when_stopped: text_sound_stopped_param_set_1
            events_when_looping: text_sound_looping_param_set_1

track_player:
    stop_all_tracks:
        __all__:
            action: stop
            fade: 1.5 sec
    stop_music_track:
        music:
            action: stop
            fade: 1.5 sec
    play_music_track:
        music:
            action: play
            fade: 1.5 sec
    pause_music_track:
        music:
             action: pause
    resume_music_track:
        music:
             action: play
    set_music_track_volume_loud:
        music:
             action: set_volume
             volume: 0.95
             fade: 0.5 sec
    set_music_track_volume_quiet:
```

(continues on next page)

(continued from previous page)

```
      music:
           action: set_volume
           volume: 0.3
           fade: 0.5 sec
  stop_all_sounds_on_music_track:
      music:
           action: stop_all_sounds
           fade: 0.5 sec
  stop_all_sounds:
      __all__:
           action: stop_all_sounds
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 35: your_machine_folder/audio/modes/mode2/config/mode2.yaml

```
#config_version=5

mode:
  priority: 1000

sounds:
    boing_mode2:
        file: 140867__juskiddink__boing.wav
        events_when_played: boing_sound_played

sound_player:
    play_sound_boing_in_mode2: boing_mode2
    play_sound_music_fade_at_mode_end:
        263774_music:
            volume: 0.8
            mode_end_action: stop
            fade_out: 1s
    play_slingshot_sound: boing_mode2
    play_slingshot_sound_with_express_config_block: boing_mode2|block
    play_slingshot_sound_with_block:
        boing_mode2:
            block: true
```

Listing 36: your_machine_folder/audio/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  priority: 500

sound_player:
```

(continues on next page)

(continued from previous page)

```
    play_sound_synthping_in_mode: 210871_synthping
    play_sound_drum_group_in_mode: drum_group
    play_slingshot_sound: 210871_synthping
    play_slingshot_sound_with_express_config_block: 210871_synthping
    play_slingshot_sound_with_block:
        210871_synthping:
            block: true
```

auditor (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 37: your_machine_folder/auditor/config/config.yaml

```
#config_version=5

game:
    balls_per_game: 1

auditor:
    events:
        - test_event1
        - test_event2
    player:
        - my_var

modes:
    - base

switches:
    s_test:
        number:
    s_start:
        number:
        tags: start
    s_ball:
        number:
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 38: your_machine_folder/auditor/modes/base/config/base.yaml

```
#config_version=5
mode:
  start_events: ball_started
```

(continues on next page)

(continued from previous page)

```
variable_player:
    add_score:
      score: 100
    add_custom:
      my_var: 100
    add_not_audited:
      not_audited: 100
```

autofire (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 39: your_machine_folder/autofire/config/config.yaml

```
#config_version=5

switches:
    s_test:
        number: 7
    s_test_disabled:
        number: 8
    s_test_nc:
        number: 1A
        type: 'NC'
    s_test_debounce_on:
        number: 9
        debounce: normal

coils:
    c_test:
        number: 4
        default_pulse_ms: 23
    c_test2:
        number: 5
        default_pulse_ms: 23
    c_test_disabled:
        number: 6
    c_test_recycle_off:
        number: 7
        default_recycle: False

autofire_coils:
    ac_test:
        coil: c_test
        switch: s_test
    ac_test_inverted:
        coil: c_test2
```

(continues on next page)

(continued from previous page)

```
        switch: s_test_nc
    ac_test_inverted2:
        coil: c_test2
        switch: s_test
        reverse_switch: True
    ac_test_timeout:
        coil: c_test
        switch: s_test
        timeout_watch_time: 1s
        timeout_max_hits: 10
        timeout_disable_time: 500ms
    ac_test_disabled:
        coil: c_test_disabled
        switch: s_test_disabled
        enable_events: enable_autofire
        disable_events: disable_autofire

    ac_test_defaults:
        coil: c_test_recycle_off
        switch: s_test_debounce_on

    ac_test_overwrites:
        coil: c_test
        switch: s_test
        switch_overwrite:
            debounce: normal
        coil_overwrite:
            recycle: False

    ac_test_overwrites2:
        coil: c_test_recycle_off
        switch: s_test_debounce_on
        switch_overwrite:
            debounce: quick
        coil_overwrite:
            recycle: True
```

ball_controller (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 40: your_machine_folder/ball_controller/config/config.yaml

```
#config_version=5
```

(continues on next page)

(continued from previous page)

```
game:
    balls_per_game: 1

machine:
    min_balls: 3

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:
    eject_coil3:
        number:

switches:
    s_start:
        number:
        tags: start
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch3:
        number:
    s_ball_switch4:
        number:
    s_ball_switch_launcher:
        number:
    s_vuk:
        number:
    s_playfield:
        number:
        tags: playfield_active

playfields:
    playfield:
        default_source_device: test_launcher
        tags: default

ball_devices:
    test_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2, s_ball_switch3, s_ball_switch4
        debug: true
        eject_targets: test_launcher
        tags: trough, drain, home
    test_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
        debug: true
    test_vuk:
        eject_coil: eject_coil3
        ball_switches: s_vuk
        debug: true
```

Listing 41: your_machine_folder/ball_controller/config/regression.yaml

```
#config_version=5

playfields:
  playfield:
    enable_ball_search: True
    default_source_device: shooter_lane
    tags: default

machine:
  balls_installed: 6

switches:
  s_shooter_lane:
    number:
  s_trough_1:
    number:
  s_trough_2:
    number:
  s_trough_3:
    number:
  s_trough_4:
    number:
  s_trough_5:
    number:
  s_trough_6:
    number:
  s_trough_jam:
    number:
  s_popBumperAreaEject:
    number:
  s_underRightRampEject:
    number:
  s_underRightRampJam:
    number:
  s_sandTrap:
    number:

coils:
  c_plunger_lane:
    number:
  c_trough_eject:
    number:
  c_PopBumperAreaEject:
    number:
  c_UpperRightEject:
    number:
  c_SandTrapEject:
    number:

ball_devices:
  shooter_lane:
    ball_switches: s_shooter_lane
```

(continues on next page)

(continued from previous page)

```
    eject_coil: c_plunger_lane
    player_controlled_eject_event: sw_plunger # for flipper launch
    mechanical_eject: true # player can plunge as well
    eject_timeouts: 2s
    ball_search_order: 1

  trough:
    tags: trough, home, drain
    ball_switches: s_trough_1, s_trough_2, s_trough_3, s_trough_4, s_trough_5, s_trough_6, s_trough_jam
    eject_coil: c_trough_eject
    confirm_eject_type: target
    eject_targets: shooter_lane
    eject_timeouts: 2s # default is 10 seconds, these needs to be lowered for multiballs
    jam_switch: s_trough_jam

  PopsEject:
    ball_switches: s_popBumperAreaEject
    eject_coil: c_PopBumperAreaEject
    ball_search_order: 1230 # default 200 so do this last
    entrance_event_timeout: 2s # default is 5 second

  underRightRampEject:
    ball_switches: s_underRightRampEject
    eject_coil: c_UpperRightEject
    ball_search_order: 1220 # default 200 so do this last
    auto_fire_on_unexpected_ball: true
    entrance_event_timeout: 1500ms  # default is 5 second
    jam_switch: s_underRightRampJam # only happens if 2 balls in there, one on top of the other
    eject_coil_jam_pulse: 100      # if jammed, pulse harder since 2 balls there (in ms)

  sandTrapEject:
    ball_switches: s_sandTrap
    eject_coil: c_SandTrapEject
    ball_search_order: 2
    auto_fire_on_unexpected_ball: true
    entrance_event_timeout: 400ms # default is 5 second

virtual_platform_start_active_switches:
  - s_trough_1
  - s_trough_2
  - s_trough_3
  - s_trough_4
```

ball_device (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 42: your_machine_folder/ball_device/config/test_single_device.yaml

```
#config_version=5

playfields:
    playfield:
        default_source_device: trough
        tags: default

coils:
    c_eject:
        number:

switches:
    s_trough:
        number:

virtual_platform_start_active_switches:
  s_trough

ball_devices:
    trough:
        eject_coil: c_eject
        ball_switches: s_trough
        tags: home, trough, drain
        debug: True
```

Listing 43: your_machine_folder/ball_device/config/test_player_controlled_eject.yaml

```
#config_version=5

switches:
  s_start_button:
    number:
    tags: start
  s_launch_button:
    number:
  s_plunger_lane:
    number:
  s_trough1:
    number:
  s_trough2:
    number:
  s_trough3:
    number:

coils:
  c_plunger:
    number:
  c_trough_eject:
    number:

ball_devices:
  bd_trough:
```

(continues on next page)

(continued from previous page)

```
    ball_switches: s_trough1, s_trough2, s_trough3
    eject_coil: c_trough_eject
    tags: trough, home, drain
    eject_targets: bd_plunger
    eject_timeouts: 3s
    debug: true
  bd_plunger:
    ball_switches: s_plunger_lane
    eject_coil: c_plunger
    player_controlled_eject_event: s_launch_button_active
    eject_timeouts: 1s
    debug: true

playfields:
  playfield:
    default_source_device: bd_plunger
    tags: default
    debug: true

virtual_platform_start_active_switches: s_trough1, s_trough2, s_trough3
```

Listing 44: your_machine_folder/ball_device/config/trough_entrance_switch.yaml

```
#config_version=5
playfields:
  playfield:
    default_source_device: bd_trough
    tags: default
switches:
  s_drain:
    number: 01
  s_trough_enter:
    number: 02
coils:
  c_drain_eject:
    number: 03
    default_pulse_ms: 20
  c_trough_release:
    number: 04
    default_pulse_ms: 20
ball_devices:
  bd_drain:
    ball_switches: s_drain
    eject_coil: c_drain_eject
    eject_targets: bd_trough
    tags: drain
    eject_timeouts: 4s
  bd_trough:
    counter:
      class: mpf.devices.ball_device.entrance_switch_counter.EntranceSwitchCounter
      entrance_switch: s_trough_enter
      entrance_switch_full_timeout: 500ms
      settle_time_ms: 3000
    ball_capacity: 3
```

(continues on next page)

(continued from previous page)

```
    eject_coil: c_trough_release
    tags: trough, home
    eject_timeouts: 3s

machine:
  balls_installed: 4
```

Listing 45: your_machine_folder/ball_device/config/test_ball_device_switch_confirmation.yaml

```
#config_version=5

game:
    balls_per_game: 1

playfields:
    playfield:
        default_source_device: test_target1
        tags: default

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:
    eject_coil3:
        number:
    eject_coil4:
        number:
    eject_coil5:
        number:

switches:
    s_start:
        number:
        tags: start
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch_launcher:
        number:
    s_launcher_confirm:
        number:
    s_ball_switch_target1:
        number:
    s_ball_switch_target2_1:
        number:
    s_ball_switch_target2_2:
        number:
    s_ball_switch_target3:
        number:
    s_playfield:
        number:
        tags: playfield_active
```

(continues on next page)

(continued from previous page)

```
ball_devices:
    test_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2
        debug: true
        confirm_eject_type: target
        eject_targets: test_launcher
        tags: trough, drain, home
    test_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
        confirm_eject_switch: s_launcher_confirm
        debug: true
        confirm_eject_type: switch
        eject_targets: test_target1, test_target2
        eject_timeouts: 6s, 10s
    test_target1:
        eject_coil: eject_coil3
        ball_switches: s_ball_switch_target1
        debug: true
        confirm_eject_type: target
    test_target2:
        eject_coil: eject_coil4
        ball_switches: s_ball_switch_target2_1, s_ball_switch_target2_2
        debug: true
        tags: trough, drain, home
        confirm_eject_type: target
        eject_targets: test_target3
    test_target3:
        eject_coil: eject_coil5
        ball_switches: s_ball_switch_target3
        debug: true
```

Listing 46: your_machine_folder/ball_device/config/test_enable_coil.yaml

```
#config_version=5

playfields:
    playfield:
        default_source_device: test
        tags: default

coils:
    eject_coil:
        default_hold_power: 0.25
        default_pulse_ms: 20
        number:

switches:
    s_ball1:
        number:
    s_ball2:
        number:
```

(continues on next page)

(continued from previous page)

```
ball_devices:
    test:
        eject_coil: eject_coil
        eject_coil_enable_time: 400ms
        ball_switches: s_ball1, s_ball2
        tags: home, trough
        debug: true
```

Listing 47: your_machine_folder/ball_device/config/test_ball_device_manual_with_target.yaml

```
#config_version=5

playfields:
    playfield:
        default_source_device: test_launcher
        tags: default

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:
    eject_coil3:
        number:
    eject_coil4:
        number:
    eject_coil5:
        number:
    eject_coil6:
        number:


switches:
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch_launcher:
        number:
    s_ball_switch_launcher2:
        number:
    s_ball_switch_target:
        number:
    s_playfield:
        number:
        tags: playfield_active
    s_launch:
        number:
        tags: launch
    s_vuk:
        number:
```

(continues on next page)

(continued from previous page)

```
ball_devices:
    test_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2
        debug: true
        confirm_eject_type: target
        eject_targets: test_launcher
        eject_timeouts: 3s
        tags: trough, drain, home
    test_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
        debug: true
        eject_timeouts: 6s, 10s
        eject_targets: playfield, test_target
        mechanical_eject: true
        confirm_eject_type: target
    test_target:
        eject_coil: eject_coil3
        ball_switches: s_ball_switch_target
        debug: true
        eject_timeouts: 6s
        confirm_eject_type: target
    test_launcher_manual_on_unexpected:
        eject_coil: eject_coil4
        ball_switches: s_ball_switch_launcher2
        debug: true
        eject_timeouts: 6s
        eject_targets: playfield
        mechanical_eject: true
        auto_fire_on_unexpected_ball: false
        confirm_eject_type: target
    test_vuk:
        eject_coil: eject_coil5
        ball_switches: s_vuk
        debug: true
        eject_timeouts: 3s
        eject_targets: test_launcher
        auto_fire_on_unexpected_ball: false
        confirm_eject_type: target
```

Listing 48: your_machine_folder/ball_device/config/test_gottlieb_trough_with_initial_balls.yaml

```
#config_version=5

config: test_gottlieb_trough.yaml

virtual_platform_start_active_switches:
  trough_entry
```

Listing 49: your_machine_folder/ball_device/config/test_jam_and_ball_left.yaml

```
#config_version=5
playfields:
    playfield:
        tags: default
        default_source_device: bd_plunger

switches:
    s_plunger_lane:
        number: 1
    s_trough1:
        number: 2
    s_trough2:
        number: 3
    s_trough3:
        number: 4
    s_trough4:
        number: 5
    s_trough_jam:
        number: 6
    s_playfield:
        number: 7
        tags: playfield_active
    s_start:
        number: 10
        tags: start

coils:
    c_trough_eject:
        number: A2-B0-7
        default_pulse_ms: 10
    c_plunger:
        number: A2-B1-6
        default_pulse_ms: 40
ball_devices:
    bd_plunger:
        ball_switches: s_plunger_lane
        mechanical_eject: true
        eject_timeouts: 3s
        eject_coil: c_plunger
        debug: true
        file_log: full
    bd_trough:
        ball_switches: s_trough1, s_trough2, s_trough3, s_trough4, s_trough_jam
        eject_coil: c_trough_eject
        tags: trough, home, drain
        jam_switch: s_trough_jam
        eject_coil_jam_pulse: 10ms
        eject_targets: bd_plunger
        eject_timeouts: 1500ms
        debug: yes
```

Listing 50: your_machine_folder/ball_device/config/test_modern_trough_plunger_setup.yaml

```
#config_version=5
playfields:
    playfield:
        default_source_device: bd_plunger
        tags: default

coils:
    c_trough_eject:
        number:
    c_autolauncher:
        number:

switches:
    s_trough_switch1:
        number:
    s_trough_switch2:
        number:
    s_trough_switch3:
        number:
    s_trough_jam:
        number:
    s_ball_switch_plunger_lane:
        number:
    s_playfield:
        number:
        tags: playfield_active

ball_devices:
    bd_trough:
        eject_coil: c_trough_eject
        ball_switches: s_trough_switch1, s_trough_switch2, s_trough_switch3
        jam_switch: s_trough_jam
        eject_targets: bd_plunger
        eject_timeouts: 3s
        tags: trough, drain, home
        debug: true
    bd_plunger:
        eject_coil: c_autolauncher
        ball_switches: s_ball_switch_plunger_lane
        mechanical_eject: True
        eject_targets: playfield
        eject_timeouts: 4s
        debug: true
```

Listing 51: your_machine_folder/ball_device/config/test_gottlieb_trough.yaml

```
#config_version=5

game:
    balls_per_game: 3
    allow_start_with_ball_in_drain: True
```

(continues on next page)

(continued from previous page)

```
machine:
    min_balls: 3

playfields:
    playfield:
        default_source_device: plunger
        tags: default

coils:
    outhole:
        number: 1
    trough:
        number: 2

switches:
    start:
        number: 1
        tags: start
    outhole:
        number: 2
    trough_entry:
        number: 3
    plunger:
        number: 4
    playfield:
        number: 5
        tags: playfield_active

ball_devices:
    outhole:
        tags: drain
        ball_switches: outhole
        eject_timeouts: 2s
        eject_coil: outhole
        eject_targets: trough
        confirm_eject_type: target
        debug: true
    trough:
        tags: trough, home
        entrance_switch: trough_entry
        entrance_switch_full_timeout: 3s
        eject_coil: trough
        eject_targets: plunger
        confirm_eject_type: target
        ball_capacity: 3
        debug: true
    plunger:
        ball_switches: plunger
        mechanical_eject: true
        eject_timeouts: 4s
        debug: true
```

Listing 52: your_machine_folder/ball_device/config/test_ball_device_event_ejector.yaml

```
#config_version=5

game:
    balls_per_game: 1

playfields:
    playfield:
        default_source_device: test_trough

switches:
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_playfield:
        number:
        tags: playfield_active

ball_devices:
    test_trough:
        ejector:
          class: mpf.devices.ball_device.event_ejector.EventEjector
          events_when_eject_try: trough_eject
        ball_switches: s_ball_switch1, s_ball_switch2
        debug: true
        tags: trough, drain, home
```

Listing 53: your_machine_folder/ball_device/config/test_pulse_eject.yaml

```
#config_version=5

playfields:
  playfield:
    default_source_device: test
    tags: default

coils:
  eject_coil:
    number:

switches:
  s_ball1:
    number:
  s_ball2:
    number:
  s_ball3:
    number:
  s_ball4:
    number:

ball_devices:
  test:
```

(continues on next page)

(continued from previous page)

```
    ejector:
      class: mpf.devices.ball_device.pulse_coil_ejector.PulseCoilEjector
      eject_coil: eject_coil
      eject_times: 40ms, 20ms, 15ms

    ball_switches: s_ball1, s_ball2, s_ball3, s_ball4
    tags: home, trough
    debug: true
```

Listing 54: your_machine_folder/ball_device/config/test_ball_device_auto_manual_plunger.yaml

```
#config_version=5

coils:
    trough_eject:
        number:
    plunger_eject:
        number:

playfields:
    playfield:
        default_source_device: plunger
        tags: default

switches:
    s_trough_1:
        number:
    s_trough_2:
        number:
    s_plunger:
        number:
    s_playfield:
        number:
        tags: playfield_active
    s_launch:
        number:
        tags: launch

ball_devices:
    trough:
        eject_coil: trough_eject
        ball_switches: s_trough_1, s_trough_2
        debug: true
        tags: trough, drain, home
        eject_targets: plunger
        confirm_eject_type: target
    plunger:
        eject_coil: plunger_eject
        ball_switches: s_plunger
        debug: true
        mechanical_eject: true
        player_controlled_eject_event: sw_launch
```

Listing 55: your_machine_folder/ball_device/config/trough_entrance_switch_initial_balls.yaml

```
#config_version=5
config:
  - trough_entrance_switch.yaml

virtual_platform_start_active_switches: s_trough_enter
```

Listing 56: your_machine_folder/ball_device/config/test_ball_device_jam_switch_initial.yaml

```
#config_version=5
config:
 - test_ball_device_jam_switch.yaml

virtual_platform_start_active_switches:
  - s_trough_jam
```

Listing 57: your_machine_folder/ball_device/config/test_ball_device_no_plunger_switch.yaml

```
#config_version=5

playfields:
    playfield:
        default_source_device: trough
        tags: default

coils:
    trough_eject:
        number:

switches:
    s_trough_1:
        number:
    s_trough_2:
        number:
    s_trough_3:
        number:
    s_trough_4:
        number:
    s_trough_jam:
        number:
    s_playfield:
        number:
        tags: playfield_active

ball_devices:
    trough:
        eject_coil: trough_eject
        ball_switches: s_trough_1, s_trough_2, s_trough_3, s_trough_4
        debug: true
        tags: trough, drain, home
```

Listing 58: your_machine_folder/ball_device/config/test_ball_device_routing.yaml

```
#config_version=5

game:
    balls_per_game: 1

playfields:
    playfield:
        default_source_device: test_target1
        tags: default

coils:
    c_trough1:
        number:
    c_trough2:
        number:
    c_launcher:
        number:
    c_target1:
        number:
    c_drain1:
        number:

switches:
    s_trough1_1:
        number:
    s_trough1_2:
        number:
    s_trough2_1:
        number:
    s_trough2_2:
        number:
    s_launcher:
        number:
    s_target1:
        number:
    s_drain1:
        number:
    s_playfield:
        number:
        tags: playfield_active

ball_devices:
    test_trough1:
        eject_coil: c_trough1
        ball_switches: s_trough1_1, s_trough1_2
        eject_targets: test_launcher
        tags: trough, drain, home
    test_launcher:
        eject_coil: c_launcher
        ball_switches: s_launcher
        eject_targets: test_trough2, test_target1
    test_target1:
```

(continues on next page)

(continued from previous page)

```
        eject_coil: c_target1
        ball_switches: s_target1
    test_trough2:
        eject_coil: c_trough2
        ball_switches: s_trough2_1, s_trough2_2
        tags: trough, drain, home
        confirm_eject_type: target
    test_drain:
        eject_coil: c_drain1
        ball_switches: s_drain1
        tags: drain
        eject_targets: playfield, test_target1, test_trough2
```

Listing 59: your_machine_folder/ball_device/config/test_ball_device_jam_switch.yaml

```
#config_version=5

coils:
    trough_eject:
        number:
    plunger_eject:
        number:

playfields:
    playfield:
        default_source_device: plunger
        tags: default

switches:
    s_trough_1:
        number:
    s_trough_2:
        number:
    s_trough_3:
        number:
    s_trough_4:
        number:
    s_trough_jam:
        number:
    s_plunger:
        number:
    s_playfield:
        number:
        tags: playfield_active
    s_launch:
        number:
        tags: launch

ball_devices:
    trough:
        eject_coil: trough_eject
        ball_switches: s_trough_1, s_trough_2, s_trough_3, s_trough_4, s_trough_jam
        jam_switch: s_trough_jam
        debug: true
```

(continues on next page)

(continued from previous page)

```
        tags: trough, drain, home
        eject_targets: plunger
        confirm_eject_type: target
        eject_coil_jam_pulse: 5
        eject_coil_reorder_pulse: 2
        eject_coil_retry_pulse: 15
    plunger:
        eject_coil: plunger_eject
        ball_switches: s_plunger
        debug: true
        mechanical_eject: true
        player_controlled_eject_event: sw_launch
```

Listing 60: your_machine_folder/ball_device/config/test_system_11_trough.yaml

```
#config_version=5

game:
    balls_per_game: 3
    allow_start_with_ball_in_drain: True

playfields:
    playfield:
        default_source_device: plunger
        tags: default

coils:
    outhole:
        number: C09
        default_pulse_ms: 20
    trough:
        number: C10
        default_pulse_ms: 20

switches:
    start:
        number: S13
        tags: start
    outhole:
        number: S15
    trough1:
        number: S16
    trough2:
        number: S17
    trough3:
        number: S18
    plunger:
        number: S28
    playfield:
        number:
        tags: playfield_active

ball_devices:
    outhole:
```

(continues on next page)

(continued from previous page)

```
        tags: drain
        ball_switches: outhole
        eject_coil: outhole
        eject_targets: trough
        confirm_eject_type: target
        debug: true
    trough:
        tags: trough, home
        ball_switches: trough1, trough2, trough3
        eject_coil: trough
        eject_targets: plunger
        confirm_eject_type: target
        debug: true
    plunger:
        ball_switches: plunger
        mechanical_eject: true
        eject_timeouts: 4s
        debug: true
```

Listing 61: your_machine_folder/ball_device/config/test_too_long_exit_count_delay.yaml

```
#config_version=5

playfields:
    playfield:
        default_source_device: plunger
        tags: default

coils:
    trough_eject:
        number:
    plunger_eject:
        number:

switches:
    s_trough_1:
        number:
    s_trough_2:
        number:
    s_trough_3:
        number:
    s_trough_4:
        number:
    s_trough_jam:
        number:
    s_plunger:
        number:
    s_playfield:
        number:
        tags: playfield_active
    s_launch:
        number:
        tags: launch
```

(continues on next page)

(continued from previous page)

```
ball_devices:
    trough:
        eject_coil: trough_eject
        ball_switches: s_trough_1, s_trough_2, s_trough_3, s_trough_4, s_trough_jam
        jam_switch: s_trough_jam
        debug: true
        tags: trough, drain, home
        eject_targets: plunger
        confirm_eject_type: target
        exit_count_delay: 3s
    plunger:
        eject_coil: plunger_eject
        ball_switches: s_plunger
        debug: true
#        mechanical_eject: true
        player_controlled_eject_event: sw_launch
        exit_count_delay: 300ms
```

Listing 62: your_machine_folder/ball_device/config/test_enable_coil_multiple.yaml

```
#config_version=5

playfields:
    playfield:
        default_source_device: test
        tags: default

coils:
    eject_coil:
        default_hold_power: 0.25
        default_pulse_ms: 20
        number:

switches:
    s_ball1:
        number:
    s_ball2:
        number:

ball_devices:
    test:
        eject_coil: eject_coil
        eject_coil_enable_time: 600ms, 200ms
        ball_switches: s_ball1, s_ball2
        tags: home, trough
        debug: true
```

Listing 63: your_machine_folder/ball_device/config/test_ball_device_event_confirmation.yaml

```
#config_version=5

game:
    balls_per_game: 1
```

(continues on next page)

(continued from previous page)

```
playfields:
    playfield:
        default_source_device: test_target1
        tags: default

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:
    eject_coil3:
        number:
    eject_coil4:
        number:
    eject_coil5:
        number:

switches:
    s_start:
        number:
        tags: start
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch_launcher:
        number:
    s_ball_switch_target1:
        number:
    s_ball_switch_target2_1:
        number:
    s_ball_switch_target2_2:
        number:
    s_ball_switch_target3:
        number:
    s_playfield:
        number:
        tags: playfield_active

ball_devices:
    test_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2
        debug: true
        confirm_eject_type: target
        eject_targets: test_launcher
        tags: trough, drain, home
    test_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
        debug: true
        confirm_eject_type: event
        confirm_eject_event: launcher_confirm
```

(continues on next page)

(continued from previous page)

```
        eject_targets: test_target1, test_target2
        eject_timeouts: 6s, 10s
    test_target1:
        eject_coil: eject_coil3
        ball_switches: s_ball_switch_target1
        debug: true
        confirm_eject_type: target
    test_target2:
        eject_coil: eject_coil4
        ball_switches: s_ball_switch_target2_1, s_ball_switch_target2_2
        debug: true
        tags: trough, drain, home
        confirm_eject_type: target
        eject_targets: test_target3
    test_target3:
        eject_coil: eject_coil5
        ball_switches: s_ball_switch_target3
        debug: true
```

Listing 64: your_machine_folder/ball_device/config/test_playfield_lock.yaml

```
#config_version=5
coils:
    eject_coil1:
        number:

switches:
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:

ball_devices:
    test_device:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2
        entrance_events: entrance_event
        debug: true

ball_holds:
    hold_test:
        hold_devices: test_device
        balls_to_hold: 1
        release_one_events: release_test
```

Listing 65: your_machine_folder/ball_device/config/test_ball_device_trigger_events.yaml

```
#config_version=5

playfields:
    playfield:
        default_source_device: test_launcher
        tags: default
```

(continues on next page)

(continued from previous page)

```
coils:
    eject_coil1:
        number:
    eject_coil2:
        number:
    eject_coil3:
        number:
    eject_coil4:
        number:
    eject_coil5:
        number:
    c_diverter:
        number:

switches:
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch_launcher:
        number:
    s_ball_switch_target:
        number:
    s_playfield:
        number:
        tags: playfield_active
    s_launch:
        number:
        tags: launch


ball_devices:
    test_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2
        debug: true
        confirm_eject_type: target
        eject_targets: test_launcher
        eject_timeouts: 3s
        tags: trough, drain, home
    test_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
        debug: true
        eject_timeouts: 6s, 10s
        eject_targets: playfield, test_target
        confirm_eject_type: target
        player_controlled_eject_event: sw_launch
    test_target:
        eject_coil: eject_coil3
        ball_switches: s_ball_switch_target
        debug: true
        eject_timeouts: 6s
```

(continues on next page)

(continued from previous page)

```
        confirm_eject_type: target
```

Listing 66: your_machine_folder/ball_device/config/test_ball_device.yaml

```
#config_version=5

game:
    balls_per_game: 1

playfields:
    playfield:
        default_source_device: test_target1
        tags: default

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:
    eject_coil3:
        number:
    eject_coil4:
        number:
    eject_coil5:
        number:

switches:
    s_start:
        number:
        tags: start
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch_launcher:
        number:
    s_ball_switch_target1:
        number:
    s_ball_switch_target2_1:
        number:
    s_ball_switch_target2_2:
        number:
    s_ball_switch_target3:
        number:
    s_ball_switch_target3_2:
        number:
    s_playfield:
        number:
        tags: playfield_active
    s_entrance:
        number:

ball_devices:
    test_trough:
```

(continues on next page)

(continued from previous page)

```
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2
        debug: true
        confirm_eject_type: target
        max_eject_attempts: 3
        eject_targets: test_launcher
        tags: trough, drain, home
    test_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
        debug: true
        confirm_eject_type: target
        eject_targets: test_target1, test_target2
        eject_timeouts: 6s, 10s
    test_target1:
        eject_coil: eject_coil3
        ball_switches: s_ball_switch_target1
        debug: true
        confirm_eject_type: target
    test_target2:
        eject_coil: eject_coil4
        ball_switches: s_ball_switch_target2_1, s_ball_switch_target2_2
        debug: true
        tags: trough, drain, home
        confirm_eject_type: target
        eject_targets: test_target3
    test_target3:
        eject_coil: eject_coil5
        ball_switches: s_ball_switch_target3, s_ball_switch_target3_2
        eject_targets: playfield, test_trough
        confirm_eject_type: target
        debug: true
    test_entrance_ignore_device:
        ball_capacity: 2
        eject_coil: eject_coil5
        entrance_switch: s_entrance
        entrance_switch_ignore_window_ms: 3000
```

Listing 67: your_machine_folder/ball_device/config/test_hold_coil.yaml

```
#config_version=5

playfields:
    playfield:
        default_source_device: test
        tags: default

coils:
    hold_coil:
        number:
    hold_coil2:
        number:
    hold_coil3:
        number:
```

(continues on next page)

(continued from previous page)

```
    hold_coil4:
        number:

switches:
    s_entrance:
        number:
    s_entrance2:
        number:
    s_entrance_and_hold3:
        number:
    s_ball4_1:
        number:
    s_ball4_2:
        number:

ball_devices:
    test:
        hold_coil: hold_coil
        entrance_switch: s_entrance
        hold_events: test_hold_event
        ball_capacity: 3
        debug: true
        confirm_eject_type: fake
    test2:
        hold_coil: hold_coil2
        entrance_switch: s_entrance2
        hold_events: test_hold_event2
        ball_capacity: 3
        tags: trough, home
        debug: true
        confirm_eject_type: fake
    test3:
        hold_coil: hold_coil3
        entrance_switch: s_entrance_and_hold3
        hold_switches: s_entrance_and_hold3
        tags: trough, home
        debug: true
        eject_timeouts: 2s
        ball_capacity: 2
    test4:
        hold_coil: hold_coil4
        hold_switches: s_ball4_1, s_ball4_2
        ball_switches: s_ball4_1, s_ball4_2
        tags: trough, home
        debug: true
```

Listing 68: your_machine_folder/ball_device/config/test_system_11_trough_startup.yaml

```
#config_version=5

config: test_system_11_trough.yaml

virtual_platform_start_active_switches:
  - trough1
```

(continues on next page)

(continued from previous page)

```
  - trough2
  - trough3
  - outhole
```

ball_holds (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 69: your_machine_folder/ball_holds/config/test_ball_holds.yaml

```
#config_version=5

game:
    balls_per_game: 1

modes:
    - mode1

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:
    eject_coil3:
        number:
    eject_coil4:
        number:

switches:
    s_start:
        number:
        tags: start
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch_launcher:
        number:
    s_ball_switch_hold1:
        number:
    s_ball_switch_hold2:
        number:
    s_ball_switch_hold3:
        number:
    s_ball_switch_hold4:
        number:
    s_ball_switch_hold5:
        number:
```

(continues on next page)

(continued from previous page)

```
    s_playfield_active:
        tags: playfield_active
        number:

playfields:
    playfield:
        default_source_device: test_launcher
        tags: default

ball_devices:
    test_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2
        debug: true
        confirm_eject_type: target
        eject_targets: test_launcher
        tags: trough, drain, home
    test_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
        debug: true
        confirm_eject_type: target
        eject_timeouts: 6s, 10s
    test_hold:
        eject_coil: eject_coil3
        ball_switches: s_ball_switch_hold1, s_ball_switch_hold2, s_ball_switch_hold3
        confirm_eject_type: target
        debug: true
    test_hold2:
        eject_coil: eject_coil4
        ball_switches: s_ball_switch_hold4, s_ball_switch_hold5
        confirm_eject_type: target
        debug: true

ball_holds:
    hold_test:
        hold_devices: test_hold
        balls_to_hold: 2
        release_one_events: release_test
    hold_test3:
        hold_devices: test_hold2

event_player:
    test_conditional_event.1{device.ball_holds.hold_test["balls_held"] > 0}:
        - "yes"
    test_conditional_event.2{device.ball_holds.hold_test["balls_held"] == 0}:
        - "no"
    test_event_when_enabled:
        - should_post_when_enabled{device.ball_holds.hold_test.enabled}
        - should_not_post_when_enabled{not device.ball_holds.hold_test.enabled}
    test_event_when_disabled:
        - should_post_when_disabled{not device.ball_holds.hold_test.enabled}
        - should_not_post_when_disabled{device.ball_holds.hold_test.enabled}
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 70: your_machine_folder/ball_holds/modes/mode1/config/mode1.yaml

```
#config_version=5
mode:
  start_events: start_mode1
  stop_events: stop_mode1
  game_mode: False

ball_holds:
    hold_test2:
        hold_devices: test_hold
        balls_to_hold: 2
        release_one_events: release_test
        tags:
        label:
```

ball_routing (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 71: your_machine_folder/ball_routing/config/config.yaml

```
#config_version=5
modes:
    - mode1

playfields:
    playfield:
        default_source_device: test_trough
        tags: default

coils:
    c_trough:
        number:
    c_device1:
        number:
    c_device2:
        number:
    c_device3:
        number:
    c_device4:
        number:

switches:
    s_ball_switch1:
```

(continues on next page)

(continued from previous page)

```
        number:
    s_ball_switch2:
        number:
    s_device1:
        number:
    s_device2:
        number:
    s_device3:
        number:
    s_device4:
        number:

ball_devices:
    test_trough:
        eject_coil: c_trough
        ball_switches: s_ball_switch1, s_ball_switch2
        eject_targets: playfield
        tags: trough, drain, home
    test_device1:
        eject_coil: c_device1
        ball_switches: s_device1
        debug: True
        eject_targets: test_device2, test_device3
    test_device2:
        eject_coil: c_device2
        ball_switches: s_device2
        debug: True
        eject_targets: playfield
    test_device3:
        eject_coil: c_device3
        ball_switches: s_device3
        debug: True
        eject_targets: test_device4
    test_device4:
        eject_coil: c_device4
        ball_switches: s_device4
        debug: True
        eject_targets: playfield
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 72: your_machine_folder/ball_routing/modes/mode1/config/mode1.yaml

```
#config_version=5
ball_routings:
    route_to_ball_device2:
      source_devices: test_device1
      target_device: test_device2
      debug: True
      enable_events: route_to_2
```

(continues on next page)

(continued from previous page)

```
      disable_events: route_to_4, no_route
    route_to_ball_device4:
      source_devices: test_device1
      target_device: test_device4
      debug: True
      enable_events: route_to_4
      disable_events: route_to_2, no_route


multiball_locks:
   lock1:
      balls_to_lock: 1
      lock_devices: test_device4
      enable_events: lock_enable
      disable_events: lock_disable
      debug: true
```

ball_save (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 73: your_machine_folder/ball_save/config/config.yaml

```
#config_version=5

game:
    balls_per_game: 1

modes:
   - mode1
   - mode2

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:

switches:
    s_start:
        number:
        tags: start
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch_launcher:
        number:
```

(continues on next page)

(continued from previous page)

```
    s_left_outlane:
        number:

playfields:
    playfield:
        default_source_device: bd_launcher
        tags: default

ball_devices:
    bd_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2
        debug: true
        confirm_eject_type: target
        eject_targets: bd_launcher
        tags: trough, drain, home
    bd_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
        debug: true
        confirm_eject_type: target
        eject_timeouts: 2s

ball_saves:
    default:
        active_time: 10s
        hurry_up_time: 2s
        grace_period: 2s
        enable_events: enable1
        timer_start_events: balldevice_bd_launcher_ball_eject_success
        early_ball_save_events: s_left_outlane_active
        auto_launch: yes
        balls_to_save: 1
        debug: yes
    unlimited:
        active_time: 30s
        hurry_up_time: 2s
        grace_period: 2s
        enable_events: enable2
        early_ball_save_events: s_left_outlane_active
        auto_launch: yes
        balls_to_save: -1
        debug: yes
    only_last:
        enable_events: enable3
        only_last_ball: True
        debug: yes
    eject_delay:
        enable_events: enable4
        eject_delay: 1s
        debug: yes
    unlimited_delay:
        enable_events: enable5
        delayed_eject_events: eject5
```

(continues on next page)

(continued from previous page)

```
    dynamic_active_time:
        active_time: current_player.save_time
        enable_events: enable6
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 74: your_machine_folder/ball_save/modes/mode2/config/mode2.yaml

```
#config_version=5
mode:
  start_events: start_mode2
  stop_events: stop_mode2

ball_saves:
    mode_ball_save_delayed:
        balls_to_save: -1
        debug: yes
        delayed_eject_events: mode_ball_save_delayed_eject
```

Listing 75: your_machine_folder/ball_save/modes/mode1/config/mode1.yaml

```
#config_version=5
mode:
  start_events: start_mode1
  stop_events: stop_mode1
  game_mode: False

ball_saves:
    mode_ball_save:
        active_time: 10s
        hurry_up_time: 2s
        grace_period: 2s
        timer_start_events: balldevice_bd_launcher_ball_eject_success
        auto_launch: yes
        balls_to_save: 1
        debug: yes
```

ball_search (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 76: your_machine_folder/ball_search/config/no_eject.yaml

```
#config_version=5

game:
    balls_per_game: 3

machine:
    min_balls: 1

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:
    eject_coil3:
        number:
    hold_coil:
        number:
    drop_target_reset1:
        number:
    drop_target_reset2:
        number:
    drop_target_knockdown2:
        number:
    drop_target_reset3:
        number:
    drop_target_reset4:
        number:
    drop_target_knockdown4:
        number:
    flipper_coil:
        number:
        default_hold_power: 0.125
    diverter_coil:
        number:
        default_hold_power: 0.250
    autofire_coil:
        number:

digital_outputs:
    c_motor_run:
        number:
        type: driver

playfields:
    playfield:
        enable_ball_search: True
        ball_search_timeout: 20s
        ball_search_wait_after_iteration: 10s
        ball_search_interval: 250ms
        default_source_device: test_launcher

servos:
```

(continues on next page)

(continued from previous page)

```
    servo1:
        number:
        reset_events:

motors:
    motor1:
        motor_left_output: c_motor_run
        position_switches:  !!omap
            - up: s_position_up
            - down: s_position_down
        reset_position: down

switches:
    s_start:
        number:
        tags: start
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch3:
        number:
    s_ball_switch4:
        number:
    s_ball_switch_launcher:
        number:
    s_vuk:
        number:
    s_lock:
        number:
    s_playfield:
        number:
        tags: playfield_active
    s_drop_target1:
        number:
    s_drop_target2:
        number:
    s_drop_target3:
        number:
    s_drop_target4:
        number:
    s_autofire:
        number:
    s_flipper:
        number:
    s_position_up:
        number:
    s_position_down:
        number:

drop_targets:
    target1:
        reset_coil: drop_target_reset1
        switch: s_drop_target1
```

(continues on next page)

(continued from previous page)

```
        ball_search_order: 10
    target2:
        reset_coil: drop_target_reset2
        knockdown_coil: drop_target_knockdown2
        switch: s_drop_target2
        ball_search_order: 11
    target3:
        reset_coil: drop_target_reset3
        switch: s_drop_target3
        ball_search_order: 12
    target4:
        reset_coil: drop_target_reset4
        knockdown_coil: drop_target_knockdown4
        switch: s_drop_target4
        ball_search_order: 13

ball_devices:
    test_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2, s_ball_switch3, s_ball_switch4
        debug: true
        eject_targets: test_launcher
        tags: trough, drain, home
        ball_search_order: 1
    test_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
        eject_timeouts: 5s
        eject_coil_jam_pulse: 5ms
        debug: true
        ball_search_order: 2
        tags: no-eject-on-ballsearch
    test_vuk:
        eject_coil: eject_coil3
        ball_switches: s_vuk
        eject_timeouts: 2s
        debug: true
        ball_search_order: 3
    test_lock:
        hold_coil: hold_coil
        ball_switches: s_lock
        eject_timeouts: 2s
        debug: true
        ball_search_order: 4

diverters:
    diverter1:
        activation_coil: diverter_coil
        ball_search_order: 14

flippers:
    flipper1:
        main_coil: flipper_coil
        activation_switch: s_flipper
```

(continues on next page)

(continued from previous page)

```
        ball_search_order: 15
        include_in_ball_search: True

autofire_coils:
    autofire1:
        coil: autofire_coil
        switch: s_autofire
        ball_search_order: 16
```

Listing 77: your_machine_folder/ball_search/config/config_ball_device.yaml

```
#config_version=5

playfields:
    playfield:
        enable_ball_search: True
        ball_search_timeout: 20s
        ball_search_wait_after_iteration: 10s
        ball_search_interval: 250ms
        default_source_device: bd_test

switches:
  s_test:
    number:

coils:
  c_test:
    number:

ball_devices:
  bd_test:
    ball_switches: s_test
    eject_coil: c_test
    tags: trough, home, drain
    eject_timeouts: 1s
```

Listing 78: your_machine_folder/ball_search/config/config.yaml

```
#config_version=5

game:
    balls_per_game: 3

machine:
    min_balls: 1

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:
    eject_coil3:
        number:
```

(continues on next page)

(continued from previous page)

```
    hold_coil:
        number:
    drop_target_reset1:
        number:
    drop_target_reset2:
        number:
    drop_target_knockdown2:
        number:
    drop_target_reset3:
        number:
    drop_target_reset4:
        number:
    drop_target_knockdown4:
        number:
    flipper_coil:
        number:
        default_hold_power: 0.125
    diverter_coil:
        number:
        default_hold_power: 0.250
    autofire_coil:
        number:

digital_outputs:
    c_motor_run:
        number:
        type: driver

playfields:
    playfield:
        enable_ball_search: True
        ball_search_timeout: 20s
        ball_search_wait_after_iteration: 10s
        ball_search_interval: 250ms
        default_source_device: test_launcher

servos:
    servo1:
        number:
        reset_events:

motors:
    motor1:
        motor_left_output: c_motor_run
        position_switches:  !!omap
            - up: s_position_up
            - down: s_position_down
        reset_position: down

switches:
    s_start:
        number:
        tags: start
    s_ball_switch1:
```

(continues on next page)

(continued from previous page)

```
        number:
    s_ball_switch2:
        number:
    s_ball_switch3:
        number:
    s_ball_switch4:
        number:
    s_ball_switch_launcher:
        number:
    s_vuk:
        number:
    s_lock:
        number:
    s_playfield:
        number:
        tags: playfield_active
    s_drop_target1:
        number:
    s_drop_target2:
        number:
    s_drop_target3:
        number:
    s_drop_target4:
        number:
    s_autofire:
        number:
    s_flipper:
        number:
    s_position_up:
        number:
    s_position_down:
        number:

drop_targets:
    target1:
        reset_coil: drop_target_reset1
        switch: s_drop_target1
        ball_search_order: 10
    target2:
        reset_coil: drop_target_reset2
        knockdown_coil: drop_target_knockdown2
        switch: s_drop_target2
        ball_search_order: 11
    target3:
        reset_coil: drop_target_reset3
        switch: s_drop_target3
        ball_search_order: 12
    target4:
        reset_coil: drop_target_reset4
        knockdown_coil: drop_target_knockdown4
        switch: s_drop_target4
        ball_search_order: 13

ball_devices:
```

(continues on next page)

(continued from previous page)

```
    test_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2, s_ball_switch3, s_ball_switch4
        debug: true
        eject_targets: test_launcher
        tags: trough, drain, home
        ball_search_order: 1
    test_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
        eject_timeouts: 5s
        eject_coil_jam_pulse: 5ms
        debug: true
        ball_search_order: 2
    test_vuk:
        eject_coil: eject_coil3
        ball_switches: s_vuk
        eject_timeouts: 2s
        debug: true
        ball_search_order: 3
    test_lock:
        hold_coil: hold_coil
        ball_switches: s_lock
        eject_timeouts: 2s
        debug: true
        ball_search_order: 4

diverters:
    diverter1:
        activation_coil: diverter_coil
        ball_search_order: 14

flippers:
    flipper1:
        main_coil: flipper_coil
        activation_switch: s_flipper
        ball_search_order: 15
        include_in_ball_search: True

autofire_coils:
    autofire1:
        coil: autofire_coil
        switch: s_autofire
        ball_search_order: 16
```

Listing 79: your_machine_folder/ball_search/config/config_with_balls.yaml

```
#config_version=5

config:
    - config.yaml

virtual_platform_start_active_switches:
    - s_ball_switch1
```

(continues on next page)

(continued from previous page)

```
    - s_ball_switch2
```

Listing 80: your_machine_folder/ball_search/config/missing_initial.yaml

```
#config_version=5

machine:
    balls_installed: 3

config:
  - config.yaml

virtual_platform_start_active_switches:
  - s_ball_switch1
```

Listing 81: your_machine_folder/ball_search/config/mechanical_eject.yaml

```
#config_version=5

game:
    balls_per_game: 3

machine:
    min_balls: 1

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:

playfields:
    playfield:
        enable_ball_search: True
        ball_search_timeout: 20s
        ball_search_wait_after_iteration: 10s
        ball_search_interval: 250ms
        default_source_device: test_launcher

switches:
    s_start:
        number:
        tags: start
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch3:
        number:
    s_ball_switch4:
        number:
    s_ball_switch_launcher:
        number:
```

(continues on next page)

(continued from previous page)

```
ball_devices:
    test_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2, s_ball_switch3, s_ball_switch4
        debug: true
        eject_targets: test_launcher
        tags: trough, drain, home
        ball_search_order: 1
    test_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
        mechanical_eject: True
        debug: true
        ball_search_order: 2
        auto_fire_on_unexpected_ball: False
```

bcp (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 82: your_machine_folder/bcp/config/config.yaml

```
#config_version=5

modes:
  - mode1
  - mode2

switches:
    s_start:
        number: 1002
        tags: start
    s_test:
        number: 1000
    s_test2:
        number: 1001
    s_ball_switch_launcher:
        number: 1005
        label: Launcher
    s_ball_switch1:
        number: 1003
        label: Ball One
    s_ball_switch2:
        number: 1004
```

(continues on next page)

(continued from previous page)

```
        label: Ball Two

game:
    balls_per_game: 3

coils:
    eject_coil2:
        number: 1001
    eject_coil1:
        number: 1000

lights:
    l_test2:
        number: 1001
        label: Other Light
    l_test:
        number: 1000
        label: Light One

playfields:
    playfield:
        default_source_device: bd_launcher
        tags: default

ball_devices:
    bd_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2
        debug: true
        confirm_eject_type: target
        eject_targets: bd_launcher
        tags: trough, drain, home
    bd_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
        debug: true
        confirm_eject_type: target
        eject_timeouts: 2s

event_player:
    send_test_trigger: trigger_test
```

Listing 83: your_machine_folder/bcp/config/multiple_connections_config.yaml

```
#config_version=5

bcp:
    debug: True
    connections:
        local_display:
            host: localhost
            port: 5050
            type: mpf.core.bcp.bcp_socket_client.BCPClientSocket
            required: True
```

(continues on next page)

(continued from previous page)

```
            exit_on_close: True
        another_display:
            host: localhost
            port: 9001
            type: mpf.core.bcp.bcp_socket_client.BCPClientSocket
            required: True
            exit_on_close: True
```

Listing 84: your_machine_folder/bcp/config/test_bcp_processor.yaml

```
#config_version=5
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 85: your_machine_folder/bcp/modes/mode2/config/mode2.yaml

```
#config_version=5

mode:
  start_events: start_mode2
  stop_events: stop_mode2
  game_mode: False
```

Listing 86: your_machine_folder/bcp/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  start_events: start_mode1
  stop_events: stop_mode1
  game_mode: False
  priority: 200
```

bitmap_fonts (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 87: your_machine_folder/bitmap_fonts/config/test_bitmap_fonts.yaml

```
#config_version=5
```

(continues on next page)

(continued from previous page)

```
displays:
  default:
    width: 800
    height: 600

slides:
  static_text:
    - type: text
      text: TEST
      font_name: F1fuv
      bitmap_font: True
      animations:
        add_to_slide:
          - property: rotation
            value: 360
            duration: 2s
          - property: scale
            value: 0.01
            duration: 1s
    - type: text
      text: STATIC TEXT
      font_name: test_font
      bitmap_font: True
      y: 200
    - type: text
      text: Bitmap Font Test @!$
      font_name: test_font_2
      bitmap_font: True
      y: top - 100
      opacity: 0
      animations:
        add_to_slide:
          - property: opacity
            value: 1.0
            duration: 1s

bitmap_fonts:
  F1fuv:
    file: F1fuv.png
    descriptor: [ ' !"#$%&,()*+`-./', '0123456789:;<=>?', '@ABCDEFGHIJKLMNO', 'PQRSTUVWXYZ[\]^_', ''
↪'abcdefghijklmno', 'pqrstuvwxyz{|}~ ']

slide_player:
  static_text: static_text
```

blinkenlight (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 88: your_machine_folder/blinkenlight/config/config.yaml

```yaml
#config_version=5

modes:
   - mode1
   - mode2

lights:
    l_light1:
        channels:
            red:
                number: 1
            green:
                number: 2
            blue:
                number: 3
    l_light2:
        channels:
            red:
                number: 4
            green:
                number: 5
            blue:
                number: 6
    l_light3:
        channels:
            red:
                number: 7
            green:
                number: 8
            blue:
                number: 9

blinkenlights:
  my_blinkenlight1:
    cycle_duration: 3s
    off_when_multiple: true
    light: l_light1
  my_blinkenlight2:
    color_duration: 2s
    off_when_multiple: false
    light: l_light2
  my_blinkenlight3:
    cycle_duration: 2s
    off_when_multiple: false
    light: l_light3
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 89: your_machine_folder/blinkenlight/modes/mode2/config/mode2.yaml

```
#config_version=5

mode:
  start_events: start_mode2
  stop_events: stop_mode2
  game_mode: False
  priority: 101

blinkenlight_player:
  mode2_add_color_to_first_blinkenlight:
    my_blinkenlight1: orange
  mode2_add_color2_to_first_blinkenlight:
    my_blinkenlight1:
      color: turquoise
      key: second
  mode2_remove_mode_colors_from_first_blinkenlight:
    my_blinkenlight1:
      action: remove_mode
  mode2_add_color_to_second_blinkenlight:
    my_blinkenlight2: magenta

shows:
  blinkenlight_token_show:
    - time: 0
      blinkenlights:
        (blinkenlight_token): (color_token)
    - time: 10

show_player:
  play_blinkenlight_token_show:
    blinkenlight_token_show:
      show_tokens:
        blinkenlight_token: my_blinkenlight2
        color_token: gray
```

Listing 90: your_machine_folder/blinkenlight/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  start_events: start_mode1
  stop_events: stop_mode1
  game_mode: False
  priority: 100

blinkenlight_player:
  add_color_to_all_blinkenlights:
    my_blinkenlight1:
      action: add
      key: mykey1
      color: blue
    my_blinkenlight2: green
```

(continues on next page)

(continued from previous page)

```
    my_blinkenlight3: cyan
  add_color_to_first_blinkenlight:
    my_blinkenlight1: red
  add_color_to_second_blinkenlight:
    my_blinkenlight2:
      action: add
      key: mykey3
      color: yellow
  add_color_to_third_blinkenlight:
    my_blinkenlight3: purple
  remove_color_from_first_blinkenlight:
    my_blinkenlight1:
      action: remove
      key: mykey1
  remove_color_from_second_blinkenlight:
    my_blinkenlight2:
      action: remove
      key: mykey3
  remove_color_from_third_blinkenlight:
    my_blinkenlight3: stop
  remove_all_colors_from_all_blinkenlights:
    my_blinkenlight1:
      action: remove_all
    my_blinkenlight2:
      action: remove_all
    my_blinkenlight3:
      action: remove_all
  add_color_to_first_blinkenlight_with_duplicate_key:
    my_blinkenlight1:
      action: add
      color: darkred
      key: mykey1
```

blocking_events (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 91: your_machine_folder/blocking_events/config/config.yaml

```
#config_version=5

modes:
  - mode1
  - mode2
  - mode3
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 92: your_machine_folder/blocking_events/modes/mode2/config/mode2.yaml

```
#config_version=5

mode:
  priority: 200
  game_mode: False

blocking:
  event1:
    all: True
```

Listing 93: your_machine_folder/blocking_events/modes/mode3/config/mode3.yaml

```
#config_version=5

mode:
  priority: 300
  game_mode: False

blocking:
  event1:
    block: 1
```

Listing 94: your_machine_folder/blocking_events/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  priority: 100
  game_mode: False

blocking:
  event1:
    all: True
```

bonus (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 95: your_machine_folder/bonus/config/config.yaml

```
#config_version=5

modes:
    - bonus
    - mode1
    - service
    - tilt

machine:
    min_balls: 0

game:
  balls_per_game: 10  # we have a lot of bonus tests to run :)

switches:
    s_start:
        number:
        tags: start
    s_slam_tilt:
        number:
        tags: slam_tilt
    s_door_open:
        number: 1
        tags: service_door_open, power_off
    s_service_enter:
        number: 17
        tags: service_enter
    s_service_esc:
        number: 18
        tags: service_esc
    s_service_up:
        number: 19
        tags: service_up

player_vars:
    bonus_multiplier:
        initial_value: 1
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 96: your_machine_folder/bonus/modes/bonus/config/bonus.yaml

```
#config_version=5

mode_settings:
  keep_multiplier: True
```

(continues on next page)

(continued from previous page)

```
  bonus_entries:
      - event: bonus_ramps
        score: 1000
        player_score_entry: ramps
        reset_player_score_entry: True
        skip_if_zero: false
      - event: bonus_modes
        score: 5000
        player_score_entry: modes
        reset_player_score_entry: False
```

Listing 97: your_machine_folder/bonus/modes/mode1/config/mode1.yaml

```
#config_version=5
mode:
    start_events: start_mode1
    stop_events: stop_mode1
    priority: 200

variable_player:
    hit_target:
        score: 1337
    score_ramps:
        ramps: 1
    score_modes:
        modes: 1
    add_multiplier:
        bonus_multiplier: 1
```

bonus_additional_events (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 98: your_machine_folder/bonus_additional_events/config/config.yaml

```
#config_version=5

modes:
    - bonus
    - mode1

machine:
    min_balls: 0

game:
  balls_per_game: 10  # we have a lot of bonus tests to run :)

switches:
```

(continues on next page)

(continued from previous page)

```
    s_start:
        number:
        tags: start

player_vars:
    bonus_multiplier:
        initial_value: 1
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 99: your_machine_folder/bonus_additional_events/modes/bonus/config/bonus.yaml

```
#config_version=5

mode_settings:
  keep_multiplier: True
  end_bonus_event: stop_bonus
  bonus_entries:
      - event: bonus_ramps
        score: 1000
        player_score_entry: ramps
        reset_player_score_entry: True
        skip_if_zero: false
      - event: bonus_modes
        score: 5000
        player_score_entry: modes
        reset_player_score_entry: False
```

Listing 100: your_machine_folder/bonus_additional_events/modes/mode1/config/mode1.yaml

```
#config_version=5
mode:
    start_events: start_mode1
    stop_events: stop_mode1
    priority: 200

variable_player:
    hit_target:
        score: 1337
    score_ramps:
        ramps: 1
    score_modes:
        modes: 1
    add_multiplier:
        bonus_multiplier: 1
```

bonus_dynamic_keep_multiplier (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 101: your_machine_folder/bonus_dynamic_keep_multiplier/config/config.yaml

```
#config_version=5

modes:
    - bonus
    - mode1

machine:
    min_balls: 0

game:
  balls_per_game: 10  # we have a lot of bonus tests to run :)

switches:
    s_start:
        number:
        tags: start

player_vars:
    bonus_multiplier:
        initial_value: 1
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 102: your_machine_folder/bonus_dynamic_keep_multiplier/modes/bonus/config/bonus.yaml

```
#config_version=5

mode_settings:
  keep_multiplier: current_player.ball == 1
  bonus_entries:
      - event: bonus_ramps
        score: 1000
        player_score_entry: ramps
        reset_player_score_entry: True
        skip_if_zero: false
      - event: bonus_modes
        score: 5000
        player_score_entry: modes
        reset_player_score_entry: False
```

Listing 103: your_machine_folder/bonus_dynamic_keep_multiplier/modes/mode1/config/mode1.yaml

```
#config_version=5
mode:
    start_events: start_mode1
    stop_events: stop_mode1
    priority: 200

variable_player:
    hit_target:
        score: 1337
    score_ramps:
        ramps: 1
    score_modes:
        modes: 1
    add_multiplier:
        bonus_multiplier: 1
```

bonus_no_keep_multiplier (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 104: your_machine_folder/bonus_no_keep_multiplier/config/config.yaml

```
#config_version=5

modes:
    - bonus
    - mode1

machine:
    min_balls: 0

game:
  balls_per_game: 10  # we have a lot of bonus tests to run :)

switches:
    s_start:
        number:
        tags: start

player_vars:
    bonus_multiplier:
        initial_value: 1
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 105: your_machine_folder/bonus_no_keep_multiplier/modes/bonus/config/bonus.yaml

```
#config_version=5

mode_settings:
  bonus_entries:
      - event: bonus_ramps
        score: 1000
        player_score_entry: ramps
        reset_player_score_entry: True
        skip_if_zero: True
      - event: bonus_modes
        score: 5000
        player_score_entry: modes
        reset_player_score_entry: False
```

Listing 106: your_machine_folder/bonus_no_keep_multiplier/modes/mode1/config/mode1.yaml

```
#config_version=5
mode:
    start_events: start_mode1
    stop_events: stop_mode1
    priority: 200

variable_player:
    hit_target:
        score: 1337
    score_ramps:
        ramps: 1
    score_modes:
        modes: 1
    add_multiplier:
        bonus_multiplier: 1
```

carousel (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 107: your_machine_folder/carousel/config/config.yaml

```
#config_version=5

modes:
    - carousel
    - second_carousel
    - conditional_carousel
```

(continues on next page)

(continued from previous page)

```
    - blocking_carousel

machine:
    min_balls: 0

switches:
    s_start:
        number:
        tags: start
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 108: your_machine_folder/carousel/modes/conditional_carousel/config/conditional_carousel

```
#config_version=5
mode:
  start_events: start_mode3
  stop_events: stop_mode3, conditional_carousel_item_selected
  code: mpf.modes.carousel.code.carousel.Carousel

mode_settings:
  selectable_items:
    - item1{not current_player.hide_item1}
    - item2{False}
    - item3{machine.player2_score}
    - item4{current_player.show_item4}
  select_item_events: select
  next_item_events: next
  previous_item_events: previous
```

Listing 109: your_machine_folder/carousel/modes/second_carousel/config/second_carousel.yaml

```
#config_version=5
mode:
  start_events: start_mode1
  stop_events: stop_mode1, carousel_item_selected
  code: mpf.modes.carousel.code.carousel.Carousel

mode_settings:
  selectable_items: item1, item2, item3
  select_item_events: select, select_additional
  next_item_events: next, next2
  previous_item_events: previous, previous2
```

Listing 110: your_machine_folder/carousel/modes/carousel/config/carousel.yaml

```
#config_version=5
mode:
  start_events: start_mode1
  stop_events: stop_mode1, carousel_item_selected
  code: mpf.modes.carousel.code.carousel.Carousel

mode_settings:
  selectable_items: item1, item2, item3
  select_item_events: select, select_additional
  next_item_events: next, next2
  previous_item_events: previous, previous2
```

Listing 111: your_machine_folder/carousel/modes/blocking_carousel/config/blocking_carousel.yaml

```
#config_version=5
mode:
  start_events: start_mode4
  stop_events: stop_mode4, carousel_item_selected
  code: mpf.modes.carousel.code.carousel.Carousel

mode_settings:
  selectable_items: item1, item2, item3
  select_item_events: select_item
  next_item_events: s_flipper_right_inactive
  previous_item_events: s_flipper_left_inactive
  block_events: flipper_cancel
  release_events: both_flippers_inactive
```

coil_player (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 112: your_machine_folder/coil_player/config/coil_player.yaml

```
#config_version=5

modes:
  - mode1

coils:
  coil_1:
    number:
    default_hold_power: 1.0
  coil_2:
    number:
  coil_3:
```

(continues on next page)

(continued from previous page)

```
    number:
    default_hold_power: 1.0

coil_player:
  event1: coil_1
  event2:
    coil_1:
      action: pulse
      pulse_power: 1.0
    coil_2:
      action: pulse
      pulse_power: 0.5
  event3:
    coil_1:
      action: pulse
      pulse_ms: 49
  event4:
    coil_1:
      action: enable
  event5:
    coil_1:
      action: disable
  event6: coil_2
  event7:
    coil_3:
      action: enable
      hold_power: 0.5
  event8:
    coil_3: disable
  event9:
    coil_3: 30
  event10:
    coil_1:
      action: on
  event11:
    coil_1:
      action: off
  pulse_1_100:
    coil_1:
      action: pulse
      pulse_ms: 100
  pulse_1_50_max_wait_ms:
    coil_1:
      action: pulse
      pulse_ms: 50
      max_wait_ms: 100
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 113: your_machine_folder/coil_player/modes/mode1/config/mode1.yaml

```
#config_version=5
mode:
  start_events: start_mode1
  stop_events: stop_mode1
  game_mode: False

coil_player:
  event1_mode:
    coil_3: enable
```

color (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 114: your_machine_folder/color/config/test_color.yaml

```
#config_version=5

displays:
  default:
    width: 400
    height: 300

slides:
  slide1:
    - type: text
      text: RED
      color: red
      y: 75
    - type: text
      text: 0000FF80
      color: 0000ff80
    - type: text
      text: 00FF00
      color: 00ff00


slide_player:
  slide1: slide1
```

combo_switches (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 115: your_machine_folder/combo_switches/config/combo_switches.yaml

```
#config_version=5

modes:
  - mode1

switches:
  switch1:
    number:
  switch2:
    number:
  switch3:
    number:
  switch4:
    number:
  switch5:
    number:
    tags: tag1
  switch6:
    number:
    tags: tag1
  switch7:
    number:
    tags: tag2
  switch8:
    number:
    tags: tag2
  switch9:
    number:
    tags: left_flipper
  switch10:
    number:
    tags: right_flipper

combo_switches:
  tag_combo:
    tag_1: tag1
    tag_2: tag2
  switch_combo:
    switches_1: switch1
    switches_2: switch2
  multiple_switch_combo:
    switches_1: switch1, switch2
    switches_2: switch3, switch4
  custom_offset:
    switches_1: switch1
    switches_2: switch2
    max_offset_time: 1s
  custom_hold:
```

(continues on next page)

(continued from previous page)

```
    switches_1: switch1
    switches_2: switch2
    hold_time: 1s
  custom_release:
    switches_1: switch1
    switches_2: switch2
    release_time: 1s
  custom_times_multiple_switches:
    tag_1: tag1
    tag_2: tag2
    max_offset_time: 1s
    hold_time: 1s
    release_time: 1s
    debug: true
  custom_events:
    switches_1: switch1
    switches_2: switch2
    events_when_both: active_event, active_event2
    events_when_inactive: inactive_event
    events_when_one: one_event
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 116: your_machine_folder/combo_switches/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  priority: 100
  game_mode: no

combo_switches:
  mode1_combo:
    switches_1: switch1
    switches_2: switch2
```

config_errors (example config files)

config_interface (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 117: your_machine_folder/config_interface/config/test_config_interface_missing_version.yaml

```
game:
    balls_per_game: 1
```

Listing 118: your_machine_folder/config_interface/config/test_config_interface_old_version.yaml

```
#config_version=2

game:
    balls_per_game: 1
```

Listing 119: your_machine_folder/config_interface/config/test_config_interface.yaml

```
#config_version=5

game:
    balls_per_game: 1

test_section:
  true_key1: true
  true_key2: True
  true_key3: yes
  true_key4: Yes
  false_key1: false
  false_key2: False
  false_key3: no
  false_key4: No
  on_string: on
  off_string: off
  int_6400: 6400
  str_001: 001
  int_100: 100
  int_6: 6
  int_7: 07
  str_00ff00: 00ff00
  str_003200: 003200
  str_plus5: +5
  str_plus0point5: +0.5
  case_sensitive_1: test
  Case_sensitive_2: test
  case_sensitive_3: Test


# test an empty collection
switches:
```

config_loader (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 120: your_machine_folder/config_loader/config/config.yaml

```
#config_version=5
modes:
  - mode1
  - mode2

show_player:
  event4: show1


shows:
  test_show:
    - duration: 1
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 121: your_machine_folder/config_loader/modes/mode2/config/mode2.yaml

```
#config_version=5
mode:
  priority: 200
  game_mode: False
```

Listing 122: your_machine_folder/config_loader/modes/game/config/game.yaml

```
#config_version=5

event_player:
  test_event: another_event

shows:
  game_show:
    - duration: 1
```

Listing 123: your_machine_folder/config_loader/modes/mode1/shows/mode1_show.yaml

```
#show_version=5

- time: 0
  bananas:
    banana2: express
- time: 2
```

Listing 124: your_machine_folder/config_loader/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  priority: 100
  game_mode: False

show_player:
  event6: mode1_show
```

Show file examples

Here are some example show files that go along with the above config(s).

Listing 125: your_machine_folder/config_loader/shows/show1.yaml

```
#show_version=5

- time: 0
  bananas:
    banana1: express
- time: 2
```

config_players (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 126: your_machine_folder/config_players/config/test_config_players.yaml

```
#config_version=5
modes:
  - mode1
  - mode2

banana_player:
  event1: express
  event2:
    some: key
  event3:
    this_banana:
      some: key
    that_banana:
      some: key

show_player:
  event4: show1
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 127: your_machine_folder/config_players/modes/mode2/config/mode2.yaml

```
#config_version=5
mode:
  priority: 200
  game_mode: False

banana_player:
```

Listing 128: your_machine_folder/config_players/modes/mode1/shows/mode1_show.yaml

```
#show_version=5

- time: 0
  bananas:
    banana2: express
- time: 2
```

Listing 129: your_machine_folder/config_players/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  priority: 100
  game_mode: False

banana_player:
  event5: express

show_player:
  event6: mode1_show
```

Show file examples

Here are some example show files that go along with the above config(s).

Listing 130: your_machine_folder/config_players/shows/show1.yaml

```
#show_version=5

- time: 0
  bananas:
    banana1: express
- time: 2
```

config_processor (example config files)

counters (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 131: your_machine_folder/counters/config/config.yaml

```
#config_version=5

lights:
  l_chest_matrix_green_2:
    number:
  l_chest_matrix_green_3:
    number:
  l_chest_matrix_green_4:
    number:
  l_chest_matrix_green_5:
    number:

modes:
  - mode1
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 132: your_machine_folder/counters/modes/mode1/config/mode1.yaml

```
#config_version=5
counters:
  my_counter:
    starting_count: 0
    count_complete_value: 5
    count_events: count_up

light_player:
  "{device.counters.my_counter.value > 0}":
    l_chest_matrix_green_5: green
  "{device.counters.my_counter.value > 1}":
    l_chest_matrix_green_4: green
  "{device.counters.my_counter.value > 2}":
    l_chest_matrix_green_3: green
  "{device.counters.my_counter.value > 3}":
    l_chest_matrix_green_2: green
  "{current_player.progress_value > 0}":
    l_chest_matrix_green_5: green
```

credits (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 133: your_machine_folder/credits/config/config_credit_tiers.yaml

```
#config_version=5

modes:
    - credits

machine:
    min_balls: 0

switches:
    s_left_coin:
        number:
    s_right_coin:
        number:
    s_start:
        number:
        tags: start

credits:
  max_credits: 30
  free_play: no
  switches:
    - switch: s_left_coin
      value: .25
    - switch: s_right_coin
      value: 1
  pricing_tiers:
    - price: .5
      credits: 1
    - price: 2
      credits: 5
    - price: 5
      credits: 15
  fractional_credit_expiration_time: 15m
  credit_expiration_time: 2h
  persist_credits_while_off_time: 1h
  free_play_string: FREE PLAY
  credits_string: CREDITS
```

Listing 134: your_machine_folder/credits/config/config.yaml

```
#config_version=5
```

(continues on next page)

(continued from previous page)

```
modes:
    - credits

machine:
    min_balls: 0

switches:
    s_left_coin:
        number:
    s_center_coin:
        number:
    s_right_coin:
        number:
    s_esc:
        number:
    s_start:
        number:
        tags: start

coils:
    c_eject:
        number:

settings:
  replay_score:
    label: Replay Score
    values:
      500000: "500000 (default)"
      1000000: "1000000"
      1500000: "1500000"
    default: 500000
    key_type: int
    sort: 100
  credits_price_one_credit:
    label: Price for one credit
    values:
      .25: "25ct"
      .5: "50ct"
      .75: "75ct"
      1: "1 dollar"
      2: "2 dollar"
    default: .5
    key_type: float
    sort: 500
  credits_price_tier2:
    label: Price for price tier 2
    values:
      .25: "25ct"
      .5: "50ct"
      .75: "75ct"
      1: "1 dollar"
      2: "2 dollar"
      3: "3 dollar"
      4: "4 dollar"
```

(continues on next page)

(continued from previous page)

```
      5: "5 dollar"
    default: 2
    key_type: float
    sort: 510
  credits_credits_tier2:
    label: Number of credits for tier 2
    values:
      2: "2"
      3: "3"
      4: "4"
      5: "5"
      6: "6"
      7: "7"
      8: "8"
      9: "9"
      10: "10"
    default: 5
    key_type: int
    sort: 520

credits:
  max_credits: 12
  free_play: no
  service_credits_switch: s_esc
  switches:
    - switch: s_left_coin
      type: money
      value: .25
      label: Left Quarter
    - switch: s_center_coin
      type: money
      value: .25
      label: Center Quarter
    - switch: s_right_coin
      type: money
      value: 1
      label: Right Dollar
  events:
    - event: game_ending{current_player.score > settings.replay_score}
      type: award
      credits: 1
  pricing_tiers:
    - price: settings.credits_price_one_credit
      credits: 1
    - price: settings.credits_price_tier2
      credits: settings.credits_credits_tier2
  fractional_credit_expiration_time: 15m
  credit_expiration_time: 2h
  persist_credits_while_off_time: 1h
  free_play_string: FREE PLAY
  credits_string: CREDITS
```

Listing 135: your_machine_folder/credits/config/config_freeplay.yaml

```
#config_version=5

modes:
    - credits

machine:
    min_balls: 0

switches:
    s_left_coin:
        number:
    s_center_coin:
        number:
    s_right_coin:
        number:
    s_esc:
        number:
    s_start:
        number:
        tags: start

coils:
    c_eject:
        number:

credits:
  max_credits: 12
  free_play: yes
  service_credits_switch: s_esc
  switches:
    - switch: s_left_coin
      type: money
      value: .25
    - switch: s_center_coin
      type: money
      value: .25
    - switch: s_right_coin
      type: money
      value: 1
  pricing_tiers:
    - price: .50
      credits: 1
    - price: 2
      credits: 5
  fractional_credit_expiration_time: 15m
  credit_expiration_time: 2h
  persist_credits_while_off_time: 1h
  free_play_string: FREE PLAY
  credits_string: CREDITS
```

Listing 136: your_machine_folder/credits/config/config_inhibit.yaml

```
#config_version=5

modes:
    - credits

machine:
    min_balls: 0

switches:
    s_left_coin:
        number:
    s_start:
        number:
        tags: start

digital_outputs:
  c_coin_inhibit:
    type: driver
    number:

credits:
  max_credits: 4
  free_play: no
  switches:
    - switch: s_left_coin
      value: 1
  coin_inhibit_disable_output: c_coin_inhibit
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 137: your_machine_folder/credits/modes/credits/config/config.yaml

```
#config_version=5

mode:
  priority: 11000
  start_events: machine_reset_phase_3
  stop_on_ball_end: False
```

custom_code (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 138: your_machine_folder/custom_code/config/config.yaml

```
#config_version=5

custom_code:
  - code.test_code.TestCustomCode
```

data_manager (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 139: your_machine_folder/data_manager/config/config.yaml

```
#config_version=5

mpf:
    paths:
        absolute_test: /data/test_dir/test_file.yaml
        relative_test: subdir/subdir2/test.yaml
        disabled_test: False
```

device (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 140: your_machine_folder/device/config/config.yaml

```
#config_version=5

lights:
    light_01:
        number: 0
        label: Test 0
        subtype: matrix
        debug: True
    light_02:
        number: 1
        label: Test 1
        subtype: matrix
        debug: True
```

(continues on next page)

(continued from previous page)

```
    gi_01:
        number: 1
        subtype: gi
        debug: True
    gi_02:
        number: 2
        subtype: gi
        debug: True
    flasher_01:
        number: flasher_01
        platform: drivers
        debug: True
    flasher_02:
        number: flasher_02
        platform: drivers
        debug: True
    flasher_03:
        number: flasher_03
        platform: drivers
        debug: True

coils:
    flasher_01:
        number: 4
        label: Test flasher
        default_pulse_ms: 40
        max_hold_power: 1.0
    flasher_02:
        number: 5
        label: Test flasher 2
        default_pulse_ms: 100
        max_hold_power: 1.0
    flasher_03:
        number: 6
        max_hold_power: 1.0

show_player:
   flash2:
      flash_show:
         action: play
         show_tokens:
            flashers: flasher_01, flasher_02

shows:
   flash_show:
     - flashers:
         (flashers): 100ms
       events: test
       duration: 1s

flasher_player:
   flash:
      flasher_01: 100ms
```

Listing 141: your_machine_folder/device/config/config_dual_wound_coil.yaml

```
#config_version=5

coils:
    c_hold:
        number:
        default_hold_power: 1.0
    c_power:
        number:
        default_pulse_ms: 20

switches:
    s_eos:
        number:

dual_wound_coils:
    c_test:
        hold_coil: c_hold
        main_coil: c_power
    c_test_eos:
        hold_coil: c_hold
        main_coil: c_power
        eos_switch: s_eos
```

Listing 142: your_machine_folder/device/config/coils.yaml

```
#config_version=5
coils:
    coil_01:
        number: 1
        default_pulse_ms: 30
        allow_enable: True
    coil_02:
        number: 2
        default_pulse_ms: 60
    coil_03:
        number: 3
    coil_max_hold_duration:
        number: 4
        default_hold_power: 0.5
        max_hold_duration: 5s
    coil_pulse_with_timed_enable:
        number: 5
        default_pulse_ms: 60
        default_pulse_power: 0.25
        default_hold_power: 0.5
        default_timed_enable_ms: 200
        pulse_with_timed_enable: true
```

device_collection (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 143: your_machine_folder/device_collection/config/test_device_collection.yaml

```
#config_version=5

lights:
  led1:
    number: 1
    tags: tag1, tag2
  led2:
    number: 2
    tags: tag1
  led3:
    number: 3
    tags: tag2
  led4:
    number: 4
```

digital_output (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 144: your_machine_folder/digital_output/config/config.yaml

```
#config_version=5

digital_outputs:
    light_output:
        number: 1
        type: light
        light_subtype: test_subtype
    driver_output:
        number: 1
        type: driver
```

digital_score_reels (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 145: your_machine_folder/digital_score_reels/config/test_digital_score_reels.yaml

```
#config_version=5

digital_score_reels:
  player_score:
    reel_count: 4
    include_player_number: true
    frames:
      - character: 1
        frame: 2
      - character: 2
        frame: 4
      - character: 3
        frame: 6
      - character: 4
        frame: 8
      - character: 5
        frame: 10
      - character: 6
        frame: 12
      - character: 7
        frame: 14
      - character: 8
        frame: 16
      - character: 9
        frame: 18
      - character: 0
        frame: 20
  arbitrary_event:
    reel_count: 3
    start_value: X
    frames:
      - character: A
        frame: 1
      - character: B
        frame: 2
      - character: C
        frame: 3
      - character: D
        frame: 4
      - character: E
        frame: 5
      - character: X
        frame: 6
```

display (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 146: your_machine_folder/display/config/test_display_multiple.yaml

```
#config_version=5
displays:
  window:
    width: 401
    height: 301
  display2:
    width: 402
    height: 302
    default: true
```

Listing 147: your_machine_folder/display/config/test_display_single.yaml

```
#config_version=5
displays:
  window:
    width: 401
    height: 301
```

Listing 148: your_machine_folder/display/config/test_display.yaml

```
#config_version=5

window:
  width: 800
  height: 600
  title: Mission Pinball Framework - Demo Man
  resizable: true
  fullscreen: false
  borderless: false
  exit_on_escape: true

displays:
  window:
    width: 600
    height: 200
  dmd:
    width: 128
    height: 32

widget_styles:
  text_default:
    font_name: Quadrit
    font_size: 10
    adjust_top: 2
    adjust_bottom: 3
  medium:
    font_name: pixelmix
    font_size: 8
    adjust_top: 1
    adjust_bottom: 1
```

(continues on next page)

(continued from previous page)

```
  small:
    font_name: smallest_pixel-7
    font_size: 9
    adjust_top: 2
    adjust_bottom: 3
  tall_title:
    font_name: big_noodle_titling
    font_size: 20

slides:
  window_slide_1:
  - type: display
    width: 516
    height: 128
    source_display: dmd
    effects:
      - type: dmd
        dot_color: ff5500
        background_color: 220000
  - type: text
    style: tall_title
    text: MISSION PINBALL FRAMEWORK
    anchor_y: top
    y: top-2
    font_size: 30
    color: white
  - type: rectangle
    width: 518
    height: 130
    color: 444444
  - type: text
    style: tall_title
    text: DEMO MAN
    anchor_x: right
    anchor_y: bottom
    y: bottom+2
    x: right-42
    font_size: 30
    color: red
  asset_status:
  - type: text
    text: "LOADING ASSETS"

slide_player:
  mc_ready.1: window_slide_1
  mc_ready.2:
    asset_status:
      target: dmd
```

Listing 149: your_machine_folder/display/config/test_display_none.yaml

```
#config_version=5
```

diverter (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 150: your_machine_folder/diverter/config/config.yaml

```
#config_version=5

coils:
    eject_coil1:
        number: 1
    eject_coil2:
        number: 2
    c_diverter:
        number: 3
        default_hold_power: 0.250
    c_diverter_disable:
        number: 4

switches:
    s_ball_switch1:
        number: 1
    s_ball_switch2:
        number: 2
    s_ball_switch3:
        number: 100
    s_diverter:
        number: 3
    s_playfield:
        number: 4
        tags: playfield_active
    s_target1:
        number: 5
    s_target2:
        number: 6
    s_target3:
        number: 7

playfields:
    playfield:
        default_source_device: test_target
        tags: default
```

(continues on next page)

(continued from previous page)

```
ball_devices:
    test_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2, s_ball_switch3
        confirm_eject_type: target
        eject_targets: test_target, playfield
        tags: trough, drain, home
    test_target:
        eject_coil: eject_coil2
        ball_switches: s_target1, s_target2, s_target3
        confirm_eject_type: target
        eject_targets: playfield

virtual_platform_start_active_switches:
  - s_ball_switch1
  - s_ball_switch2
  - s_ball_switch3
```

Listing 151: your_machine_folder/diverter/config/test_hold_activation_time.yaml

```
#config_version=5

config:
- config.yaml

diverters:
    d_test_hold_activation_time:
        activation_coil: c_diverter
        activation_switches: s_diverter
        type: hold
        feeder_devices: test_trough
        targets_when_active: playfield
        targets_when_inactive: test_target
        activation_time: 4s
        debug: True
```

Listing 152: your_machine_folder/diverter/config/test_diverter_dual_wound_coil.yaml

```
#config_version=5

config:
- config.yaml

coils:
    c_hold:
        number: 5
    c_power:
        number: 6

dual_wound_coils:
    c_dual_wound:
        hold_coil: c_hold
```

(continues on next page)

(continued from previous page)

```
        main_coil: c_power

diverters:
    d_test_dual_wound:
        activation_coil: c_dual_wound
        activation_switches: s_diverter
        type: hold
        feeder_devices: test_trough
        targets_when_active: playfield
        targets_when_inactive: test_target
        debug: True
```

Listing 153: your_machine_folder/diverter/config/test_diverter_auto_disable.yaml

```
#config_version=5

diverters:
    d_test:
        activation_coil: c_diverter
        type: hold
        debug: True
        activation_switches: s_activate
        disable_switches: s_disable

coils:
    c_diverter:
        number: 10
        default_hold_power: 0.250

switches:
    s_activate:
        number: 1
    s_disable:
        number: 2
```

Listing 154: your_machine_folder/diverter/config/test_missing_ball_at_source.yaml

```
#config_version=5

config:
- config.yaml

diverters:
    d_test:
        activation_coil: c_diverter
        feeder_devices: test_trough
        targets_when_active: playfield
        targets_when_inactive: test_target
        activation_time: 4s
        debug: True
```

Listing 155: your_machine_folder/diverter/config/test_eject_to_oposide_sides.yaml

```
#config_version=5

config:
- config.yaml

diverters:
    d_test_hold:
        activation_coil: c_diverter
        type: hold
        feeder_devices: test_trough, test_trough2
        targets_when_active: playfield
        targets_when_inactive: test_target
        allow_multiple_concurrent_ejects_to_same_side: False
        cool_down_time: 3s
        debug: True

coils:
    eject_coil3:
        number: 10

switches:
    s_ball_switch4:
        number: 10
    s_ball_switch5:
        number: 11

ball_devices:
    test_trough2:
        eject_coil: eject_coil3
        ball_switches: s_ball_switch4, s_ball_switch5
        confirm_eject_type: target
        eject_targets: test_target, playfield
        tags: trough, drain, home

virtual_platform_start_active_switches:
  - s_ball_switch4
  - s_ball_switch5
```

Listing 156: your_machine_folder/diverter/config/diverter_with_activation_events.yaml

```
#config_version=5

coils:
  test_coil:
    number:
    allow_enable: True

diverters:
    test_diverter:
        activation_coil: test_coil
        type: hold
        activate_events: activate_test_diverter
```

(continues on next page)

(continued from previous page)

```
        enable_events: ball_started
        disable_events: ball_ended
        targets_when_active: playfield
        targets_when_inactive: playfield
```

Listing 157: your_machine_folder/diverter/config/test_pulsed_activation_time.yaml

```
#config_version=5

config:
- config.yaml

diverters:
    d_test_pulse:
        activation_coil: c_diverter
        deactivation_coil: c_diverter_disable
        type: pulse
        feeder_devices: test_trough
        targets_when_active: playfield
        targets_when_inactive: test_target
        debug: True
```

Listing 158: your_machine_folder/diverter/config/test_eject_to_oposide_sides2.yaml

```
#config_version=5

config:
- config.yaml

diverters:
    d_test_hold:
        activation_coil: c_diverter
        type: hold
        feeder_devices: test_trough, test_trough2
        targets_when_active: playfield
        targets_when_inactive: test_target
        allow_multiple_concurrent_ejects_to_same_side: False
        cool_down_time: 3s
        debug: True

coils:
    eject_coil3:
        number: 10

switches:
    s_ball_switch4:
        number: 10
    s_ball_switch5:
        number: 11
    s_ball_switch6:
        number: 12

ball_devices:
```

(continues on next page)

(continued from previous page)

```
    test_trough2:
        eject_coil: eject_coil3
        ball_switches: s_ball_switch4, s_ball_switch5, s_ball_switch6
        confirm_eject_type: target
        eject_targets: test_target, playfield
        tags: trough, drain, home

virtual_platform_start_active_switches:
  - s_ball_switch4
  - s_ball_switch5
  - s_ball_switch6
```

Listing 159: your_machine_folder/diverter/config/test_delayed_eject.yaml

```
#config_version=5

config:
- config.yaml

diverters:
    d_test_delayed_eject:
        activation_coil: c_diverter
        type: hold
        feeder_devices: test_trough
        targets_when_active: playfield
        targets_when_inactive: test_target
        activation_time: 4s
        debug: True
```

Listing 160: your_machine_folder/diverter/config/test_hold_no_activation_time.yaml

```
#config_version=5

config:
- config.yaml

diverters:
    d_test_hold:
        activation_coil: c_diverter
        activation_switches: s_diverter
        type: hold
        feeder_devices: test_trough
        targets_when_active: playfield
        targets_when_inactive: test_target
        debug: True
```

Listing 161: your_machine_folder/diverter/config/test_activation_switch_and_eject_confirm_switch.

```
#config_version=5

config:
- config.yaml
```

(continues on next page)

(continued from previous page)

```
diverters:
    d_test_hold_activation_time:
        activation_coil: c_diverter
        activation_switches: s_diverter
        type: hold
        feeder_devices: test_trough2
        targets_when_active: playfield
        targets_when_inactive: test_target
        activation_time: 4s
        debug: True

coils:
    eject_coil3:
        number: 10

switches:
    s_ball_switch10:
        number:
    s_ball_switch11:
        number:
    s_diverter:
        number: 12

ball_devices:
    test_trough2:
        eject_coil: eject_coil3
        ball_switches: s_ball_switch10, s_ball_switch11
        confirm_eject_type: switch
        confirm_eject_switch: s_diverter
        eject_targets: test_target, playfield
        tags: trough, drain, home

virtual_platform_start_active_switches:
  - s_ball_switch10
  - s_ball_switch11
```

Listing 162: your_machine_folder/diverter/config/only_events_no_coils.yaml

```
#config_version=5

config:
- config.yaml

diverters:
    d_test_with_events:
        debug: true
        feeder_devices: test_trough
        targets_when_active: playfield
        targets_when_inactive: test_target

servos:
    s_diverter:
        number:
        positions:
```

(continues on next page)

(continued from previous page)

```
            0.7: diverter_d_test_with_events_activating
            0.2: diverter_d_test_with_events_deactivating

steppers:
    s_diverter:
        number:
        named_positions:
            20: diverter_d_test_with_events_activating
            400: diverter_d_test_with_events_deactivating
```

Listing 163: your_machine_folder/diverter/config/test_diverter_with_switch.yaml

```
#config_version=5

diverters:
    d_test:
        activation_coil: c_diverter
        type: hold
        debug: True
        activation_switches: s_activate
        deactivation_switches: s_deactivate
        disable_switches: s_disable

coils:
    c_diverter:
        number: 10
        default_hold_power: 0.250

switches:
    s_activate:
        number: 1
    s_disable:
        number: 2
    s_deactivate:
        number: 3
```

dmd (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 164: your_machine_folder/dmd/config/testRgbDmd.yaml

```
#config_version=5

rgb_dmds:
```

(continues on next page)

(continued from previous page)

```
  test_dmd:
    label: Test
    hardware_brightness: settings.dmd_brightness

settings:
  dmd_brightness:
    label: DMD Brightness
    values:
      0.1: "10%"
      0.25: "25%"
      0.5: "50%"
      0.75: "75%"
      1.0: "100% (default)"
    default: 1.0
    key_type: float
    sort: 100
```

Listing 165: your_machine_folder/dmd/config/testDmd.yaml

```
#config_version=5

dmds:
  test_dmd:
    label: Test
```

Listing 166: your_machine_folder/dmd/config/test_color_dmd.yaml

```
#config_version=5

displays:
  default:
    width: 800
    height: 600
  dmd:
    width: 128
    height: 32

slides:
  slide1:
    - type: display
      width: 640
      height: 160
      source_display: dmd
      effects:
        - type: color_dmd
    - type: text
      text: COLOR DMD TEST
      y: 200
    - type: rectangle
      width: 642
      height: 162
      color: gray
  slide2:
```

(continues on next page)

(continued from previous page)

```
    - type: display
      y: top - 25
      anchor_y: top
      width: 640
      height: 160
      source_display: dmd
      effects:
        - type: color_dmd
    - type: display
      width: 640
      height: 160
      source_display: dmd
      effects:
        - type: dmd
    - type: display
      y: bottom + 25
      anchor_y: bottom
      width: 640
      height: 160
      source_display: dmd
      effects:
        - type: scanlines
        - type: monochrome
  dmd_slide:
    - type: text
      text: DMD TEXT
      anchor_x: center
      x: 128
      animations:
        show_slide:
          - property: x
            value: 10%
            duration: .25s
          - property: x
            value: 35%
            repeat: true
            duration: 250ms
    - type: rectangle
      width: 8
      height: 32
      color: red
      x: 4
    - type: rectangle
      width: 8
      height: 32
      color: orange
      x: 12
    - type: rectangle
      width: 8
      height: 32
      color: yellow
      x: 20
    - type: rectangle
      width: 8
```

(continues on next page)

(continued from previous page)

```
      height: 32
      color: green
      x: 28
    - type: rectangle
      width: 8
      height: 32
      color: blue
      x: 36
    - type: rectangle
      width: 8
      height: 32
      color: purple
      x: 44
    - type: rectangle
      width: 8
      height: 32
      color: pink
      x: 52
    - type: rectangle
      width: 8
      height: 32
      color: dddddd
      x: 60
    - type: rectangle
      width: 8
      height: 32
      color: bbbbbb
      x: 68
    - type: rectangle
      width: 8
      height: 32
      color: 888888
      x: 76
    - type: rectangle
      width: 8
      height: 32
      color: 666666
      x: 84
    - type: rectangle
      width: 8
      height: 32
      color: 444444
      x: 92
    - type: rectangle
      width: 8
      height: 32
      color: 333333
      x: 100
    - type: rectangle
      width: 8
      height: 32
      color: 222222
      x: 108
    - type: rectangle
```

(continues on next page)

(continued from previous page)

```
      width: 8
      height: 32
      color: 111111
      x: 116
    - type: rectangle
      width: 8
      height: 32
      color: 000000
      x: 124

slide_player:
  slide1: slide1
  slide2: slide2
  dmd_slide:
    dmd_slide:
      target: dmd
```

Listing 167: your_machine_folder/dmd/config/test_dmd.yaml

```
#config_version=5

displays:
  default:
    width: 800
    height: 600
  dmd:
    width: 128
    height: 32
widgets:
  right_dmd_widget:
    type: text
    text: "Right Widget"
    x: right
  left_dmd_widget:
    type: text
    text: "Left Widget"
    x: left
  top_dmd_widget:
    type: text
    text: "Top Widget"
    y: 100%
  bottom_dmd_widget:
    type: text
    text: "Bottom Widget"
    y: 0%
slides:
  container_slide:
    - type: display
      width: 640
      height: 160
      source_display: dmd
      effects:
        - type: dmd
    - type: text
```

(continues on next page)

(continued from previous page)

```
    text: TRADITIONAL DMD TEST
    y: 200
  - type: rectangle
    width: 642
    height: 162
    color: gray
dmd_slide:
  - type: text
    text: DMD TEXT
    anchor_x: center
    x: 128
    animations:
      show_slide:
        - property: x
          value: 10%
          duration: .25s
        - property: x
          value: 35%
          repeat: true
          duration: 250ms
  - type: rectangle
    width: 8
    height: 32
    color: ffffff
    x: 4
  - type: rectangle
    width: 8
    height: 32
    color: eeeeee
    x: 12
  - type: rectangle
    width: 8
    height: 32
    color: dddddd
    x: 20
  - type: rectangle
    width: 8
    height: 32
    color: cccccc
    x: 28
  - type: rectangle
    width: 8
    height: 32
    color: bbbbbb
    x: 36
  - type: rectangle
    width: 8
    height: 32
    color: aaaaaa
    x: 44
  - type: rectangle
    width: 8
    height: 32
    color: 999999
```

(continues on next page)

(continued from previous page)

```
      x: 52
    - type: rectangle
      width: 8
      height: 32
      color: 888888
      x: 60
    - type: rectangle
      width: 8
      height: 32
      color: 777777
      x: 68
    - type: rectangle
      width: 8
      height: 32
      color: 666666
      x: 76
    - type: rectangle
      width: 8
      height: 32
      color: 555555
      x: 84
    - type: rectangle
      width: 8
      height: 32
      color: 444444
      x: 92
    - type: rectangle
      width: 8
      height: 32
      color: 333333
      x: 100
    - type: rectangle
      width: 8
      height: 32
      color: 222222
      x: 108
    - type: rectangle
      width: 8
      height: 32
      color: 111111
      x: 116
    - type: rectangle
      width: 8
      height: 32
      color: 000000
      x: 124

slide_player:
  container_slide: container_slide
  dmd_slide:
    dmd_slide:
      target: dmd
  show_gamma_test: dmd_gamma_test
```

(continues on next page)

(continued from previous page)

```
widget_player:
  position_widget_right:
    right_dmd_widget:
      target: dmd
  position_widget_left:
    left_dmd_widget:
      target: dmd
  position_widget_top:
    top_dmd_widget:
      target: dmd
  position_widget_bottom:
    bottom_dmd_widget:
      target: dmd
```

drop_targets (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 168: your_machine_folder/drop_targets/config/test_multiple_drop_resets_on_startup.yaml

```
#config_version=5

switches:
  switch1:
    number:

coils:
  coil1:
    number:

drop_targets:
   m1:
     debug: True
     switch: switch1

drop_target_banks:
   multiple_resets_on_game_start:
     drop_targets: m1
     reset_coils: coil1
     reset_events:
       game_started.0: 0
       game_started.1: 3s
       game_started.2: 6s
```

Listing 169: your_machine_folder/drop_targets/config/test_drop_targets.yaml

```
#config_version=5

switches:
  switch1:
    number:
  switch2:
    number:
  switch3:
    number:
  switch4:
    number:
  switch5:
    number:
  switch6:
    number:
  switch7:
    number:
  switch8:
    number:
  switch9:
    number:
  switch10:
    number:
  switch11:
    number:

coils:
  coil1:
    number:
  coil2:
    number:
  coil3:
    number:
  coil4:
    number:
    default_hold_power: 0.250
  coil5:
    number:
  coil6:
    number:
  coil7:
    number:

modes:
  - mode1

drop_targets:
   left1:
     debug: True
     switch: switch1
   left2:
     debug: True
```

(continues on next page)

(continued from previous page)

```
    switch: switch2
  left3:
    debug: True
    switch: switch3
  left4:
    debug: True
    switch: switch4
  left5:
    debug: True
    switch: switch5
  left6:
    debug: True
    switch: switch6
    reset_coil: coil2
    knockdown_coil: coil3
    knockdown_events: knock_knock
    reset_events: reset_target
  left7:
    debug: True
    reset_coil: coil4
    switch: switch7
    enable_keep_up_events: keep_up
    disable_keep_up_events: no_more_keep_up
    ignore_switch_ms: 100
    max_reset_attempts: 3
  right1:
   switch: switch8
  right2:
   switch: switch9
  center1:
   switch: switch10
   ball_search_order: 1
   ignore_switch_ms: 1000
   reset_events: reset_center1
   reset_coil: coil6
   knockdown_coil: coil7
   knockdown_events: knockdown_center1

drop_target_banks:
  left_bank:
    debug: True
    drop_targets: left1, left2, left3
    reset_coils: coil1
    reset_events:
      drop_target_bank_left_bank_down: 1s
  right_bank:
    drop_targets: right1, right2
    reset_coils: coil5
    ignore_switch_ms: 1000
    reset_events: reset_right_bank
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 170: your_machine_folder/drop_targets/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  priority: 100
  game_mode: False

drop_target_banks:
   left_bank_2:
     drop_targets: left4, left5, left6
     reset_coils: coil2
     reset_on_complete: 1s
```

event_manager (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 171: your_machine_folder/event_manager/config/test_event_manager.yaml

```
#config_version=5

event_player:
    test_event_player1:
        - test_event_player2
        - test_event_player3
    test_event_player_delayed:
        - test_event_player2|2s
        - test_event_player3:2s

random_event_player:
    test_random_event_player1:
      events:
        - test_random_event_player2
        - test_random_event_player3
      scope: machine

modes:
  - test_mode
  - game_mode
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 172: your_machine_folder/event_manager/modes/test_mode/config/test_mode.yaml

```
#config_version=5
mode:
  start_events: test_mode_start
  stop_events: test_mode_end
  game_mode: False

event_player:
    test_event_player_mode1:
        - test_event_player_mode2
        - test_event_player_mode3

random_event_player:
    test_random_event_player_mode1:
      scope: machine
      events:
        - test_random_event_player_mode2
        - test_random_event_player_mode3
    test_random_event_player_weighted:
      scope: machine
      force_different: False
      force_all: False
      events:
        out3: 1
        out4: 1000
```

Listing 173: your_machine_folder/event_manager/modes/game_mode/config/game_mode.yaml

```
#config_version=5
mode:
  start_events: game_mode_start
  stop_events: game_mode_end

random_event_player:
    test_random_event_player_mode2:
        - out1
        - out2
```

event_players (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 174: your_machine_folder/event_players/config/test_event_player.yaml

```
#config_version=5

modes:
  - mode1

event_player:
    play_express_single: event1
    play_express_multiple: event1, event2
    play_single_list:
        - event1
    play_single_string:
        event1
    play_multiple_list:
        - event1
        - event2
        - event3
    play_multiple_string:
        event1, event2, event3
    play_multiple_args2:
        event1:
          a: b
          c: d
    play_multiple_args:
        event1: {"a": "b"}
        event2: {}
        event3: {"a": 1, "b": 2}
    test_conditional{arg.abc==1}: condition_ok
    test_conditional.2{arg.abc==1}: condition_ok2
    test_conditional.3: priority_ok
    test_time_delay1: td1|1500ms
    test_time_delay2: td2|1.5s
    test_conditional_mode{mode.mode1.active}: mode1_active
    test_conditional_mode{not mode.mode1.active}: mode1_not_active
    test_conditional_handlers:
        - event_always
        - event_if_modeactive{mode.mode1.active}
        - event_if_modestopping{mode.mode1.stopping}
    test_conditional_multiples:
      conditional_response{value==0}:
        amount: zero
      conditional_response{value==1}:
        amount: one
      conditional_response{value>1}:
        amount: greater
    play_placeholder_event:
      - my_event_(machine.test)_123
    play_placeholder_args:
      loaded_event_int:
        foo:
          value: machine.testint
          type: int
      loaded_event_float:
```

(continues on next page)

(continued from previous page)

```
        foo:
          value: machine.testfloat
          type: float
      loaded_event_bool:
        foo:
          value: machine.testbool
          type: bool
      loaded_event_string:
        foo:
          value: machine.teststring
          type: string
      loaded_event_notype:
        foo:
          value: machine.testnotype
    play_event_with_kwargs:
      - event_always
      - event_(name)
    play_event_with_param_kwargs:
      event_with_param_kwargs:
        foo:
          value: (result)
          type: string
        maths:
          value: 5 * (initial)
          type: int

shows:
  test_event_show:
    - events:
      - event1
      - event2
      - event3
```

Listing 175: your_machine_folder/event_players/config/test_queue_event_player.yaml

```
#config_version=5

modes:
    - mode1

queue_event_player:
    play:
      queue_event: queue_event1
      events_when_finished: queue_event1_finished

queue_relay_player:
    relay.1:
      post: relay_start
      wait_for: relay_done
    relay:
      post: relay2_start
      wait_for: relay2_done
    relay_with_args:
      post: relay_with_args_start
```

(continues on next page)

(continued from previous page)

```
    wait_for: relay_with_args_done
    pass_args: True
```

Listing 176: your_machine_folder/event_players/config/test_random_event_player.yaml

```
#config_version=5

modes:
  - mode2

game:
  balls_per_game: 1

switches:
  s_ball:
    number:

coils:
  c_eject:
    number:

playfields:
  playfield:
    default_source_device: s_trough
    tags: default

ball_devices:
  s_trough:
    ball_switches: s_ball
    eject_coil: c_eject
    tags: trough, drain, home

random_event_player:
  test_machine_force_different:
    scope: machine
    force_different: true
    events:
      - event1
      - event2
      - event3
      - event4
  test_machine_force_all:
    scope: machine
    force_all: true
    events:
      - event1
      - event2
      - event3
      - event4
  test_machine_disable_random:
    scope: machine
    disable_random: true
    events:
      - event1
```

(continues on next page)

(continued from previous page)

```
      - event2
      - event3
      - event4
  test_machine_conditional_random:
    scope: machine
    events:
      - event1{False==True}
      - event2{True==True}
      - event3{event_arg=="foo"}
      - event4{machine.settings.foo=="bar"}
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 177: your_machine_folder/event_players/modes/mode2/config/mode2.yaml

```
#config_version=5
mode:
  start_events: start_mode2
  stop_events: stop_mode2

random_event_player:
  test_player_force_different:
    force_different: true
    events:
      - event1
      - event2
      - event3
      - event4
  test_player_force_all:
    force_all: true
    events:
      - event1
      - event2
      - event3
      - event4
  test_player_disable_random:
    disable_random: true
    events:
      - event1
      - event2
      - event3
      - event4
```

Listing 178: your_machine_folder/event_players/modes/mode1/config/mode1.yaml

```
#config_version=5
mode:
```

(continues on next page)

(continued from previous page)

```
  game_mode: False

queue_relay_player:
    relay3:
      post: relay3_start
      wait_for: relay3_done
```

extra_ball (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 179: your_machine_folder/extra_ball/config/config.yaml

```
#config_version=5

modes:
    - mode1

playfields:
  playfield:
    default_source_device: bd_launcher
    tags: default

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:

switches:
    s_trough1:
        number:
    s_trough2:
        number:
    s_ball_switch_launcher:
        number:

ball_devices:
    bd_trough:
        eject_coil: eject_coil1
        ball_switches: s_trough1, s_trough2
        debug: true
        confirm_eject_type: target
        eject_targets: bd_launcher
        tags: trough, drain, home
    bd_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
```

(continues on next page)

(continued from previous page)

```
        debug: true
        confirm_eject_type: target
        eject_timeouts: 2s

extra_ball_groups:
  main:
    enabled: yes
    award_events: award_group_eb
    max_lit: 1
    max_per_ball: 2
  disabled_eb:
    enabled: no
  no_memory:
    lit_memory: false
    max_per_game: 2

event_player:
  ball_started{current_player.ball==1 and not is_extra_ball}: first_ball
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 180: your_machine_folder/extra_ball/modes/mode1/config/mode1.yaml

```
#config_version=5
mode:
  start_events: start_mode1
  stop_events: stop_mode1

extra_balls:
    eb1:
        award_events: award_eb1
        max_per_game: 1
    eb2:
        light_events: light_eb2
        max_per_game: 2
    eb3:
        award_events: award_eb3
        enabled: false
    eb4:
        light_events: light_eb4
        enabled: false
    eb5:
        award_events: award_eb5
        light_events: light_eb5
        group: main
    eb6:
        light_events: light_eb6
        group: main
    eb7:
        light_events: light_eb7
```

(continues on next page)

(continued from previous page)

```
        group: main
    eb8:
        light_events: light_eb8
        award_events: award_eb8
        group: disabled_eb
    eb9:
        group: no_memory
        light_events: light_eb9
        award_events: award_eb9
        max_per_game: None
```

fast (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 181: your_machine_folder/fast/config/error_lights.yaml

```
#config_version=5

hardware:
    platform: fast

fast:
    driverboards: fast
    ports: com3, com4, com5, com6
    debug: true

lights:
  test_led:
    start_channel: 3
    type: rgb
  test_led1:
    previous: test_led
    type: rgb
```

Listing 182: your_machine_folder/fast/config/config.yaml

```
#config_version=5

hardware:
    platform: fast

fast:
    driverboards: fast
    ports: com3, com4, com5, com6
```

(continues on next page)

(continued from previous page)

```
    debug: true
    firmware_updates:
      - type: net
        version: "2.04"
        file: "firmware/FAST_NET_01_04_00.txt"

switches:
    s_test:
        number: 7
        platform_settings:
            debounce_open: 26
            debounce_close: 5
    s_test_nc:
        number: 26
        type: 'NC'
    s_slingshot_test:
        number: 22
    s_flipper:
        number: 1
    s_flipper_eos:
        number: 2
    s_autofire:
        number: 3
    s_test3:
        number: 3-1
    s_nux_up:
        number: 0-11
    s_nux_down:
        number: 0-12

digital_outputs:
  c_nux_motor:
    number: 0-1
    type: driver

motors:
  motorized_drop_target_bank:
    motor_left_output: c_nux_motor
    position_switches: !!omap
      - up: s_nux_up
      - down: s_nux_down
    reset_position: up
    go_to_position:
      go_up: up
      go_down: down

coils:
    c_test:
        number: 4
        default_pulse_ms: 23
        default_recycle: True
        platform_settings:
            recycle_ms: 27
    c_test_allow_enable:
```

(continues on next page)

(continued from previous page)

```
        number: 6
        default_pulse_ms: 23
        max_hold_power: 1.0
    c_slingshot_test:
        number: 7
    c_pulse_pwm32_mask:
        number: 17
        default_pulse_power: 0.53
        default_hold_power: 0.40
    c_hold_ssm:
        number: 19
        default_hold_power: 0.25    # approximately
        platform_settings:
            hold_pwm_patter: "84224244"
    c_long_pulse:
        number: 18
        default_pulse_ms: 2000
        max_hold_power: 1.0
    c_timed_enable:
        number: 22
        default_pulse_ms: 20
        default_timed_enable_ms: 200
        default_hold_power: 0.25
    c_default_timed_enable:
        number: 23
        default_pulse_ms: 20
        default_timed_enable_ms: 200
        default_hold_power: 0.25
        pulse_with_timed_enable: true
    c_flipper_main:
        number: 32
        default_pulse_ms: 10
        default_hold_power: 0.125
    c_flipper_hold:
        number: 3-5
        default_hold_power: 0.125

autofire_coils:
    ac_slingshot_test:
        coil: c_slingshot_test
        switch: s_slingshot_test
    ac_inverted_switch:
        coil: c_slingshot_test
        switch: s_test_nc
    ac_same_switch1:
        coil: c_test
        switch: s_autofire
        enable_events: ac_same_switch
    ac_same_switch2:
        coil: c_test_allow_enable
        switch: s_autofire
        enable_events: ac_same_switch
    ac_broken_combination:
        coil: c_flipper_hold
```

(continues on next page)

(continued from previous page)

```
        switch: s_slingshot_test
    ac_different_boards:
        coil: c_flipper_hold
        switch: s_test
    ac_board_3:
        coil: c_flipper_hold
        switch: s_test3

servos:
    servo1:
        number: 3
    servo20:
        number: 3-2

flippers:
    f_test_single:
        debug: true
        main_coil_overwrite:
            pulse_ms: 11
        main_coil: c_flipper_main
        activation_switch: s_flipper

    f_test_hold:
        debug: true
        main_coil: c_flipper_main
        hold_coil: c_flipper_hold
        activation_switch: s_flipper

    f_test_hold_eos:
        debug: true
        main_coil: c_flipper_main
        hold_coil: c_flipper_hold
        activation_switch: s_flipper
        eos_switch: s_flipper_eos
        use_eos: true

lights:
  test_pdb_light:
    number: 35
    subtype: matrix
  test_gi:
    number: 42
    subtype: gi
  test_led:
    number: 2-23
    type: grb
  test_led2:
    previous: test_led
    type: grb
  l_o_circle:
    number: 0-10
    type: grb
```

Listing 183: your_machine_folder/fast/config/config_v1.yaml

```
#config_version=5

hardware:
    platform: fast

fast:
    driverboards: fast
    ports: com3, com4, com5, com6
    debug: true
    firmware_updates:
      - type: net
        version: "1.04"
        file: "firmware/FAST_NET_01_04_00.txt"

switches:
    s_test:
        number: 7
        platform_settings:
            debounce_open: 26
            debounce_close: 5
    s_test_nc:
        number: 26
        type: 'NC'
    s_slingshot_test:
        number: 22
    s_flipper:
        number: 1
    s_flipper_eos:
        number: 2
    s_autofire:
        number: 3
    s_test3:
        number: 3-1
    s_nux_up:
        number: 0-11
    s_nux_down:
        number: 0-12

digital_outputs:
  c_nux_motor:
    number: 0-1
    type: driver

motors:
  motorized_drop_target_bank:
    motor_left_output: c_nux_motor
    position_switches: !!omap
      - up: s_nux_up
      - down: s_nux_down
    reset_position: up
    go_to_position:
      go_up: up
```

(continues on next page)

(continued from previous page)

```
      go_down: down

coils:
    c_test:
        number: 4
        default_pulse_ms: 23
        default_recycle: True
        platform_settings:
            recycle_ms: 27
    c_test_allow_enable:
        number: 6
        default_pulse_ms: 23
        max_hold_power: 1.0
    c_slingshot_test:
        number: 7
    c_pulse_pwm32_mask:
        number: 17
        default_pulse_power: 0.53
        default_hold_power: 0.40
    c_hold_ssm:
        number: 19
        default_hold_power: 0.25    # approximately
        platform_settings:
            hold_pwm_patter: "84224244"
    c_long_pulse:
        number: 18
        default_pulse_ms: 2000
        max_hold_power: 1.0
    c_timed_enable:
        number: 22
        default_pulse_ms: 20
        default_timed_enable_ms: 200
        default_hold_power: 0.25
    c_default_timed_enable:
        number: 23
        default_pulse_ms: 20
        default_timed_enable_ms: 200
        default_hold_power: 0.25
        pulse_with_timed_enable: true
    c_flipper_main:
        number: 32
        default_pulse_ms: 10
        default_hold_power: 0.125
    c_flipper_hold:
        number: 3-5
        default_hold_power: 0.125

autofire_coils:
    ac_slingshot_test:
        coil: c_slingshot_test
        switch: s_slingshot_test
    ac_inverted_switch:
        coil: c_slingshot_test
        switch: s_test_nc
```

(continues on next page)

(continued from previous page)

```
    ac_same_switch1:
        coil: c_test
        switch: s_autofire
        enable_events: ac_same_switch
    ac_same_switch2:
        coil: c_test_allow_enable
        switch: s_autofire
        enable_events: ac_same_switch
    ac_broken_combination:
        coil: c_flipper_hold
        switch: s_slingshot_test
    ac_different_boards:
        coil: c_flipper_hold
        switch: s_test
    ac_board_3:
        coil: c_flipper_hold
        switch: s_test3

servos:
    servo1:
        number: 3
    servo20:
        number: 3-2

flippers:
    f_test_single:
        debug: true
        main_coil_overwrite:
            pulse_ms: 11
        main_coil: c_flipper_main
        activation_switch: s_flipper

    f_test_hold:
        debug: true
        main_coil: c_flipper_main
        hold_coil: c_flipper_hold
        activation_switch: s_flipper

    f_test_hold_eos:
        debug: true
        main_coil: c_flipper_main
        hold_coil: c_flipper_hold
        activation_switch: s_flipper
        eos_switch: s_flipper_eos
        use_eos: true

lights:
  test_pdb_light:
    number: 35
    subtype: matrix
  test_gi:
    number: 42
    subtype: gi
  test_led:
```

(continues on next page)

(continued from previous page)

```
    number: 2-23
    type: grb
  test_led2:
    previous: test_led
    type: grb
  l_o_circle:
    number: 0-10
    type: grb
```

flippers (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 184: your_machine_folder/flippers/config/hold_no_eos.yaml

```
#config_version=5

hardware:
  platform: fast
  driverboards: fast

switches:
    s_left_flipper:
        number: 0-0
        tags: left_flipper
    s_right_flipper:
        number: 0-1
        tags: right_flipper

coils:
    c_flipper_left_main:
        number: 0-0
        default_pulse_ms: 30
    c_flipper_left_hold:
        number: 0-1
        default_hold_power: 1.0
    c_flipper_right_main:
        number: 0-2
        default_pulse_ms: 30
    c_flipper_right_hold:
        number: 0-3
        default_hold_power: 1.0

flippers:
    left_flipper:
```

(continues on next page)

(continued from previous page)

```
        main_coil: c_flipper_left_main
        hold_coil: c_flipper_left_hold
        activation_switch: s_left_flipper
        enable_events: machine_reset_phase_3
    right_flipper:
        main_coil: c_flipper_right_main
        hold_coil: c_flipper_right_hold
        activation_switch: s_right_flipper
        enable_events: machine_reset_phase_3
```

Listing 185: your_machine_folder/flippers/config/config.yaml

```
#config_version=5

game:
    balls_per_game: 1

coils:
    c_flipper_main:
        number:
        default_pulse_ms: 10
        default_hold_power: 0.125
    c_flipper_hold:
        number:
        default_hold_power: 1.0


switches:
    s_flipper:
        number: 1
        tags: left_flipper
    s_flipper_eos:
        number: 2

flippers:
    f_test_single:
        debug: true
        main_coil: c_flipper_main
        activation_switch: s_flipper
        sw_flip_events: flip_single
        sw_release_events: release_single

    f_test_hold:
        debug: true
        main_coil: c_flipper_main
        hold_coil: c_flipper_hold
        activation_switch: s_flipper

    f_test_hold_eos:
        debug: true
        main_coil: c_flipper_main
        hold_coil: c_flipper_hold
        activation_switch: s_flipper
        eos_switch: s_flipper_eos
```

(continues on next page)

(continued from previous page)

```
        use_eos: true
        sw_flip_events: flip_hold
        sw_release_events: release_hold

    f_test_flippers_with_settings:
        debug: true
        main_coil: c_flipper_main
        power_setting_name: flipper_power
        activation_switch: s_flipper
```

Listing 186: your_machine_folder/flippers/config/software_eos_repulse.yaml

```
#config_version=5

hardware:
  platform: fast
  driverboards: fast

switches:
    s_flipper_single:
        number: 0-0
    s_flipper_single_eos:
        number: 0-1
    s_flipper_dual_wound:
        number: 0-2
    s_flipper_dual_wound_eos:
        number: 0-3

coils:
    c_flipper_single_main:
        number: 0-0
        default_pulse_ms: 30
        default_hold_power: .3
    c_flipper_dual_wound_hold:
        number: 0-1
        default_hold_power: 1.0
    c_flipper_dual_wound_main:
        number: 0-2
        default_pulse_ms: 30

flippers:
    single_flipper:
        main_coil: c_flipper_single_main
        activation_switch: s_flipper_single
        eos_switch: s_flipper_single_eos
        use_eos: true
        repulse_on_eos_open: true
        enable_events: enable_flipper_single
        disable_events: disable_flipper_single
    dual_wound_flipper:
        main_coil: c_flipper_dual_wound_main
        hold_coil: c_flipper_dual_wound_hold
        activation_switch: s_flipper_dual_wound
        eos_switch: s_flipper_dual_wound_eos
```

(continues on next page)

(continued from previous page)

```
      use_eos: true
      repulse_on_eos_open: true
      enable_events: enable_flipper_dual_wound
      disable_events: disable_flipper_dual_wound
```

fonts (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 187: your_machine_folder/fonts/config/built_in_dmd_fonts.yaml

```
#config_version=5

displays:
  window:
    width: 800
    height: 600
  dmd:
    width: 128
    height: 32
    default: yes

slides:
  window:
    - type: display
      width: 640
      height: 160
      effects:
        type: dmd
    - type: text
      text: DMD FONT & POSITIONING TEST
      font_size: 50
      y: 410
    - type: rectangle
      width: 642
      height: 162
      color: gray
  dmd_small:
    - type: text
      style: small
      text: DMD_SMALL
      anchor_y: top
      y: top
    - type: text
      style: small
      text: DMD_SMALL
      anchor_y: bottom
      y: bottom
```

(continues on next page)

(continued from previous page)

```
  dmd_med:
    - type: text
      style: medium
      text: DMD_MED
      anchor_y: top
      y: top
    - type: text
      style: medium
      text: DMD_MED
      anchor_y: bottom
      y: bottom
  dmd_big:
    - type: text
      style: big
      text: DMD_BIG
      anchor_y: top
      y: top
    - type: text
      style: big
      text: DMD_BIG
      anchor_y: bottom
      y: bottom

slide_player:
  window_slide:
    window:
      target: window
  dmd_small: dmd_small
  dmd_med: dmd_med
  dmd_big: dmd_big
```

game (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 188: your_machine_folder/game/config/config.yaml

```
#config_version=5

game:
    balls_per_game: 3
    start_game_event: start_my_game
    add_player_event: add_my_player

machine:
```

(continues on next page)

(continued from previous page)

```
    min_balls: 2
    balls_installed: 2

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:

switches:
    s_start:
        number:
        tags: start
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch3:
        number:
    s_ball_switch_launcher:
        number:

playfields:
    playfield:
        default_source_device: bd_launcher
        tags: default
        enable_ball_search: True
    second_playfield:
        default_source_device: bd_launcher

ball_devices:
    bd_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2, s_ball_switch3
        debug: true
        confirm_eject_type: target
        eject_targets: bd_launcher
        tags: trough, drain, home
    bd_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
        debug: true
        confirm_eject_type: target
        eject_targets: playfield, second_playfield
        eject_timeouts: 2s
```

Listing 189: your_machine_folder/game/config/config_with_balls.yaml

```
#config_version=5

config:
    - config.yaml

virtual_platform_start_active_switches:
```

(continues on next page)

(continued from previous page)

```
  - s_ball_switch1
```

head2head (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 190: your_machine_folder/head2head/config/config.yaml

```
#config_version=5

playfields:
    playfield:    # remove default playfield
        _delete: True
    playfield_front:
        label: Playfield Front
        default_source_device: bd_feeder_front
    playfield_back:
        label: Playfield Back
        default_source_device: bd_feeder_back

switches:
  s_trough1_front:
    number:
  s_trough2_front:
    number:
  s_trough3_front:
    number:
  s_trough4_front:
    number:
  s_trough1_back:
    number:
  s_trough2_back:
    number:
  s_trough3_back:
    number:
  s_trough4_back:
    number:
  s_launcher_lane_front:
    number:
  s_launcher_lane_back:
    number:
  s_middle_front1:
    number:
  s_middle_back1:
    number:
  s_feeder_front:
    number:
  s_feeder_back:
```

(continues on next page)

(continued from previous page)

```
    number:
  s_launcher_diverter_front:
    number:
  s_launcher_diverter_back:
    number:
  s_transfer_front_back:
    number:
  s_transfer_back_front:
    number:
  s_playfield_front:
    number:
    tags: playfield_front_active
  s_playfield_back:
    number:
    tags: playfield_back_active

coils:
  c_trough_eject_front:
    number:
  c_trough_eject_back:
    number:
  c_launcher_eject_front:
    number:
  c_launcher_eject_back:
    number:
  c_lock_figur_front:
    number:
    default_hold_power: 0.125
  c_lock_figur_back:
    number:
    default_hold_power: 0.125
  c_feeder_front:
    number:
    default_hold_power: 0.125
  c_feeder_back:
    number:
    default_hold_power: 0.125

ball_devices:
  bd_trough_front:
    ball_switches: s_trough1_front, s_trough2_front, s_trough3_front, s_trough4_front
    eject_coil: c_trough_eject_front
    eject_targets: bd_launcher_front
    tags: trough, home, drain
    captures_from: playfield_front
    ball_missing_target: playfield_front
    debug: true
  bd_trough_back:
    ball_switches: s_trough1_back, s_trough2_back, s_trough3_back, s_trough4_back
    eject_coil: c_trough_eject_back
    eject_targets: bd_launcher_back
    tags: trough, home, drain
    captures_from: playfield_back
    ball_missing_target: playfield_back
```

(continues on next page)

(continued from previous page)

```
    debug: true
  bd_launcher_front:
    ball_switches: s_launcher_lane_front
    confirm_eject_type: switch
    confirm_eject_switch: s_launcher_diverter_back
    eject_coil: c_launcher_eject_front
    eject_targets: bd_feeder_back, bd_trough_back
    captures_from: playfield_front
    ball_missing_target: playfield_back
    debug: true
  bd_launcher_back:
    ball_switches: s_launcher_lane_back
    confirm_eject_type: switch
    confirm_eject_switch: s_launcher_diverter_front
    eject_coil: c_launcher_eject_back
    eject_targets: bd_feeder_front, bd_trough_front
    captures_from: playfield_back
    ball_missing_target: playfield_front
    debug: true
  bd_middle_front:
    hold_switches: s_middle_front1
    ball_switches: s_middle_front1
    confirm_eject_type: target
    hold_coil: c_lock_figur_front
    eject_targets: playfield_front
    captures_from: playfield_back
    ball_missing_target: playfield_front
    target_on_unexpected_ball: playfield_front
    debug: true
  bd_middle_back:
    hold_switches: s_middle_back1
    ball_switches: s_middle_back1
    confirm_eject_type: target
    hold_coil: c_lock_figur_back
    eject_targets: playfield_back
    captures_from: playfield_front
    ball_missing_target: playfield_back
    target_on_unexpected_ball: playfield_back
    debug: true
  bd_feeder_front:
    ball_switches: s_feeder_front
    hold_switches: s_feeder_front
    hold_coil: c_feeder_front
    eject_targets: playfield_front
    captures_from: playfield_front
    ball_missing_target: playfield_front
    eject_timeouts: 2s
    debug: true
  bd_feeder_back:
    ball_switches: s_feeder_back
    hold_switches: s_feeder_back
    hold_coil: c_feeder_back
    eject_targets: playfield_back
    captures_from: playfield_back
```

(continues on next page)

(continued from previous page)

```
    ball_missing_target: playfield_back
    eject_timeouts: 2s
    debug: true

playfield_transfers:
    transfer_front_back:
        ball_switch: s_transfer_front_back
        captures_from: playfield_front
        eject_target: playfield_back
    transfer_back_front:
        ball_switch: s_transfer_back_front
        captures_from: playfield_back
        eject_target: playfield_front
```

high_score (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 191: your_machine_folder/high_score/config/high_score.yaml

```
#config_version=5

modes:
  - high_score
  - tilt

switches:
  s_tilt:
    tags: tilt_warning
    number:
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 192: your_machine_folder/high_score/modes/high_score/config/high_score.yaml

```
#config_version=5
high_score:
  _overwrite: True
  categories: !!omap
  - score:
      - GRAND CHAMPION
      - HIGH SCORE 1
      - HIGH SCORE 2
      - HIGH SCORE 3
      - HIGH SCORE 4
```

(continues on next page)

(continued from previous page)

```
  - loops:
      - LOOP CHAMP
  defaults:
    score:
      - BRI: 4242
      - GHK: 2323
      - JK: 1337
      - QC: 42
      - MPF: 23
    loops:
      - JK: 42
```

high_score_reverse (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 193: your_machine_folder/high_score_reverse/config/high_score.yaml

```
#config_version=5

modes:
  - high_score
  - tilt

switches:
  s_tilt:
    tags: tilt_warning
    number:
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 194: your_machine_folder/high_score_reverse/modes/high_score/config/high_score.yaml

```
#config_version=5
high_score:
  _overwrite: True
  categories: !!omap
  - score:
      - GRAND CHAMPION
      - HIGH SCORE 1
      - HIGH SCORE 2
      - HIGH SCORE 3
      - HIGH SCORE 4
  - loops:
      - LOOP CHAMP
```

(continues on next page)

(continued from previous page)

```
  - time_to_wizard:
      - FASTEST WIZARD
      - ALMOST FASTEST WIZARD
  defaults:
    score:
      - BRI: 4242
      - GHK: 2323
      - JK: 1337
      - QC: 42
      - MPF: 23
    loops:
      - JK: 42
    time_to_wizard:
      - JK: 300
      - BM: 350
  reverse_sort:
    - time_to_wizard
```

i2c_servo_controller (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 195: your_machine_folder/i2c_servo_controller/config/config.yaml

```
#config_version=5
hardware:
    servo_controllers: i2c_servo_controller

servos:
    servo1:
        number: 3
    servo2:
        number: bus1-64-7
    servo3:
        number: 4
```

info_lights (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 196: your_machine_folder/info_lights/config/config.yaml

```
#config_version=5

machine:
    min_balls: 0

switches:
    s_start:
        number: 1
        tags: start

lights:
    match00:
        number:
        subtype: matrix
    match10:
        number:
        subtype: matrix
    match20:
        number:
        subtype: matrix
    match30:
        number:
        subtype: matrix
    match40:
        number:
        subtype: matrix
    match50:
        number:
        subtype: matrix
    match60:
        number:
        subtype: matrix
    match70:
        number:
        subtype: matrix
    match80:
        number:
        subtype: matrix
    match90:
        number:
        subtype: matrix
    bip1:
        number:
        subtype: matrix
    bip2:
        number:
        subtype: matrix
    bip3:
        number:
        subtype: matrix
    player1:
        number:
```

(continues on next page)

(continued from previous page)

```
        subtype: matrix
    player2:
        number:
        subtype: matrix
    tilt:
        number:
    gameOver:
        number:

info_lights:
    match_00:
        light: match00
    match_10:
        light: match10
    match_20:
        light: match20
    match_30:
        light: match30
    match_40:
        light: match40
    match_50:
        light: match50
    match_60:
        light: match60
    match_70:
        light: match70
    match_80:
        light: match80
    match_90:
        light: match90
    ball_1:
        light: bip1
    ball_2:
        light: bip2
    ball_3:
        light: bip3
    player_1:
        light: player1
    player_2:
        light: player2
    tilt:
        light: tilt
    game_over:
        light: gameOver
```

keyboard (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 197: your_machine_folder/keyboard/config/test_keyboard.yaml

```
#config_version=5
keyboard:
  a:
    switch: switch_a
  b:
    switch: switch_b
    toggle: true
  c:
    switch: switch_c
    invert: true
  d:
    event: event_d
  e:
    event: event_e
    params:
      foo: bar
      mission: pinball
  f:
    mc_event: event_f
  g:
    mc_event: event_g
    params:
      foo: bar
      mission: pinball
  shift-a:
    switch: shift_a
  shift+b:
    switch: shift_b
  shift-ctrl-c:
    switch: shift_ctrl_c
  1:
    switch: switch_1
  .:
    switch: switch_period
  /:
    switch: switch_slash
```

kickback (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 198: your_machine_folder/kickback/config/config.yaml

```
#config_version=5

coils:
```

(continues on next page)

(continued from previous page)

```yaml
  kickback_coil:
    number:
    default_pulse_ms: 100

switches:
  s_kickback:
    number:

kickbacks:
  kickback_test:
    coil: kickback_coil
    switch: s_kickback
    enable_events: kickback_enable
    disable_events: kickback_kickback_test_fired

ball_saves:
  kickback_save:
    balls_to_save: 1
    active_time: 5s
    enable_events: kickback_kickback_test_fired
```

light (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 199: your_machine_folder/light/config/lights_on_drivers.yaml

```yaml
#config_version=5

hardware:
  platform: virtual

coils:
    coil_01:
        number: 1
        allow_enable: True

lights:
    light_on_driver:
        number: coil_01
        platform: drivers
        debug: True
```

Listing 200: your_machine_folder/light/config/light_default_color_correction.yaml

```
#config_version=5

light_settings:
    default_color_correction_profile: correction_profile_1
    color_correction_profiles:
        correction_profile_1:
            gamma: 1
            whitepoint: [0.9, 0.8, 0.7]
            linear_slope: 0.75
            linear_cutoff: 0.1

lights:
  led1:
    number: 1
```

Listing 201: your_machine_folder/light/config/matrix_lights.yaml

```
#config_version=5
lights:
    light_01:
        number: 0
        subtype: matrix
        debug: True
    light_02:
        number: 1
        subtype: matrix
        debug: True
```

Listing 202: your_machine_folder/light/config/light.yaml

```
#config_version=5

light_settings:
    color_correction_profiles:
        correction_profile_1:
            gamma: 1
            whitepoint: [0.9, 0.8, 0.7]
            linear_slope: 0.75
            linear_cutoff: 0.1

named_colors:
    jans_red: [251, 23, 42]

lights:
  led1:
    number: 1
    default_on_color: red
    debug: True
    x: 0.4
    y: 0.5
    z: 0
  led2:
```

(continues on next page)

(continued from previous page)

```
    channels:
      red:
        number: 4
      green:
        number: 3
      blue:
        number: 2
    debug: True
    x: 0.6
    y: 0.7
  led_bgr_2:
    type: bgr
    number: 42
    debug: True
  led3:
    channels:
      red:
        - number: 7
      green:
        - number: 8
      blue:
        - number: 9
      white:
        - number: 10
    debug: True
  led4:
    number: 11
    fade_ms: 1s
  led_corrected:
    number:
    color_correction_profile: correction_profile_1
  led_www:
    number: 23
    type: www
    debug: True
```

Listing 203: your_machine_folder/light/config/light_groups.yaml

```
#config_version=5

light_stripes:
  stripe1:
    number_start: 10
    light_template:
      tags: test
    count: 5
    debug: True
  stripe2:
    number_start: 200
    number_template: 7-{}
    count: 5
    direction: 90
    start_x: 10
    start_y: 20
```

(continues on next page)

(continued from previous page)

```
    distance: 5
    debug: True
  stripe3:
    start_channel: ABC-123
    count: 5
    direction: 90
    start_x: 10
    start_y: 20
    distance: 5
    debug: True
    light_template:
      type: rgbw

light_rings:
  ring1:
    number_start: 20
    count: 12
    radius: 3
    start_angle: 90
    center_x: 100
    center_y: 50
    debug: True

neoseg_displays:
  neoSeg_0:
    start_channel: 0-0-0
    size: 8digit
    light_template:
      type: w
      subtype: led
  neoSeg_1:
    start_channel: 0-0-120
    size: 2digit
    light_template:
      type: w
      subtype: led
```

light_player (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 204: your_machine_folder/light_player/config/light_player_named_colors.yaml

```
#config_version=5
lights:
```

(continues on next page)

(continued from previous page)

```
  l_gi_1:
    number:
  l_gi_2:
    number:


named_colors:
   tt_yellow: [255, 220, 0]

show_player:
   skill_started:
      giSwipeDown:
         show_tokens:
            frontColor: black
            backColor: tt_yellow

shows:
  giSwipeDown:
    - time: 0
      lights:
        l_gi_1: (frontColor)
        l_gi_2: (backColor)
```

Listing 205: your_machine_folder/light_player/config/light_player.yaml

```
#config_version=5

modes:
 - mode1
 - mode2

lights:
  led1:
    debug: True
    number:
    tags: tag1
  led2:
    debug: True
    number:
    tags: tag1
  led3:
    debug: True
    number:
    tags:
  led4:
    debug: True
    number:
    tags:
  led5:
    debug: True
    number:
    default_on_color: red
  led6:
    debug: True
```

(continues on next page)

(continued from previous page)

```
    number:

light_player:
  "{machine.a == 7}":
    led1: red
  event1:
    led1:
      color: red
      fade: 0
      priority: 200
    led2:
      color: ff0000
      fade: 0
    led3:
      color: red
      fade: 0
  event2:
    tag1:
      color: blue
      fade: 200ms
      priority: 100
  event3:
    led1: lime-f500
    led2: lime - f 500ms
    led3: 00ff00-f.5s
  event4:
    tag1: 00ffff
  event5:
    led5: on

shows:
  show1:
  - time: 0
    lights:
      led1: red
      led2: red
      led3: red
  show2:
  - time: 0
    lights:
      led1: red
      led2: red
      led3: red
  - time: 1
  show3:
  - time: 0
    lights:
      led1: blue
      led2: blue
      led3: blue
  - time: 1
  show2_stay_on:
  - time: 0
    duration: -1
```

(continues on next page)

(continued from previous page)

```
    lights:
      led1: red
      led2: red
      led3: red

show_player:
  play_show1: show_ext1
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 206: your_machine_folder/light_player/modes/mode2/config/mode2.yaml

```
#config_version=5
mode:
  priority: 200
  start_events: ball_starting
  stop_events: ball_ending

light_player:
  "{machine.test == 23}":
    led4: red
  "{current_player.test == 42}":
    led5: red
```

Listing 207: your_machine_folder/light_player/modes/mode1/config/mode1.yaml

```
#config_version=5
mode:
  priority: 100
  game_mode: False

light_player:
  event5:
    led1:
      color: orange
    led2:
      color: orange
    led3:
      color: orange
      priority: 200
  "{machine.test == 23}":
    led4: red
```

Show file examples

Here are some example show files that go along with the above config(s).

Note that there are multiple shows here.

Listing 208: your_machine_folder/light_player/shows/show_ext1.yaml

```
#show_version=5
- shows:
    attract:
      show: show_ext2
      show_tokens:
        light_color: blue
  duration: 3s
- shows:
    attract:
      show: show_ext2
      show_tokens:
        light_color: red
  duration: 3s
```

Listing 209: your_machine_folder/light_player/shows/show_ext2.yaml

```
#show_version=5
- duration: -1
  shows:
    instance1:
      show: show_ext3
      show_tokens:
        led1: led1
        led2: led2
        led3: led3
        color_on: (light_color)
        color_off: black
    instance2:
      show: show_ext3
      show_tokens:
        led1: led4
        led2: led5
        led3: led6
        color_on: (light_color)
        color_off: black
```

Listing 210: your_machine_folder/light_player/shows/show_ext3.yaml

```
#show_version=5
- lights:
    (led1): (color_on)
    (led2): (color_off)
    (led3): (color_off)
  duration: 1s
- lights:
    (led1): (color_off)
    (led2): (color_on)
    (led3): (color_off)
  duration: 1s
- lights:
    (led1): (color_off)
```

(continues on next page)

(continued from previous page)

```
    (led2): (color_off)
    (led3): (color_on)
  duration: 1s
```

light_segment_displays (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 211: your_machine_folder/light_segment_displays/config/config.yaml

```
#config_version=5

hardware:
  segment_displays: light_segment_displays

lights:
  segment1_a:
    number:
  segment1_b:
    number:
  segment1_c:
    number:
  segment1_d:
    number:
  segment1_e:
    number:
  segment1_f:
    number:
  segment1_g:
    number:
  segment2_a:
    number:
  segment2_b:
    number:
  segment2_c:
    number:
  segment2_d:
    number:
  segment2_e:
    number:
  segment2_f:
    number:
  segment2_g:
    number:
  segment3_x0:
```

(continues on next page)

(continued from previous page)

```
    number:
  segment3_x1:
    number:
  segment3_x2:
    number:
  segment3_x3:
    number:
  segment4_x0:
    number:
  segment4_x1:
    number:
  segment4_x2:
    number:
  segment4_x3:
    number:
  segment5_a:
    number:
  segment5_b:
    number:
  segment5_c:
    number:
  segment5_d:
    number:
  segment5_e:
    number:
  segment5_f:
    number:
  segment5_g:
    number:
  segment5_h:
    number:

neoseg_displays:
  neoSeg_0:
    start_channel: 0-0-0
    size: 8digit
    light_template:
      type: w
      subtype: led
  neoSeg_1:
    start_channel: 0-0-120
    size: 8digit
    light_template:
      type: w
      subtype: led

segment_displays:
  display1:
    number: 1
    size: 4
    platform_settings:
      lights:
        - a: segment1_a
          b: segment1_b
```

(continues on next page)

(continued from previous page)

```
        c: segment1_c
        d: segment1_d
        e: segment1_e
        f: segment1_f
        g: segment1_g
      - a: segment2_a
        b: segment2_b
        c: segment2_c
        d: segment2_d
        e: segment2_e
        f: segment2_f
        g: segment2_g
    type: 7segment
  display2:
    number: 2
    size: 4
    platform_settings:
      lights:
        - x0: segment3_x0
          x1: segment3_x1
          x2: segment3_x2
          x3: segment3_x3
        - x0: segment4_x0
          x1: segment4_x1
          x2: segment4_x2
          x3: segment4_x3
      type: bcd
  display3:
    number: 4
    size: 1
    platform_settings:
      lights:
        - a: segment5_a
          b: segment5_b
          c: segment5_c
          d: segment5_d
          e: segment5_e
          f: segment5_f
          g: segment5_g
          h: segment5_h
      type: 8segment
  neoSegTop:
    number: 1
    size: 16
    integrated_dots: true
    use_dots_for_commas: true
    default_transition_update_hz: 30
    platform_settings:
      light_groups:
         - neoSeg_0
         - neoSeg_1
      type: 14segment

segment_display_player:
```

(continues on next page)

(continued from previous page)

```
  show_1337:
    display1:
      text: "1337"
  display1_color_red_green_blue_yellow:
    display1:
      action: set_color
      color: [red, green, blue, yellow]
  display1_color_white:
    display1:
      action: set_color
      color: [white]
  show_88:
    display1:
      text: "88"
  show_11:
    display1:
      text: "11"
  remove_text_display1:
    display1:
      action: remove
  show_centered_11:
    neoSegTop:
      text: "       11       "
```

Listing 212: your_machine_folder/light_segment_displays/config/config_dots.yaml

```
#config_version=5

hardware:
  segment_displays: light_segment_displays

lights:
  segment1_a:
    number:
  segment1_b:
    number:
  segment1_c:
    number:
  segment1_d:
    number:
  segment1_e:
    number:
  segment1_f:
    number:
  segment1_g:
    number:
  segment1_dp:
    number:
  segment2_a:
    number:
  segment2_b:
    number:
  segment2_c:
    number:
```

(continues on next page)

(continued from previous page)

```
  segment2_d:
    number:
  segment2_e:
    number:
  segment2_f:
    number:
  segment2_g:
    number:
  segment2_dp:
    number:

segment_displays:
  display1:
    number: 1
    size: 2
    integrated_dots: true
    platform_settings:
      lights:
        - a: segment1_a
          b: segment1_b
          c: segment1_c
          d: segment1_d
          e: segment1_e
          f: segment1_f
          g: segment1_g
          dp: segment1_dp
        - a: segment2_a
          b: segment2_b
          c: segment2_c
          d: segment2_d
          e: segment2_e
          f: segment2_f
          g: segment2_g
          dp: segment2_dp
      type: 7segment

segment_display_player:
  show_37dot:
    display1:
      text: "37."
      color: [red, blue]
```

lisy (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 213: your_machine_folder/lisy/config/config.yaml

```
#config_version=5

hardware:
    platform: lisy

lisy:
    debug: True
    connection: network
    network_port: 1234
    network_host: "localhost"
#    connection: serial
#    port: com1
#    baud: 115200

switches:
    s_test00:
        number: 00
    s_test37:
        number: 37
    s_test77_nc:
        number: 77
        type: 'NC'

coils:
    c_test:
        number: 0
    c_test_allow_enable:
        number: 1
        default_hold_power: 1.0
    c_trough_eject:
        number: 103
        default_pulse_ms: 3s

digital_outputs:
    game_over_relay:
        number: 1
        type: light
        enable_events: ball_started
        disable_events: ball_will_end

lights:
  test_light:
    number: 3

segment_displays:
  info_display:
    number: 0
  player1_display:
    number: 1
  player2_display:
    number: 2
```

(continues on next page)

(continued from previous page)

```
hardware_sound_systems:
    default:
        label: LISY

hardware_sound_player:
    test2:
        2:
            action: play
    play_file:
        "some_file": play_file
    play_file_loop:
        "some_file":
          action: play_file
          platform_options:
            loop: True
            no_cache: False
    play_text:
        text:
          action: text_to_speech
          value: "Hello MPF"
          platform_options:
            loop: False
            no_cache: True
    volume_05:
        set_volume:
          action: set_volume
          value: 0.5
    increase_volume:
        0.1: increase_volume
    decrease_volume:
        decrease_volume:
          action: decrease_volume
          value: 0.01
    test3:
        3: play
    test_stop: stop
```

Listing 214: your_machine_folder/lisy/config/config_modern.yaml

```
#config_version=5

hardware:
    platform: lisy

lisy:
    connection: serial
    port: com1
    baud: 115200
    debug: true

switches:
    s_test00:
        number: 00
    s_flipper:
```

(continues on next page)

(continued from previous page)

```
        number: 1
    s_flipper_eos:
        number: 2
    s_slingshot:
        number: 3
    s_test37:
        number: 37
    s_test77_nc:
        number: 77
        type: 'NC'

coils:
    c_test:
        number: 0
    c_test_allow_enable:
        number: 1
        default_hold_power: 1.0
    c_flipper_main:
      number: 5
      default_pulse_ms: 30
    c_flipper_hold:
      number: 6
      allow_enable: True
    c_slingshot:
      number: 7

lights:
  test_light0:
    start_channel: 0
    type: rgb
    subtype: light
  test_light1:
    previous: test_light0
    type: rgbw
    subtype: light

flippers:
    f_test_hold_eos:
        debug: true
        main_coil: c_flipper_main
        hold_coil: c_flipper_hold
        activation_switch: s_flipper
        eos_switch: s_flipper_eos
        use_eos: true

autofire_coils:
    ac_slingshot:
        coil: c_slingshot
        switch: s_slingshot
```

Listing 215: your_machine_folder/lisy/config/config_system11.yaml

```
#config_version=5
```

(continues on next page)

(continued from previous page)

```
hardware:
    platform: lisy
    coils: system11

lisy:
    connection: network
    network_port: 1234
    network_host: "localhost"

system11:
  ac_relay_driver: c_ac_relay

switches:
    s_test00:
        number: 00
    s_test37:
        number: 37
    s_test77_nc:
        number: 77
        type: 'NC'

coils:
    c_test:
        number: 0
    c_test1_c_side:
        number: 1c
    c_test1_a_side:
        number: 1a
    c_ac_relay:
        number: 8
        allow_enable: True

segment_displays:
  info_display:
    number: 0
  player1_display:
    number: 1
  player2_display:
    number: 2
```

logic_blocks (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 216: your_machine_folder/logic_blocks/config/config.yaml

```
#config_version=5
```

(continues on next page)

(continued from previous page)

```
lights:
  led1:
    number:
  led2:
    number:
  led3:
    number:

switches:
  s_qualify1:
    number:
  s_qualify2:
    number:

# system wide logic blocks
accruals:
    accrual1:
        events:
            - accrual1_step1a, accrual1_step1b, accrual1_step1c
            - accrual1_step2a, accrual1_step2b, accrual1_step2c
            - accrual1_step3a, accrual1_step3b, accrual1_step3c
        events_when_complete: accrual1_complete1, accrual1_complete2
        enable_events: accrual1_enable
        disable_events: accrual1_disable
        reset_events: accrual1_reset
        events_when_hit: accrual1_hit
        advance_random_events: accrual1_random_advance
    accrual2:
        events:
            - accrual2_step1
            - accrual2_step2
        restart_events: accrual2_restart
    accrual3:
        events:
            - accrual3_step1
            - accrual3_step2
        reset_on_complete: False
        disable_on_complete: True
        enable_events: accrual3_enable
        disable_events: accrual3_disable
        reset_events: accrual3_reset
    accrual4:
        events:
            - accrual4_step1
            - accrual4_step2
        reset_on_complete: False
        disable_on_complete: False
        enable_events: accrual4_enable
        disable_events: accrual4_disable
        reset_events: accrual4_reset
    accrual10:
        events:
            - accrual10_step1
            - accrual10_step2
```

(continues on next page)

(continued from previous page)

```
        reset_on_complete: True
        disable_on_complete: False
        enable_events: accrual10_enable
        disable_events: accrual10_disable
        reset_events: accrual10_reset
    accrual7:
        events:
          - accrual7_step1
          - accrual7_step2
          - accrual7_step3
        events_when_complete: accrual7_complete
        events_when_hit: accrual7_hit
        reset_on_complete: True
        disable_on_complete: False
        enable_events: accrual7_enable
        disable_events: accrual7_disable
        reset_events: accrual7_reset
        logic_block_timeout: 50
counters:
    counter1:
        count_events: counter1_count
        starting_count: 5
        count_complete_value: 0
        direction: down
        enable_events: counter1_enable
        disable_events: counter1_disable
        restart_events: counter1_restart
        reset_events: counter1_reset
    counter3:
        count_events: counter3_count
        starting_count: 0
        count_complete_value: 5
        count_interval: -1
        direction: up
        enable_events: counter3_enable
        disable_events: counter3_disable
        restart_events: counter3_restart
        reset_events: counter3_reset
        multiple_hit_window: 1s
    counter4:
        count_events: counter4_count
        starting_count: machine.start if machine.start else 0
        count_complete_value: current_player.hits
        direction: up
        enable_events: counter4_enable
        disable_events: counter4_disable
        restart_events: counter4_restart
        reset_events: counter4_reset
    counter5:
        count_events: counter5_count
    counter9:
        count_events: counter9_count
        starting_count: 5
        count_complete_value: 0
```

(continues on next page)

(continued from previous page)

```
        direction: down
        enable_events: counter9_enable
        disable_events: counter9_disable
        restart_events: counter9_restart
        reset_events: counter9_reset
        logic_block_timeout: 50
sequences:
    sequence1:
        events:
            - sequence1_step1a, sequence1_step1b
            - sequence1_step2a, sequence1_step2b
            - sequence1_step3a, sequence1_step3b
        events_when_complete: sequence1_complete
        enable_events: sequence1_enable
        disable_events: sequence1_disable
        reset_events: sequence1_reset
    sequence2:
        events:
            - sequence2_step1a, sequence2_step1b
            - sequence2_step2a, sequence2_step2b
            - sequence2_step3a, sequence2_step3b
        events_when_complete: sequence2_complete
        enable_events: sequence2_enable
        disable_events: sequence2_disable
        reset_events: sequence2_reset
        logic_block_timeout: 50

# logic blocks in mode1
modes:
    - mode1
    - mode2
    - mode3
    - mode4
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 217: your_machine_folder/logic_blocks/modes/mode2/config/mode2.yaml

```
#config_version=5
mode:
  start_events: start_mode2
  stop_events: stop_mode2

counters:
  counter_with_lights:
      count_events: counter_with_lights_count
      enable_events: counter_with_lights_enable
      starting_count: 0
```

(continues on next page)

(continued from previous page)

```
      count_complete_value: 3
      direction: up
      persist_state: True

show_player:
  logicblock_counter_with_lights_updated:
    counter_show:
      start_step: device.counters.counter_with_lights["value"] + 1

shows:
  counter_show:
    - duration: -1
      lights:
        led1: on
        led2: stop
        led3: stop
    - duration: -1
      lights:
        led1: stop
        led2: on
        led3: stop
    - duration: -1
      lights:
        led1: stop
        led2: stop
        led3: on
```

Listing 218: your_machine_folder/logic_blocks/modes/mode3/config/mode3.yaml

```
#config_version=5
mode:
  start_events: start_mode3
  stop_events: stop_mode3

counters:
  qualify1:
      count_events: qualify1_count, s_qualify1_active
      disable_events: disable_qualify
      enable_events: enable_qualify
      start_enabled: True
      events_when_complete: disable_qualify, qualify_start_mode1
      starting_count: 0
      count_complete_value: 3
      persist_state: True
      debug: True
  qualify2:
      count_events: qualify2_count, s_qualify2_active
      disable_events: disable_qualify
      enable_events: enable_qualify
      start_enabled: True
      events_when_complete: disable_qualify, qualify_start_mode2
      starting_count: 0
      count_complete_value: 3
      persist_state: True
```

(continues on next page)

(continued from previous page)

```
    debug: True
```

Listing 219: your_machine_folder/logic_blocks/modes/mode1/config/mode1.yaml

```
#config_version=5
mode:
  start_events: start_mode1
  stop_events: stop_mode1

counters:
  counter2:
      count_events: counter2_count
      events_when_hit: counter2_hit
      events_when_complete: counter2_complete
      starting_count: 0
      count_complete_value: 3
      direction: up
      reset_on_complete: True
      disable_on_complete: False
  counter_persist:
      count_events: counter_persist_count
      enable_events: counter_persist_enable
      direction: down
      starting_count: 5
      count_complete_value: 0
      persist_state: true
accruals:
  accrual5:
      events:
        - accrual5_step1
        - accrual5_step2
      persist_state: True
```

Listing 220: your_machine_folder/logic_blocks/modes/mode4/config/mode4.yaml

```
#config_version=5
mode:
  start_events: start_mode4
  stop_events: stop_mode4

counters:
  counter6:
      count_events: counter6_count
      events_when_hit: counter6_hit
      events_when_complete: counter6_complete
      starting_count: 0
      count_complete_value: 10
      direction: up
      reset_on_complete: True
      disable_on_complete: False
      control_events:
          - event: increase_counter6_5
            action: add
```

(continues on next page)

(continued from previous page)

```
          value: 5
        - event: increase_counter6_3
          action: add
          value: 3
        - event: increase_counter6_0
          action: add
          value: 0
        - event: reduce_counter6_5
          action: subtract
          value: 5
        - event: reduce_counter6_3
          action: subtract
          value: 3
        - event: reduce_counter6_0
          action: subtract
          value: 0
        - event: set_counter6_25
          action: jump
          value: 25
        - event: set_counter6_0
          action: jump
          value: 0
counter7:
    count_events: counter7_count
    events_when_hit: counter7_hit
    events_when_complete: counter7_complete
    starting_count: 5
    count_complete_value: 0
    direction: down
    reset_on_complete: True
    disable_on_complete: False
    control_events:
        - event: increase_counter7_5
          action: add
          value: 5
        - event: reduce_counter7_5
          action: subtract
          value: 5
        - event: reduce_counter7_3
          action: subtract
          value: 3
        - event: set_counter7_negative25
          action: jump
          value: -25
        - event: set_counter7_3
          action: jump
          value: 3
        - event: set_counter7_0
          action: jump
          value: 0
        - event: set_counter_placeholder
          action: jump
          value: machine.test2
        - event: subtract_counter_placeholder
```

(continues on next page)

(continued from previous page)

```
          action: subtract
          value: machine.test3
        - event: add_counter_placeholder
          action: add
          value: machine.test4

accruals:
  accrual6:
      events:
        - accrual6_step1
        - accrual6_step2
      persist_state: True
```

machine_vars (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 221: your_machine_folder/machine_vars/config/config.yaml

```
#config_version=5

machine_vars:
  test1:
    initial_value: 4
    value_type: int
    persist: True
  test2:
    initial_value: '5'
    value_type: str
    persist: True
  test3:
    initial_value: 6
    value_type: int
    persist: False

event_player:
  "{machine.time.second >= 30}": test_event3
  "{machine.time.second >= 40}": test_event4
```

magnet (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 222: your_machine_folder/magnet/config/config.yaml

```
#config_version=5

coils:
  magnet_coil1:
    number:
    default_pulse_ms: 100
    default_hold_power: 0.375
  magnet_coil2:
    number:
    default_pulse_ms: 100
    default_hold_power: 0.375
  magnet_coil3:
    number:
    default_pulse_ms: 100
    default_hold_power: 0.375

switches:
  grab_switch1:
    number:
  grab_switch2:
    number:
  grab_switch3:
    number:

magnets:
  magnet1:
    magnet_coil: magnet_coil1
    grab_switch: grab_switch1
    enable_events: magnet1_enable
    disable_events: magnet1_disable
    release_ball_events: magnet1_release
    fling_ball_events: magnet1_fling

  magnet_ball_save:
    magnet_coil: magnet_coil2
    grab_switch: grab_switch2
    enable_events: magnet_ball_save_enable
    disable_events: magnet_magnet_ball_save_grabbed_ball
    fling_ball_events: magnet_magnet_ball_save_grabbed_ball

  magnet_auto_enable:
    magnet_coil: magnet_coil3
    grab_switch: grab_switch3

ball_saves:
  magnet_save:
    balls_to_save: 1
    active_time: 5s
    enable_events: magnet_magnet_ball_save_grabbing_ball
```

match_mode (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 223: your_machine_folder/match_mode/config/config_highscore.yaml

```
#config_version=5

config:
  - config.yaml

modes:
  - high_score
  - service
```

Listing 224: your_machine_folder/match_mode/config/config.yaml

```
#config_version=5

game:
  balls_per_game: 1

credits:
  free_play: no
  events:
    - event: add_credit
      type: award
      credits: 1
    - event: match_has_match
      type: award
      credits: winners

modes:
  - match
  - credits
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 225: your_machine_folder/match_mode/modes/match/config/match.yaml

```
#config_version=5

slide_player:
  match_has_match: match
```

(continues on next page)

(continued from previous page)

```
  match_no_match: no_match

slides:
  match:
    - type: text
      text: asd
  no_match:
    - type: text
      text: asd
```

mma8451 (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 226: your_machine_folder/mma8451/config/config.yaml

```
#config_version=5

hardware:
  platform: virtual
  accelerometers: mma8451

accelerometers:
    test_accelerometer:
        number: 29
        level_x: 0
        level_y: 0
        level_z: 1
        platform: mma8451
        platform_settings:
          i2c_platform: virtual
```

mode_tests (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 227: your_machine_folder/mode_tests/config/test_empty_modes_section.yaml

```
#config_version=5

modes:
  # empty mode section
```

Listing 228: your_machine_folder/mode_tests/config/test_modes_in_game.yaml

```
#config_version=5

modes:
  - mode_restart_on_next_ball
```

Listing 229: your_machine_folder/mode_tests/config/test_loading_invalid_modes.yaml

```
#config_version=5

modes:
  - invalid
  - mode2
```

Listing 230: your_machine_folder/mode_tests/config/test_mode_without_config.yaml

```
#config_version=5

modes:
  - mode_without_config
```

Listing 231: your_machine_folder/mode_tests/config/test_modes.yaml

```
#config_version=5

modes:
  - mode1
  - mode2
  - mode3
  - mode4
```

Listing 232: your_machine_folder/mode_tests/config/test_missing_mode_section.yaml

```
#config_version=5

modes:
  - broken_mode2
  - mode2
```

Listing 233: your_machine_folder/mode_tests/config/test_broken_mode_config.yaml

```
#config_version=5

modes:
  - mode2
  - broken_mode
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 234: your_machine_folder/mode_tests/modes/mode2/config/mode2.yaml

```
#config_version=5

mode:
  stop_events: stop_mode2
  stop_priority: 2
  restart_on_next_ball: true
```

Listing 235: your_machine_folder/mode_tests/modes/mode_in_sub_folder/mode5/mode8/config/moc

```
#config_version=5
mode:
  start_events: start_mode8
  stop_events: stop_mode8
  priority: 200
  start_priority: 1
  stop_on_ball_end: false
  game_mode: False

mode_settings:
  this: true

config:
  - test.yaml
```

Listing 236: your_machine_folder/mode_tests/modes/mode_in_sub_folder/mode5/mode8/config/test

```
#config_version=5
mode_settings:
  test: 123
```

Listing 237: your_machine_folder/mode_tests/modes/mode_in_sub_folder/mode5/config/mode5.yam

```
#config_version=5
mode:
  start_events: start_mode5
```

(continues on next page)

(continued from previous page)

```
  stop_events: stop_mode5
  priority: 200
  start_priority: 1
  stop_on_ball_end: false
  game_mode: False

mode_settings:
  this: true

config:
  - test.yaml
```

Listing 238: your_machine_folder/mode_tests/modes/mode_in_sub_folder/mode5/config/test.yaml

```
#config_version=5
mode_settings:
  test: 123
```

Listing 239: your_machine_folder/mode_tests/modes/mode_in_sub_folder/sub_mode/mode7/config/t

```
#config_version=5
mode_settings:
  test: 123
```

Listing 240: your_machine_folder/mode_tests/modes/mode_in_sub_folder/sub_mode/mode7/config/r

```
#config_version=5
mode:
  start_events: start_mode7
  stop_events: stop_mode7
  priority: 200
  start_priority: 1
  stop_on_ball_end: false
  game_mode: False

mode_settings:
  this: true

config:
  - test.yaml
```

Listing 241: your_machine_folder/mode_tests/modes/mode_in_sub_folder/sub_mode/mode6/config/t

```
#config_version=5
mode_settings:
  test: 123
```

Listing 242: your_machine_folder/mode_tests/modes/mode_in_sub_folder/sub_mode/mode6/config/r

```
#config_version=5
mode:
  start_events: start_mode6
```

(continues on next page)

(continued from previous page)

```
  stop_events: stop_mode6
  priority: 200
  start_priority: 1
  stop_on_ball_end: false
  game_mode: False

mode_settings:
  this: true

config:
  - test.yaml
```

Listing 243: your_machine_folder/mode_tests/modes/broken_mode2/config/broken_mode2.yaml

```
#config_version=5

event_player:
  test: test2
```

Listing 244: your_machine_folder/mode_tests/modes/mode3/code/mode3.py

```
from mpf.core.mode import Mode


class Mode3(Mode):
    def mode_init(self):
        self.custom_code = True

    def mode_start(self, **kwargs):
        pass

    def mode_stop(self, **kwargs):
        pass
```

Listing 245: your_machine_folder/mode_tests/modes/mode3/config/mode3.yaml

```
#config_version=5

mode:
  code: mode3.Mode3
```

Listing 246: your_machine_folder/mode_tests/modes/mode_restart_on_next_ball/config/mode_resta

```
#config_version=5
mode:
  start_events: start_mode_restart_on_next_ball
  restart_on_next_ball: True
```

Listing 247: your_machine_folder/mode_tests/modes/mode1/config/mode1.yaml

```
#config_version=5
mode:
```

(continues on next page)

(continued from previous page)

```
  start_events: start_mode1
  stop_events: stop_mode1
  priority: 200
  start_priority: 1
  stop_on_ball_end: false
  game_mode: False

mode_settings:
  this: true

config:
  - test.yaml
```

Listing 248: your_machine_folder/mode_tests/modes/mode1/config/test.yaml

```
#config_version=5
mode_settings:
  test: 123
```

Listing 249: your_machine_folder/mode_tests/modes/mode4/config/mode4.yaml

```
#config_version=5
mode:
  start_events: start_mode4
  use_wait_queue: True
  game_mode: False
```

Listing 250: your_machine_folder/mode_tests/modes/broken_mode/config/broken_mode.yaml

```
#config_version=5

mode:
  invalid_key: crap
```

modes (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 251: your_machine_folder/modes/config/test_modes.yaml

```
#config_version=5
modes:
  - mode1
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 252: your_machine_folder/modes/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  priority: 300
```

motor (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 253: your_machine_folder/motor/config/multiposition_motor_home_in_the_middle.yaml

```
#config_version=5

switches:
    s_multiposition_motor_1:
        number:
    s_multiposition_motor_2:
        number:
    s_multiposition_motor_3:
        number:
    s_multiposition_motor_4:
        number:

digital_outputs:
    c_multiposition_motor_left:
        number:
        type: driver
    c_multiposition_motor_right:
        number:
        type: driver

motors:
```

(continues on next page)

(continued from previous page)

```
    multiposition_motor2:
        debug: True
        motor_left_output: c_multiposition_motor_left
        motor_right_output: c_multiposition_motor_right
        position_switches:  !!omap
            - position1: s_multiposition_motor_1
            - position2: s_multiposition_motor_2
            - position3: s_multiposition_motor_3
            - position4: s_multiposition_motor_4
        reset_position: position2
        go_to_position:
            goto_position1: position1
            goto_position2: position2
            goto_position3: position3
            goto_position4: position4
```

Listing 254: your_machine_folder/motor/config/drop_target.yaml

```
#config_version=5

switches:
    s_position_up:
        number:
    s_position_down:
        number:

digital_outputs:
    c_motor_run:
        number:
        type: driver

motors:
    motorized_drop_target_bank:
        debug: True
        motor_left_output: c_motor_run
        position_switches:  !!omap
            - up: s_position_up
            - down: s_position_down
        reset_position: down
        go_to_position:
            go_up: up
            go_down: down
            go_down2: down
```

Listing 255: your_machine_folder/motor/config/ghostbusters.yaml

```
#config_version=5

switches:
    s_slimer_home:
        number: 8-1
    s_slimer_away:
        number: 8-2
```

(continues on next page)

(continued from previous page)

```
digital_outputs:
    c_slimer_motor_forward:
        number: 8-3
        type: light
    c_slimer_motor_backward:
        number: 8-4
        type: light

motors:
    ghostbusters_slimer:
        debug: True
        motor_left_output: c_slimer_motor_forward
        motor_right_output: c_slimer_motor_backward
        position_switches:  !!omap
            - home: s_slimer_home
            - away: s_slimer_away
        reset_position: home
        go_to_position:
            slimer_home: home
            slimer_away: away
```

Listing 256: your_machine_folder/motor/config/multiposition_motor.yaml

```
#config_version=5

switches:
    s_multiposition_motor_1:
        number:
    s_multiposition_motor_2:
        number:
    s_multiposition_motor_3:
        number:
    s_multiposition_motor_4:
        number:

digital_outputs:
    c_multiposition_motor_left:
        number:
        type: driver
    c_multiposition_motor_right:
        number:
        type: driver

motors:
    multiposition_motor:
        debug: True
        motor_left_output: c_multiposition_motor_left
        motor_right_output: c_multiposition_motor_right
        position_switches:  !!omap
            - position1: s_multiposition_motor_1
            - position2: s_multiposition_motor_2
            - position3: s_multiposition_motor_3
            - position4: s_multiposition_motor_4
```

(continues on next page)

(continued from previous page)

```
        reset_position: position4
        go_to_position:
            goto_position1: position1
            goto_position2: position2
            goto_position3: position3
            goto_position4: position4
```

Listing 257: your_machine_folder/motor/config/multiposition_motor_start_on_end_switch.yaml

```
#config_version=5

switches:
    s_multiposition_motor_1:
        number:
    s_multiposition_motor_2:
        number:
    s_multiposition_motor_3:
        number:
    s_multiposition_motor_4:
        number:

digital_outputs:
    c_multiposition_motor_left:
        number:
        type: driver
    c_multiposition_motor_right:
        number:
        type: driver

motors:
    multiposition_motor:
        debug: True
        motor_left_output: c_multiposition_motor_left
        motor_right_output: c_multiposition_motor_right
        position_switches:  !!omap
            - position1: s_multiposition_motor_1
            - position2: s_multiposition_motor_2
            - position3: s_multiposition_motor_3
            - position4: s_multiposition_motor_4
        reset_position: position2
        go_to_position:
            goto_position1: position1
            goto_position2: position2
            goto_position3: position3
            goto_position4: position4

virtual_platform_start_active_switches: s_multiposition_motor_4
```

mpf_plugin_config_player_validation (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 258: your_machine_folder/mpf_plugin_config_player_validation/config/mpf_plugin_validatio

```
#config_version=5

show_player:
  event1: show1
```

Show file examples

Here are some example show files that go along with the above config(s).

Listing 259: your_machine_folder/mpf_plugin_config_player_validation/shows/show1.yaml

```
#show_version=5
- time: 0
  slides:
    slide1:  # device
      type: text  # device_settings
      text: TEST 1
      color: ff0000
      font_size: 100
- time: 1
  slides:
    slide_7:  # device
      - type: text  # device_settings
        text: TEXT FROM SLIDE_PLAYER LIST
        color: red
        font_size: 15
        y: 66%
      - type: text
        text: WIDGET 2
        color: purple
        font_size: 15
        y: 33%
- time: 2
  slides:
    slide_8:  # device
      widgets:  # device_settings
      - type: text
        text: TEXT FROM SLIDE_PLAYER WIDGET LIST
        color: green
        font_size: 15
        y: 66%
      - type: text
        text: WIDGET 2
        color: lime
        font_size: 15
        y: 33%
      target: display1
```

(continues on next page)

(continued from previous page)

```
      transition: move_in
- time: 3
  slides: slide2
- time: 4
  slides:
    slide_9:
      widgets:  # device_settings
      - type: text
        text: TEXT FROM SLIDE_PLAYER WIDGET LIST
        color: green
        font_size: 15
        y: 66%
      - type: text
        text: WIDGET 2
        color: lime
        font_size: 15
        y: 33%
      target: display1
      transition: move_in
    slide_10:
      widgets:  # device_settings
      - type: text
        text: TEXT FROM SLIDE_PLAYER WIDGET LIST
        color: green
        font_size: 15
        y: 66%
      - type: text
        text: WIDGET 2
        color: lime
        font_size: 15
        y: 33%
      target: dmd
      transition: move_in
```

mpftestcase (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 260: your_machine_folder/mpftestcase/config/test_mpftestcase.yaml

```
#config_version=5

switches:
  switch1:
    number:
```

multiball (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 261: your_machine_folder/multiball/config/config.yaml

```
#config_version=5

game:
    balls_per_game: 1

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:
    eject_coil3:
        number:

event_player:
    test_event_when_enabled:
        - should_post_when_enabled{device.multiballs.mb1.enabled}
        - should_not_post_when_enabled{not device.multiballs.mb1.enabled}
    test_event_when_disabled:
        - should_post_when_disabled{not device.multiballs.mb1.enabled}
        - should_not_post_when_disabled{device.multiballs.mb1.enabled}

switches:
    s_start:
        number:
        tags: start
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch3:
        number:
    s_ball_switch4:
        number:
    s_ball_switch5:
        number:
    s_ball_switch6:
        number:
    s_lock1:
        number:
    s_lock2:
        number:
    s_ball_switch_launcher:
        number:

playfields:
```

(continues on next page)

(continued from previous page)

```
    playfield:
        default_source_device: bd_launcher
        tags: default

ball_devices:
    bd_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2, s_ball_switch3, s_ball_switch4, s_ball_switch5,␣
↪s_ball_switch6
        confirm_eject_type: target
        eject_targets: bd_launcher
        tags: trough, drain, home
    bd_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
        confirm_eject_type: target
        eject_timeouts: 2s
    bd_lock:
        eject_coil: eject_coil3
        ball_switches: s_lock1, s_lock2
        eject_timeouts: 2s

modes:
    - mode1
    - mode2
    - mode3
    - mode4
    - mode5

multiballs:
    mb1:
        ball_count: 1
        ball_count_type: add
        shoot_again: 30s
        enable_events: mb1_enable
        disable_events: mb1_disable
        start_events: mb1_start
        stop_events: mb1_stop
    mb2:
        ball_count: 2
        ball_count_type: add
        shoot_again: -1
        enable_events: mb2_enable
        disable_events: mb2_disable
        start_events: mb2_start
        stop_events: mb2_stop
    mb3:
        ball_count: 1
        ball_count_type: add
        shoot_again: 0
        enable_events: mb3_enable
        disable_events: mb3_disable
        start_events: mb3_start
        stop_events: mb3_stop
```

(continues on next page)

(continued from previous page)

```
    mb10:
        ball_count: 3
        ball_count_type: total
        shoot_again: 20s
        start_events: mb10_start
    mb_add_a_ball:
        ball_count: 2
        start_or_add_a_ball_events: start_or_add
        add_a_ball_events: add_ball
    mb_placeholder:
        ball_count: 2
        shoot_again: machine.shoot_again_sec * 1000
        start_events: mb_placeholder_start
        stop_events: mb_placeholder_stop
    mb_alltimers:
        ball_count: 2
        shoot_again: 30s
        hurry_up_time: 10s
        grace_period: 5s
        start_events: mb_alltimers_start
        stop_events: mb_alltimers_stop
    mb_add_a_ball_timers:
        ball_count: 2
        shoot_again: 30s
        hurry_up_time: 10s
        grace_period: 5s
        add_a_ball_events: add_ball
        add_a_ball_shoot_again: 20s
        add_a_ball_hurry_up_time: 5s
        add_a_ball_grace_period: 10s
        start_events: mb_add_a_ball_timers_start
        stop_events: mb_add_a_ball_timers_stop
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 262: your_machine_folder/multiball/modes/mode2/config/mode2.yaml

```
#config_version=5
mode:
  start_events: start_mode2
  stop_events: stop_mode2

multiballs:
    mb5:
        ball_count: 1
        ball_count_type: add
        start_events: mb5_start
```

Listing 263: your_machine_folder/multiball/modes/mode3/config/mode3.yaml

```
#config_version=5
mode:
  start_events: start_mode3
  stop_events: stop_mode3

multiballs:
    mb_autostart:
        ball_count: 2
        start_events: mode_mode3_started
```

Listing 264: your_machine_folder/multiball/modes/mode5/config/mode5.yaml

```
#config_version=5
mode:
  start_events: start_mode5
  stop_events: stop_mode5

multiballs:
    mb_mode5:
        ball_count: 2
        shoot_again: 30s
        hurry_up_time: 10s
        grace_period: 5s
        start_events: mb_mode5_start
        stop_events: mb_mode5_stop

    mb_mode5_lean:
        ball_count: 2
        shoot_again: 30s
        start_events: mb_mode5_lean_start
        stop_events: mb_mode5_lean_stop
```

Listing 265: your_machine_folder/multiball/modes/mode1/config/mode1.yaml

```
#config_version=5
mode:
  start_events: start_mode1
  stop_events: stop_mode1

multiballs:
    mb4:
        ball_count: 1
        ball_count_type: add
        shoot_again: 30s
        enable_events: mb4_enable
        disable_events: mb4_disable
        start_events: mb4_start
        stop_events: mb4_stop


    mb11:
        ball_count: 2
```

(continues on next page)

(continued from previous page)

```
        ball_count_type: total
        shoot_again: 20s
        start_events: mb11_start
        ball_locks: bd_lock

    mb12:
        ball_count: current_player.lock_mb6_locked_balls
        ball_count_type: add
        shoot_again: 20s
        start_events: mb12_start
        ball_locks: bd_lock

    mb6:
        ball_count: 2
        ball_count_type: add
        shoot_again: 0
        start_events: mb6_start
        ball_locks: bd_lock

multiball_locks:
    lock_mb6:
        lock_devices: bd_lock
        balls_to_lock: 2
        reset_count_for_current_player_events: mb6_start
        disable_events: mb6_start
```

Listing 266: your_machine_folder/multiball/modes/mode4/config/mode4.yaml

```
#config_version=5
mode:
  start_events: start_mode4
  stop_events: stop_mode4

multiballs:
    mb4_autostart:
        ball_count: 2
        ball_count_type: total
        shoot_again: 0s
        start_events: multiball_lock_lock_mb_autostart_full
        ball_locks: bd_lock


multiball_locks:
    lock_mb_autostart:
        lock_devices: bd_lock
        balls_to_lock: 1
```

multiball_locks (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 267: your_machine_folder/multiball_locks/config/testPhysicalOnly.yaml

```
#config_version=5
config: config.yaml

modes:
    - physical_only
```

Listing 268: your_machine_folder/multiball_locks/config/testVirtualOnly.yaml

```
#config_version=5
config: config.yaml

modes:
    - virtual_only
```

Listing 269: your_machine_folder/multiball_locks/config/config.yaml

```
#config_version=5

game:
    balls_per_game: 2

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:
    eject_coil3:
        number:
    eject_coil4:
        number:

switches:
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch3:
        number:
    s_ball_switch4:
        number:
    s_ball_switch5:
        number:
    s_ball_switch6:
        number:
    s_lock1:
```

(continues on next page)

(continued from previous page)

```
        number:
    s_lock2:
        number:
    s_lockt1:
        number:
    s_lockt2:
        number:
    s_lockt3:
        number:
    s_lockb1:
        number:
    s_lockb2:
        number:

playfields:
    playfield:
        default_source_device: bd_trough
        tags: default

ball_devices:
    bd_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2, s_ball_switch3, s_ball_switch4, s_ball_switch5,␣
↪s_ball_switch6
        tags: trough, drain, home
        eject_timeouts: 2s
    bd_lock:
        eject_coil: eject_coil2
        ball_switches: s_lock1, s_lock2
        eject_timeouts: 2s
    bd_lock_triple:
        eject_coil: eject_coil3
        ball_switches: s_lockt1, s_lockt2, s_lockt3
        eject_timeouts: 2s
    bd_lock_block:
        eject_coil: eject_coil4
        ball_switches: s_lockb1, s_lockb2
        eject_timeouts: 2s

multiballs:
    mb:
        ball_count: 2
        shoot_again: 0
        start_events: mb_start
        ball_locks: bd_lock
```

Listing 270: your_machine_folder/multiball_locks/config/testDefault.yaml

```
#config_version=5
config: config.yaml

modes:
    - default
    - blocking
```

Listing 271: your_machine_folder/multiball_locks/config/testSourceDevices.yaml

```
#config_version=5
config: config.yaml

modes:
    - source_devices
```

Listing 272: your_machine_folder/multiball_locks/config/testMinVirtualPhysical.yaml

```
#config_version=5
config: config.yaml

modes:
    - min_virtual_physical
```

Listing 273: your_machine_folder/multiball_locks/config/testNoVirtual.yaml

```
#config_version=5
config: config.yaml

modes:
    - no_virtual
```

Listing 274: your_machine_folder/multiball_locks/config/testPhysicalOnlyNoStealing.yaml

```
#config_version=5
config: config.yaml

modes:
    - physical_only_no_stealing
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 275: your_machine_folder/multiball_locks/modes/default/config/default.yaml

```
#config_version=5
mode:
  start_events: start_default

event_player:
  test_event_when_enabled:
    - should_post_when_enabled{device.multiball_locks.lock_default.enabled}
    - should_not_post_when_enabled{not device.multiball_locks.lock_default.enabled}
  test_event_when_disabled:
    - should_post_when_disabled{not device.multiball_locks.lock_default.enabled}
    - should_not_post_when_disabled{device.multiball_locks.lock_default.enabled}
```

(continues on next page)

(continued from previous page)

```
multiball_locks:
  lock_default:
    lock_devices: bd_lock
    balls_to_lock: 2
    locked_ball_counting_strategy: virtual_only
  lock_triple:
    lock_devices: bd_lock_triple
    balls_to_lock: 3
    locked_ball_counting_strategy: virtual_only
  lock_with_block:
    lock_devices: bd_lock_block
    balls_to_lock: 2
    locked_ball_counting_strategy: virtual_only
    blocking_facility: foo
```

Listing 276: your_machine_folder/multiball_locks/modes/virtual_only/config/virtual_only.yaml

```
#config_version=5
mode:
  start_events: start_virtual_only

multiball_locks:
    lock_virtual_only:
        lock_devices: bd_lock
        balls_to_lock: 2
        locked_ball_counting_strategy: virtual_only
        debug: True
```

Listing 277: your_machine_folder/multiball_locks/modes/source_devices/config/source_devices.yaml

```
#config_version=5
mode:
  start_events: start_source_devices

multiball_locks:
  lock2:
    lock_devices: bd_lock
    source_devices: bd_lock_triple
    balls_to_lock: 2
    locked_ball_counting_strategy: virtual_only
  lock1:
    lock_devices: bd_lock_triple
    balls_to_lock: 2
    locked_ball_counting_strategy: virtual_only
```

Listing 278: your_machine_folder/multiball_locks/modes/blocking/config/blocking.yaml

```
#config_version=5
mode:
  start_events: start_blocking
  priority: 1000
```

(continues on next page)

(continued from previous page)

```
blocking:
  balldevice_bd_lock_block_ball_enter:
    foo: 102
```

Listing 279: your_machine_folder/multiball_locks/modes/physical_only_no_stealing/config/physical_

```
#config_version=5
mode:
  start_events: start_physical_only

multiball_locks:
    lock_physical_only:
        lock_devices: bd_lock
        balls_to_lock: 2
        locked_ball_counting_strategy: physical_only
        empty_lock_devices_on_ball_end: true
        debug: True
```

Listing 280: your_machine_folder/multiball_locks/modes/physical_only/config/physical_only.yaml

```
#config_version=5
mode:
  start_events: start_physical_only

multiball_locks:
    lock_physical_only:
        lock_devices: bd_lock
        balls_to_lock: 2
        locked_ball_counting_strategy: physical_only
        debug: True
    lock_physical_only_smaller_than_device:
        lock_devices: bd_lock_triple
        balls_to_lock: 2
        locked_ball_counting_strategy: physical_only
        debug: True
```

Listing 281: your_machine_folder/multiball_locks/modes/min_virtual_physical/config/min_virtual_ph

```
#config_version=5
mode:
  start_events: start_min_virtual_physical

multiball_locks:
    lock_min_virtual_physical:
        lock_devices: bd_lock
        balls_to_lock: 2
        locked_ball_counting_strategy: min_virtual_physical
        debug: True
```

Listing 282: your_machine_folder/multiball_locks/modes/no_virtual/config/no_virtual.yaml

```
#config_version=5
mode:
```

(continues on next page)

(continued from previous page)

```
  start_events: start_no_virtual

multiball_locks:
    lock_no_virtual:
        lock_devices: bd_lock
        balls_to_lock: 2
        locked_ball_counting_strategy: no_virtual
        debug: True
```

mypinballs (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 283: your_machine_folder/mypinballs/config/config.yaml

```
#config_version=5

hardware:
  segment_displays: mypinballs

mypinballs:
  port: /dev/ttyUSB0
  debug: True

segment_displays:
  display1:
    number: 1
  display2:
    number: 2
  display6:
    number: 6
```

null (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 284: your_machine_folder/null/config/null.yaml

```
#config_version=5
```

openpixel (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 285: your_machine_folder/openpixel/config/config.yaml

```
#config_version=5

lights:
  test_led:
    number: 99
    type: grb
  test_led2:
    number: 0-20
    type: grb
  test_led3:
    number: 1-99
    type: grb
```

Listing 286: your_machine_folder/openpixel/config/fadecandy.yaml

```
#config_version=5

config:
- config.yaml

lights:
  test_rgbw:
    channels:
      red:
        - number: 2-0
      green:
        - number: 2-1
      blue:
        - number: 2-2
      white:
        - number: 2-3
  test_rgbw2:
    channels:
      red:
        - number: 2-4
```

(continues on next page)

(continued from previous page)

```
      green:
        - number: 2-5
      blue:
        - number: 2-6
      white:
        - number: 2-7
  test_led_serial:
    previous: test_led2
    type: rgb
  test_led_serial2:
    previous: test_led_serial
    type: rgbw
```

opp (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 287: your_machine_folder/opp/config/config.yaml

```
#config_version=5

hardware:
    platform: opp

opp:
    ports: com1
    baud: 115200
    debug: True

switches:
    s_test:
        number: 0-0
    s_test_no_debounce:
        number: 0-1
        debounce: quick
    s_test_nc:
        number: 0-2
        type: 'NC'
    s_flipper:
        number: 0-3
    s_test_card2:
        number: 0-8
    s_test_neo:
        number: 1-0
```

(continues on next page)

(continued from previous page)

```
coils:
    c_test:
        number: 0-0
        default_pulse_ms: 23
    c_test_allow_enable:
        number: 0-1
        default_pulse_ms: 23
        platform_settings:
            recycle_factor: 3
        default_hold_power: 1.0
    c_flipper_hold:
        number: 0-2
        default_hold_power: 1.0
    c_flipper_main:
        number: 0-3
        default_pulse_ms: 10
        default_hold_power: 0.375
    c_holdpower_16:
        number: 1-12
        default_hold_power: 0.0625

lights:
  test_light1:
    number: 0-16
    subtype: matrix
  test_light2:
    number: 0-17
    subtype: matrix
  test_led1:
    number: 1-0
  test_led2:
    previous: test_led1
    type: rgb

autofire_coils:
    ac_slingshot_test:
        coil: c_test
        switch: s_test

    ac_slingshot_test2:
        coil: c_test_allow_enable
        switch: s_test_no_debounce

    ac_delayed_kickback:
        coil: c_test
        switch: s_test
        coil_pulse_delay: 20

flippers:
    f_test_single:
        debug: true
        #main_coil_overwrite:
        #    pulse_ms: 11
        main_coil: c_flipper_main
```

(continues on next page)

(continued from previous page)

```
        activation_switch: s_flipper

    f_test_hold:
        debug: true
        main_coil: c_flipper_main
        hold_coil: c_flipper_hold
        activation_switch: s_flipper
```

Listing 288: your_machine_folder/opp/config/config2.yaml

```
#config_version=5

hardware:
    platform: opp

opp:
    ports: com1
    baud: 115200
    debug: True

switches:
    s_test:
        number: 0-0
    s_test_no_debounce:
        number: 0-1
        debounce: quick
    s_test_nc:
        number: 0-2
        type: 'NC'
    s_flipper:
        number: 0-3
    s_test_card2:
        number: 0-8
    s_matrix_test:
        number: 3-48
    s_matrix_test2:
        number: 3-32
    s_matrix_test3:
        number: 3-95

coils:
    c_test:
        number: 0-0
        default_pulse_ms: 23
    c_test_allow_enable:
        number: 0-1
        default_pulse_ms: 23
        platform_settings:
            recycle_factor: 3
        default_hold_power: 1.0
    c_flipper_hold:
        number: 0-2
        default_hold_power: 1.0
    c_flipper_main:
```

(continues on next page)

(continued from previous page)

```
        number: 0-3
        default_pulse_ms: 10
        default_hold_power: 0.375
    c_holdpower_16:
        number: 1-12
        default_hold_power: 0.0625
    c_matrix_test:
        number: 3-0
        default_pulse_ms: 42

lights:
  test_light1:
    number: 0-16
    subtype: matrix
  test_light2:
    number: 0-17
    subtype: matrix
  test_led1:
    number: 1-0
  test_led2:
    number: 1-1

autofire_coils:
    ac_slingshot_test:
        coil: c_test
        switch: s_test

    ac_slingshot_test2:
        coil: c_test_allow_enable
        switch: s_test_no_debounce

    ac_matrix_slingshot_test:
        coil: c_matrix_test
        switch: s_matrix_test

flippers:
    f_test_single:
        debug: true
        main_coil: c_flipper_main
        activation_switch: s_flipper

    f_test_hold:
        debug: true
        main_coil: c_flipper_main
        hold_coil: c_flipper_hold
        activation_switch: s_flipper
```

Listing 289: your_machine_folder/opp/config/config_stm32.yaml

```
#config_version=5

hardware:
    platform: opp
```

(continues on next page)

(continued from previous page)

```
opp:
    ports: com1, com2
    baud: 115200
    debug: True

switches:
    s_test:
        number: 19088743-0-0
    s_test_no_debounce:
        number: 19088743-0-1
        debounce: quick

lights:
  l2-0:
    number: 2-0-16
    subtype: incand
  l2-1:
    number: 2-0-17
    subtype: incand
  l2-2:
    number: 2-0-18
    subtype: incand
  l2-3:
    number: 2-0-19
    subtype: incand
  l2-4:
    number: 2-0-20
    subtype: incand
  l2-5:
    number: 2-0-21
    subtype: incand
  l2-6:
    number: 2-0-22
    subtype: incand
  l2-7:
    number: 2-0-23
    subtype: incand
  l3-7:
    number: 2-0-31
    subtype: incand
  l_neo_0:
    number: 19088743-0-0
    subtype: led
    type: rgb
  l_neo_1:
    previous: l_neo_0
    type: rgb
    subtype: led
  m0-0:
    number: 2-0-0
    subtype: matrix
  m0-1:
    number: 2-0-1
    subtype: matrix
```

(continues on next page)

(continued from previous page)

```
  m0-63:
    number: 2-0-63
    subtype: matrix

servos:
  servo1:
    servo_min: 0
    servo_max: 1
    speed_limit: 20
    positions:
      0.392: servo_up
      0.784: servo_down
    reset_position: 0.588
    reset_events: reset_servo
    number: 19088743-0-8
```

osc (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 290: your_machine_folder/osc/config/config.yaml

```
#config_version=5

hardware:
    platform: osc

osc:
  remote_ip: 127.0.0.1
  remote_port: 8000

  events_to_send:
    - my_test_event
    - my_other_test_event
    - player_turn_started

lights:
  test_light1:
    channels:
      red:
        - number: light1/red
      blue:
        - number: light1/blue
      green:
        - number: light1/green
  test_light2:
    number: light2
```

(continues on next page)

(continued from previous page)

```
switches:
  switch_1:
    number: 1
    type: NO
  switch_abc:
    number: abc
```

p3_roc (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 291: your_machine_folder/p3_roc/config/config.yaml

```
#config_version=5

hardware:
    platform: p3_roc
    servo_controllers: i2c_servo_controller
    driverboards: pdb

p_roc:
    use_separate_thread: False
    trace_bus: True
    debug: true
    pd_led_boards:
      2:
        use_servo_0: True
      4:
        use_stepper_0: True
        use_stepper_1: True
    gpio_map:
      0: input
      1: output
      2: output
      3: input
      5: output
      7: input

digital_outputs:
    d_gpio1:
      number: gpio-1
      type: driver
    d_gpio5:
      number: gpio-5
      type: driver

switches:
```

(continues on next page)

(continued from previous page)

```
    s_test_000:
        number: A0-B0-0
    s_test_001:
        number: 0/0/3
    s_test:
        number: A1-B0-7
    s_test_no_debounce:
        number: A1-B1-0
        debounce: quick
    s_slingshot_test:
        number: A2-B1-0
    s_test_nc:
        number: A2-B1-1
        type: 'NC'
    s_flipper:
        number: 1
    s_flipper_eos:
        number: 2
    s_stepper1_home:
        number: A4-B0-0
    s_stepper2_home:
        number: A4-B0-1
    s_sling_default:  # just defaults
        number: A4-B0-2
    s_gpio0:
        number: gpio-0
    s_gpio7:
        number: gpio-7

coils:
    c_test:
        number: A1-B1-2
        default_pulse_ms: 23
    c_test_allow_enable:
        number: A1-B1-3
        default_pulse_ms: 23
        default_hold_power: 1.0
    c_slingshot_test:
        number: A0-B1-0
    c_coil_pwm_test:
        number: A0-B1-1
        default_hold_power: 0.2
    c_flipper_main:
        number: A0-B0-1
        default_pulse_ms: 10
        default_hold_power: 0.375
    c_flipper_hold:
        number: A0-B0-2
        default_hold_power: 0.125
    test_gi:
        number: A2-B0-3
        default_hold_power: 1.0
        default_pulse_ms: 0
    c_sling_pulse_power:  # just defaults
```

(continues on next page)

(continued from previous page)

```
        number: A2-B0-4
        default_pulse_power: 0.5
        default_pulse_ms: 12

    # with those two coils we test that we also configure the opposite bank on the PD-16
    # do not configure other coils on A5 or A6 in this test
    c_bank_test:
        number: A5-B0-2
    c_bank2_test:
        number: A6-B1-7


autofire_coils:
    ac_slingshot_test:
        coil: c_slingshot_test
        switch: s_slingshot_test
    ac_switch_nc_test:
        coil: c_coil_pwm_test
        switch: s_test_nc
    ac_sling_pulse_power:
        coil: c_sling_pulse_power
        switch: s_sling_default

servos:
    servo1:
        number: 3
    servo_pd_led_0:
      platform: p3_roc
      number: 2-0

accelerometers:
    p3_roc_accelerometer:
        number: 1

flippers:
    f_test_single:
        debug: true
        main_coil_overwrite:
            pulse_ms: 11
        main_coil: c_flipper_main
        activation_switch: s_flipper

    f_test_hold:
        debug: true
        main_coil: c_flipper_main
        hold_coil: c_flipper_hold
        activation_switch: s_flipper

    f_test_hold_eos:
        debug: true
        main_coil: c_flipper_main
        hold_coil: c_flipper_hold
        activation_switch: s_flipper
        eos_switch: s_flipper_eos
```

(continues on next page)

(continued from previous page)

```
        use_eos: true

    f_test_single_eos:
        debug: true
        main_coil: c_flipper_main
        activation_switch: s_flipper
        eos_switch: s_flipper_eos
        use_eos: true

lights:
  test_pdb_light:
    number: C-A2-B0-0:R-A2-B1-0
    subtype: matrix
  test_gi:
    platform: drivers
    number: test_gi
  test_led:
    number: 2-1-2-3
  test_led2:
    channels:
      red:
        number: 2-7
      green:
        number: 2-8
      blue:
        number: 2-9
  test_led3:
    previous: test_led2
    type: rgb
  test_led_inverted:
    number: 2-4-5-6
    platform_settings:
      polarity: True
    subtype: led

steppers:
  stepper1:
    number: 4-0
    debug: True
    homing_mode: switch
    homing_switch: s_stepper1_home
  stepper2:
    number: 4-1
    debug: True
    homing_mode: switch
    homing_switch: s_stepper2_home
```

p_roc (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 292: your_machine_folder/p_roc/config/config.yaml

```
#config_version=5

hardware:
    platform: p_roc

p_roc:
  driverboards: pdb
  use_separate_thread: False
  dmd_timing_cycles: 1, 2, 3, 4
  debug: true
  trace_bus: True

switches:
    s_test_000:
        number: 0
    s_test_001:
        number: 2
    s_test:
        number: 23
    s_test_no_debounce:
        number: 24
        debounce: quick
    s_slingshot_test:
        number: 40
    s_direct:
        number: SD01
    s_matrix:
        number: 2/3


coils:
    c_test:
        number: A1-B1-2
        default_pulse_ms: 23
    c_test_allow_enable:
        number: A1-B1-3
        default_pulse_ms: 23
        default_hold_power: 1.0
    c_slingshot_test:
        number: A0-B1-0
    c_test2:  # unused. just to configure bank 0
        number: A0-B0-0
    c_direct:
        number: C01
    test_gi:
```

(continues on next page)

(continued from previous page)

```
        number: A2-B0-3
        default_hold_power: 1.0
        default_pulse_ms: 0
    c_direct2_pulse_power:
        number: C02
        default_pulse_power: 0.9
        default_pulse_ms: 20

autofire_coils:
    ac_slingshot_test:
        coil: c_slingshot_test
        switch: s_slingshot_test

lights:
  test_pdb_light:
    number: C-A2-B0-0:R-A2-B1-0
    subtype: matrix
  test_direct_light:
    number: L01
  test_gi:
    platform: drivers
    number: test_gi

segment_displays:
  display1:
    number: 0
```

Listing 293: your_machine_folder/p_roc/config/snux.yaml

```
#config_version=5

hardware:
    coils: snux
    switches: snux
    platform: p_roc

p_roc:
    driverboards: wpc
    use_separate_thread: False
    trace_bus: True
    debug: true

system11:
    ac_relay_delay_ms: 75
    ac_relay_driver: c_ac_relay
    platform: p_roc

snux:
    diag_led_driver: c_diag_led_driver

switches:
    s_test_fliptronics:
        number: sf1
    s_test_direct:
```

(continues on next page)

(continued from previous page)

```
        number: sd1
    s_test_matrix:
        number: s26

coils:
    c_test_direct:
        number: c01
    c_test_a_side:
        number: c02a
    c_test_c_side:
        number: c02c
        default_hold_power: 1.0
    c_flipper_enable_driver:
        number: c23
        default_hold_power: 1.0
    c_diag_led_driver:
        number: c24
        default_hold_power: 1.0
    c_ac_relay:
        number: c25
        default_hold_power: 1.0

autofire_coils:
    ac_slingshot_test:
        coil: c_test_direct
        switch: s_test_direct
```

Listing 294: your_machine_folder/p_roc/config/wpc.yaml

```
#config_version=5

hardware:
    platform: p_roc

p_roc:
    driverboards: wpc
    use_separate_thread: False
    trace_bus: True
    debug: true

switches:
    s_test_fliptronics:
        number: sf1
    s_test_direct:
        number: sd1
    s_test_matrix:
        number: s26
    s_slingshot_test:
        number: s20

coils:
    c_test_direct:
        number: c01
        default_pulse_ms: 23
```

(continues on next page)

(continued from previous page)

```
    c_test_fliptronics:
        number: fllm
        default_pulse_ms: 23
    test_gi:
        number: g01
        default_pulse_ms: 0
    c_slingshot_test:
        number: c02
lights:
  test_light:
    number: l11
    subtype: matrix
  test_gi:
    platform: drivers
    number: test_gi

autofire_coils:
    ac_slingshot_test:
        coil: c_slingshot_test
        switch: s_slingshot_test
```

pkone (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 295: your_machine_folder/pkone/config/config.yaml

```
#config_version=5
# Hardware setup for tests: Extension boards at addresses 0 and 1, Lightshow boards at 2 (rgb) and 3␣
↪(rgbw)

hardware:
    platform: pkone

pkone:
    port: com3
    debug: true

switches:
    s_test:
        number: 0-7
    s_test_nc:
        number: 0-26
        type: 'NC'
    s_slingshot_test:
        number: 0-22
    s_flipper:
        number: 1-5
```

(continues on next page)

(continued from previous page)

```
    s_flipper_eos:
        number: 1-6
    s_autofire:
        number: 1-7
    s_up:
        number: 1-11
    s_down:
        number: 1-12
    s_test_1:
        number: 0-1
    s_test_2:
        number: 0-2
    s_test_3:
        number: 0-3
    s_test_4:
        number: 0-4
    s_test_11:
        number: 1-1
    s_test_12:
        number: 1-2
    s_test_13:
        number: 1-3
    s_test_14:
        number: 1-4

coils:
    c_test:
        number: 1-4
        default_pulse_ms: 23
        default_recycle: True
        platform_settings:
            recycle_ms: 27
    c_test_allow_enable:
        number: 1-6
        default_pulse_ms: 23
        max_hold_power: 1.0
    c_slingshot_test:
        number: 0-7
    c_long_pulse:
        number: 1-8
        default_pulse_ms: 2000
        max_hold_power: 1.0
    c_flipper_main:
        number: 1-1
        default_pulse_ms: 10
        default_hold_power: 0.125
    c_flipper_hold:
        number: 1-2
        default_hold_power: 0.125

autofire_coils:
    ac_slingshot_test:
        coil: c_slingshot_test
        switch: s_slingshot_test
```

(continues on next page)

(continued from previous page)

```
    ac_inverted_switch:
        coil: c_slingshot_test
        switch: s_test_nc
    ac_same_switch1:
        coil: c_test
        switch: s_autofire
        enable_events: ac_same_switch
    ac_same_switch2:
        coil: c_test_allow_enable
        switch: s_autofire
        enable_events: ac_same_switch
    ac_different_boards:
        coil: c_flipper_hold
        switch: s_test
    ac_board_3:
        coil: c_flipper_hold
        switch: s_test_13

flippers:
    f_test_single:
        debug: true
        main_coil_overwrite:
            pulse_ms: 11
        main_coil: c_flipper_main
        activation_switch: s_flipper
    f_test_hold:
        debug: true
        main_coil: c_flipper_main
        hold_coil: c_flipper_hold
        activation_switch: s_flipper
    f_test_hold_eos:
        debug: true
        main_coil: c_flipper_main
        hold_coil: c_flipper_hold
        activation_switch: s_flipper
        eos_switch: s_flipper_eos
        use_eos: true

servos:
    servo1:
        number: 0-11
        servo_min: 0.012
        servo_max: 0.108
        reset_position: 0
    servo2:
        number: 0-14

lights:
  test_rgb_led_1:
    start_channel: 2-1-0
    type: rgb
    subtype: led

  test_rgb_led_2:
```

(continues on next page)

(continued from previous page)

```
    previous: test_rgb_led_1
    type: rgb
    subtype: led

  test_rgb_led_3:
    previous: test_rgb_led_2
    type: rgbw
    subtype: led

  test_rgb_led_4:
    previous: test_rgb_led_3
    type: rgb
    subtype: led

  test_rgbw_led_1:
    start_channel: 3-1-0
    type: rgbw
    subtype: led

  test_rgbw_led_2:
    previous: test_rgbw_led_1
    type: rgbw
    subtype: led

  test_rgbw_led_3:
    previous: test_rgbw_led_2
    type: rgb
    subtype: led

  test_rgbw_led_4:
    previous: test_rgbw_led_3
    type: rgbw
    subtype: led

  test_simple_led:
    number: 2-17
    subtype: simple

  test_other_simple_led:
    number: 3-1
    subtype: simple
```

platform (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 296: your_machine_folder/platform/config/test_virtual.yaml

```
#config_version=5

switches:
    s_test:
        number: 1
        platform_settings:
            debounce_open: 20ms
    switch1_p_roc:  # this should not cause duplicate switch exceptions
        number: 1
        platform: p_roc
    switch1_p_fast:
        number: 1
        platform: fast

coils:
    c_test:
        default_pulse_power: 0.128
        number: 1
    c_test_no_allow_enable:
        number: 2
    c_test_allow_enable:
        number: 3
        max_hold_power: 1.0
    c_test_hold_power:
        number: 4
        default_hold_power: 0.1
    coil1_p_roc:    # this should not cause duplicate coil exceptions
        number: 1
        platform: p_roc
    coil1_fast:
        number: 1
        platform: fast

# this should not cause duplicate light exceptions
lights:
    light1_p_roc:
        number: 1
        platform: p_roc
    light1_fast:
        number: 1
        platform: fast
    light1_virtual:
        number: 1
```

Listing 297: your_machine_folder/platform/config/test_platform.yaml

```
#config_version=5

hardware:
  platform: smart_virtual, virtual

switches:
```

(continues on next page)

(continued from previous page)

```
  switch1:
    number: 1
  switch2:
    number: 2
    platform: virtual
```

player_vars (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 298: your_machine_folder/player_vars/config/player_vars.yaml

```
#config_version=5

player_vars:
  some_var:
    initial_value: 4
  some_float:
    initial_value: 4
    value_type: float
  some_string:
    initial_value: 4
    value_type: str
  some_other_string:
    initial_value: hello
    value_type: str  # required for non-ints

machine_vars:
  test1:
    initial_value: 4
    value_type: int
  test2:
    initial_value: '5'
    value_type: str

# below is the min config we need to be able to start a game

game:
    balls_per_game: 3

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:

switches:
    s_start:
```

(continues on next page)

(continued from previous page)

```
        number:
        tags: start
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch_launcher:
        number:

playfields:
    playfield:
        default_source_device: bd_launcher
        tags: default

ball_devices:
    bd_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2
        debug: true
        confirm_eject_type: target
        eject_targets: bd_launcher
        tags: trough, drain, home
    bd_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
        debug: true
        confirm_eject_type: target
        eject_timeouts: 2s
```

playfield (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 299: your_machine_folder/playfield/config/test_playfield.yaml

```
#config_version=5

switches:
  s_playfield:
    number:
    tags: playfield_active
```

playfield_transfer (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 300: your_machine_folder/playfield_transfer/config/config.yaml

```
#config_version=5

switches:
    s_transfer:
        number:

playfield_transfers:
    transfer1:
        ball_switch: s_transfer
        captures_from: playfield1
        eject_target: playfield2

    transfer2:
        transfer_events: transfer_ball
        captures_from: playfield1
        eject_target: playfield2

playfields:
    playfield1:
        label: Playfield 1
        default_source_device: None
    playfield2:
        label: Playfield 2
        default_source_device: None
```

plugin_config_player (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 301: your_machine_folder/plugin_config_player/config/plugin_config_player.yaml

```
#config_version=5

modes:
  - mode1

test_player:
  event1: some_string
  event2:
    some: dict
    with: arbitrary
    values: '.'
```

(continues on next page)

(continued from previous page)

```
  event5{foo==0}: some_string

test2_player:
  event2: slide1
  event3: slide2
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 302: your_machine_folder/plugin_config_player/modes/mode1/shows/show2.yaml

```
#show_version=5
- time: 0
  tests:
    some:
      key1: thing

- time: 1
  tests:
    some:
      key: value
      key1: value
  test2s:
    some:
      key1: value
```

Listing 303: your_machine_folder/plugin_config_player/modes/mode1/shows/show3.yaml

```
#show_version=5
- time: 0
  test3s:
    test3_something:
      test3_key: test3_value
```

Listing 304: your_machine_folder/plugin_config_player/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  priority: 400
  game_mode: False

test_player:
  event1: some_string
  event4: something

test2_player:
  event2: slide1
```

(continues on next page)

(continued from previous page)

```
  event3: slide2

show_player:
  start_show2: show2
  start_show3: show3
```

Show file examples

Here are some example show files that go along with the above config(s).

Listing 305: your_machine_folder/plugin_config_player/shows/show1.yaml

```
#show_version=5
- time: 0
  tests:
    some5:
      key5: thing

- time: 1
  tests:
    slide1:
      key6: value
      key6.1: value
    transition:
      key7: value2
      key7.1: value3
  test2s:
    some7:
      key7: value
```

pololu_maestro (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 306: your_machine_folder/pololu_maestro/config/pololu_maestro.yaml

```
#config_version=5

hardware:
  platform: virtual
  driverboards: virtual
  servo_controllers: pololu_maestro

pololu_maestro:
  port: COM5
  servo_min: 3000
  servo_max: 9000
```

(continues on next page)

(continued from previous page)

```
servos:
  servo1:
    servo_min: 0.0
    servo_max: 1.0
    reset_position: 0.5
    speed_limit: 0.5
    acceleration_limit: 0.5
    reset_events: reset_servo1
    number: 1
  servo2:
    servo_min: 0.2
    servo_max: 0.8
    reset_position: 1.0
    reset_events: reset_servo2
    number: 2
  servo1_controller_13:
    number: 13-1
```

pololu_tic (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 307: your_machine_folder/pololu_tic/config/config.yaml

```
#config_version=5

hardware:
  platform: virtual
  switches: pololu_tic
  stepper_controllers: pololu_tic

switches:
  s_home:
    number: 1337-SDA
  s_test:
    number: 1337-RX

steppers:
  stepper1:
    number: 1337
    homing_mode: switch
    homing_switch: s_home
    named_positions:
      10: test_00
      20: test_01
      50: test_10
```

randomizer (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 308: your_machine_folder/randomizer/config/randomizer.yaml

```
#config_version=5
```

rpi (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 309: your_machine_folder/rpi/config/config.yaml

```
#config_version=5

hardware:
    platform: rpi

raspberry_pi:
    ip: localhost
    port: 8888

switches:
    s_test:
        number: 1
    s_test2:
        number: 7

coils:
    c_test:
        number: 23
        default_pulse_ms: 23
    c_test_allow_enable:
        number: 30
        default_hold_power: 1.0
    c_pwm:
        number: 2
        default_hold_power: 0.2

servos:
    servo1:
        number: 10
```

rpi_dmd (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 310: your_machine_folder/rpi_dmd/config/config.yaml

```
#config_version=5

hardware:
    platform: rpi_dmd

rpi_dmd:
  gpio_slowdown: 2
  pwm_lsb_nanoseconds: 300

displays:
  dmd:
    width: 32
    height: 32

rgb_dmds:
  rpi_dmd:
    label: RPi RGB DMD
```

score_queue (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 311: your_machine_folder/score_queue/config/config.yaml

```
#config_version=5

modes:
  - mode1

coils:
    c_chime_1000:
        number:
    c_chime_100:
        number:
    c_chime_10:
        number:

score_queues:
    score:
```

(continues on next page)

(continued from previous page)

```
        chimes: c_chime_1000, c_chime_100, c_chime_10,  None
        debug: True
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 312: your_machine_folder/score_queue/modes/mode1/config/mode1.yaml

```
#config_version=5
mode:
    start_events: start_mode1
    stop_events: stop_mode1
    priority: 200

score_queue_player:
    score_2k:
        score: 2000
    score_200:
        score: 200
```

score_reels (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 313: your_machine_folder/score_reels/config/config.yaml

```
#config_version=5

switches:
    s_start:
        number: 1
        tags: start
    score_1p_10k_0:
        number: 2
    score_1p_10k_9:
        number: 3
    score_1p_1k_0:
        number: 4
    score_1p_1k_9:
        number: 5
    score_1p_100_0:
        number: 6
    score_1p_100_9:
        number: 7
    score_1p_10_0:
        number: 8
```

(continues on next page)

(continued from previous page)

```
    score_1p_10_9:
        number: 9
    score_2p_10_0:
        number: 10
    score_2p_10_9:
        number: 11

virtual_platform_start_active_switches:
    - score_1p_10k_0
    - score_1p_1k_0
    - score_1p_100_0
    - score_1p_10_0
    - score_2p_10_0

coils:
    player1_10k:
        number:
    player1_1k:
        number:
    player1_100:
        number:
    player1_10:
        number:
    player2_10:
        number:
    chime1:
        number:
    chime2:
        number:
    chime3:
        number:

score_reels:
    score_1p_10k:
        coil_inc: player1_10k
        switch_0: score_1p_10k_0
        switch_9: score_1p_10k_9
        limit_hi: 9
        limit_lo: 0
        debug: True
    score_1p_1k:
        coil_inc: player1_1k
        switch_0: score_1p_1k_0
        switch_9: score_1p_1k_9
        limit_hi: 9
        limit_lo: 0
        debug: True
    score_1p_100:
        coil_inc: player1_100
        switch_0: score_1p_100_0
        switch_9: score_1p_100_9
        limit_hi: 9
        limit_lo: 0
        debug: True
```

(continues on next page)

(continued from previous page)

```
    score_1p_10:
        coil_inc: player1_10
        switch_0: score_1p_10_0
        switch_9: score_1p_10_9
        limit_hi: 9
        limit_lo: 0
        debug: True
    score_2p_10:
        coil_inc: player2_10
        switch_0: score_2p_10_0
        switch_9: score_2p_10_9
        limit_hi: 9
        limit_lo: 0
        debug: True

score_reel_groups:
    player1:
        reels: score_1p_10k, score_1p_1k, score_1p_100, score_1p_10, None
        tags: player1
        chimes: None, chime1, chime2, chime3, None
        lights_tag: player1
        debug: True
    player2:
        reels: score_2p_10, None
        tags: player2
        chimes: chime3, None
        lights_tag: player2
        debug: True

lights:
    light_p1:
        number:
        tags: player1
    light_p2:
        number:
        tags: player2
```

scriptlet (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 314: your_machine_folder/scriptlet/config/config.yaml

```
#config_version=5

scriptlets: test_scriptlet.TestScriptlet
```

segment_display (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 315: your_machine_folder/segment_display/config/game.yaml

```
#config_version=5

segment_displays:
  display1:
    number: 1
  display2:
    number: 2
  display3:
    number: 3
  display4:
    number: 4
  display5:
    number: 5

segment_display_player:
  # empty all displays on game start and setup display5
  game_start:
    display1:
      text: ""
    display2:
      text: ""
    display3:
      text: ""
    display4:
      text: ""
    display5:
      text: "{current_player.ball:d}"

  # clear only display5 after game
  game_ended{machine.player1_score > 0}:
    display1:
      text: "{machine.player1_score:d}"
  game_ended{machine.player2_score > 0}:
    display2:
      text: "{machine.player2_score:d}"
  game_ended{machine.player3_score > 0}:
    display3:
      text: "{machine.player3_score:d}"
  game_ended{machine.player4_score > 0}:
    display4:
      text: "{machine.player4_score:d}"
  game_ended:
```

(continues on next page)

(continued from previous page)

```
    display5:
      text: ""

  # flash display on player turn
  player_turn_started.1{number==1}:
    display1:
      action: flash
  player_turn_ended.2{number==1}:
    display1:
      action: no_flash
  player_turn_started.3{number==2}:
    display2:
      action: flash
  player_turn_ended.4{number==2}:
    display2:
      action: no_flash
  player_turn_started.5{number==3}:
    display3:
      action: flash
  player_turn_ended.6{number==3}:
    display3:
      action: no_flash
  player_turn_started.7{number==4}:
    display4:
      action: flash
  player_turn_ended.8{number==4}:
    display4:
      action: no_flash

  # show score when adding players
  player_added.1{num==1}:
    display1:
      text: "{players[0].score:d}"
  player_added.2{num==2}:
    display2:
      text: "{players[1].score:d}"
  player_added.3{num==3}:
    display3:
      text: "{players[2].score:d}"
  player_added.4{num==4}:
    display4:
      text: "{players[3].score:d}"
```

Listing 316: your_machine_folder/segment_display/config/config.yaml

```
#config_version=5

modes:
 - mode1

segment_displays:
  display1:
    number: 1
    size: 10
```

(continues on next page)

(continued from previous page)

```
  display2:
    number: 2
  display3:
    number: 3
  display4:
    number: 1
    size: 10
    integrated_commas: true
  display5:
    number: 1
    size: 10
    integrated_dots: true

segment_display_player:
  test_event1:
    display1: "HELLO1"
    display2:
      text: "HELLO2"
  test_event2:
    display1:
      action: remove
  test_event3:
    display2:
      action: remove

  test_flashing:
    display1:
      action: flash
  test_no_flashing:
    display1:
      action: no_flash

  test_score:
    display1:
      text: "1: {players[0].score:d}"
    display2:
      text: "2: {machine.test:d}"

  test_score_two_player:
    display1:
      text: "{players[0].score:d}"
    display2:
      text: "{players[1].score:d}"

  test_flash:
    display1:
      priority: 10
      key: flash
      text: "TEST"
      expire: 2s

  test_update_events:
    display3:
      text: "UPDATE"
```

(continues on next page)

(continued from previous page)

```
    color: FF0000

  test_transition:
    display1:
      priority: 15
      key: transition
      text: "  SCROLL  "
      color: red
      transition:
        type: push
        direction: right
      transition_out:
        type: push
        direction: left
      expire: 2s

  test_transition_2:
    display1:
      priority: 15
      key: transition
      text: "0123456789"
      transition:
        type: split
        mode: wipe
        direction: out

  test_transition_3:
    display1:
      priority: 15
      key: transition
      text: "ABCDEFGHIJ"
      transition:
        type: uncover
        direction: right

  test_set_color_to_white:
    display3:
      action: set_color
      color: white

  test_set_color_to_red:
    display3:
      action: set_color
      color: red
```

Listing 317: your_machine_folder/segment_display/config/config_transition.yaml

```
#config_version=5
segment_displays:
  display1:
    debug: true
    number: 1
    size: 10
```

(continues on next page)

(continued from previous page)

```
segment_display_player:
  test_event1:
    display1:
      priority: 1
      text: EVENT1
      color: red
      expire: 2s
      transition:
        type: push
        direction: left
      transition_out:
        type: cover
        direction: left
  test_event2:
    display1:
      priority: 10
      text: EVENT2
      color: blue
      expire: 5s
      transition:
        type: push
        direction: right

  test_event3:
    display1:
      key: test3
      priority: 1
      text: EVENT3
      color: red
      expire: 2s
      transition:
        type: push
        direction: left
      transition_out:
        type: cover
        direction: left
  test_event4:
    display1:
      key: test4
      priority: 10
      text: EVENT4
      color: blue
      expire: 5s
      transition:
        type: push
        direction: right
```

Listing 318: your_machine_folder/segment_display/config/config_colors.yaml

```
#config_version=5
segment_displays:
  display1:
    debug: true
    number: 1
```

(continues on next page)

(continued from previous page)

```
    size: 10

segment_display_player:
  test_event1:
    display1:
      text: EVENT1
      color: [red, blue, yellow, green, white, purple]
  test_event2:
    display1:
      text: EVENT2
      color: [red, blue, yellow, green, white, purple]
      transition:
        type: uncover
        direction: left
```

Listing 319: your_machine_folder/segment_display/config/config_flashing.yaml

```
#config_version=5
segment_displays:
  display1:
    debug: true
    number: 1
    size: 10

segment_display_player:
  test_event1:
    display1:
      flashing: all
      text: EVENT1
      transition:
        type: push
        direction: left
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 320: your_machine_folder/segment_display/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  priority: 100

segment_display_player:
  mode_mode1_started:
    display1:
      text: "MODE1"
    display2:
      text: "MODE1"
      expire: 10s
```

segment_display_widget (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 321: your_machine_folder/segment_display_widget/config/test_segment_display_widget.yaml

```
#config_version=5

window:
  width: 800
  height: 600

widgets:
  segment_display_widget_top:
    type: segment_display_emulator
    name: display1
    character_count: 7
    character_slant_angle: 6
    character_spacing: 20
    segment_width: 0.11
    segment_interval: 0.04
    segment_off_color: 4b4c4a30
    segment_on_color: fe961bff
    side_bevel_enabled: true
    dot_enabled: true
    comma_enabled: true
    character_map:
      16: 54
      17: 8264
      18: 456
      19: 235
      20: 1240
    text: "*HELLO*"
    width: 600
    height: 150
    y: 450

  segment_display_widget_middle:
    type: segment_display_emulator
    name: display3
    display_type: 7seg
    character_count: 8
    character_slant_angle: 6
    character_spacing: 5
    segment_width: 0.11
    segment_interval: 0.04
    segment_off_color: 4b4c4a30
    segment_on_color: f01020ff,f01020ff,f01020ff,f01020ff,f01020ff,f01020ff,f01020ff,008000ff
    side_bevel_enabled: true
    flash_mode: "mask"
    flash_mask: "_______F"
```

(continues on next page)

(continued from previous page)

```
    flash_frequency: 4
    dot_enabled: false
    comma_enabled: true
    text: "BALL 2"
    width: 500
    height: 120
    y: 260

  segment_display_widget_bottom:
    type: segment_display_emulator
    name: display2
    character_count: 16
    character_slant_angle: 6
    character_spacing: 5
    segment_width: 0.11
    segment_interval: 0.04
    segment_off_color: 4b4c4a30
    segment_on_color: fe961bff
    side_bevel_enabled: true
    dot_enabled: true
    comma_enabled: true
    character_map:
      16: 54
      17: 8264
      18: 456
      19: 235
      20: 1240
    text: ">TESTS<"
    width: 700
    height: 100
    y: 100

widget_player:
  show_top_display: segment_display_widget_top
  show_middle_display: segment_display_widget_middle
  show_bottom_display: segment_display_widget_bottom

  update_display:
    segment_display_widget_top:
      action: update
      widget_settings:
        text: "GOODBYE"
    segment_display_widget_bottom:
      action: update
      widget_settings:
        text: ""
```

sequence_shot (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 322: your_machine_folder/sequence_shot/config/config.yaml

```yaml
#config_version=5

modes:
  - mode1

switches:
  seq2_1:
    number:
  seq2_2:
    number:
  seq2_3:
    number:
  seq2_cancel:
    number:
  seq2_delay:
    number:
  seq4_1:
    number:
  seq4_delay:
    number:

sequence_shots:
    sequence1:
        event_sequence:
            - event1
            - event2
            - event3
        cancel_events: cancel
        delay_event_list:
            delay1: 1s
        sequence_timeout: 3s
    sequence2:
        switch_sequence:
            - seq2_1
            - seq2_2
            - seq2_3
        cancel_switches: seq2_cancel
        delay_switch_list:
            seq2_delay: 1s
        sequence_timeout: 3s
    sequence3:
        event_sequence:
            - event3_1
    sequence_with_dupes:
        event_sequence:
            - event_1
            - event_2
            - event_1
```

(continues on next page)

(continued from previous page)

```
            - event_3
            - event_1
    sequence4:
        debug: True
        switch_sequence:
            - seq4_1
        delay_switch_list:
            seq4_delay: 1s
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 323: your_machine_folder/sequence_shot/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  priority: 100
  game_mode: False

sequence_shots:
    sequence_mode_event:
        event_sequence:
            - event1
            - event2
        cancel_events: cancel
        delay_event_list:
            delay1: 1s
        sequence_timeout: 3s
    sequence_mode_switch:
        switch_sequence:
            - seq2_1
            - seq2_2
        cancel_switches: seq2_cancel
        delay_switch_list:
            seq2_delay: 1s
        sequence_timeout: 3s
```

service_mode (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 324: your_machine_folder/service_mode/config/config.yaml

```
#config_version=5
```

(continues on next page)

(continued from previous page)

```
game:
    balls_per_game: 1

modes:
    - attract
    - game
    - service
    - credits

credits:
  free_play: no
  service_credits_switch: s_service_esc

coils:
    c_test:
        number: 1
        label: First coil
    c_test2:
        number: 2
        label: Second coil
    c_test3:
        number: 1000
        label: Sixth coil
    c_test4:
        number: 100
        label: Fifth coil
    c_test5:
        number: 3
        label: Third coil
    c_test6:
        default_hold_power: 1.0
        number: 10
        label: Fourth coil


switches:
    s_door_open:
        number: 1
        tags: service_door_open, power_off
    s_service_enter:
        number: 17
        tags: service_enter
    s_service_esc:
        number: 18
        tags: service_esc
    s_service_up:
        number: 19
        tags: service_up
    s_service_down:
        number: 20
        tags: service_down

lights:
    l_light1:
```

(continues on next page)

(continued from previous page)

```
      number: 1
    l_light5:
      number: 5
      label: Light Five

sound_system:
  tracks:
    sfx: []
  enabled: true

keyboard:
    right:
        switch: s_service_enter
    left:
        switch: s_service_esc
    up:
        switch: s_service_up
    down:
        switch: s_service_down
    enter:
        switch: s_door_open
        toggle: true
```

servo (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 325: your_machine_folder/servo/config/config.yaml

```
#config_version=5

servos:
    limited_servo:
        number: 1
        servo_min: 0.2
        servo_max: 0.8
    test_servo:
        number: 2
        reset_position: 0.5
        reset_events: test_reset
        positions:
            0.0: test_00
            0.1: test_01
            1.0: test_10
    test_servo_with_timeout:
        number: 3
        stop_timeout_after_last_move: 2s
        positions:
```

(continues on next page)

(continued from previous page)

```
        0.0: test_20
        0.5: test_25
        1.0: test_30
```

settings (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 326: your_machine_folder/settings/config/config.yaml

```
#config_version=5

settings:
    custom_setting_int:
        label: "Int Setting"
        key_type: int
        default: 0
        sort: 1
        values:
            0: "Zero"
            1: "One"
            2: "Two"
    custom_setting_str:
        label: "String Setting"
        key_type: str
        default: "one"
        sort: 2
        values:
            zero: "Zero"
            one: "One"
            two: "Two"
```

shapes (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 327: your_machine_folder/shapes/config/test_shapes.yaml

```
#config_version=5

displays:
  default:
```

(continues on next page)

(continued from previous page)

```
    width: 400
    height: 300

slides:
  slide1:
    - type: points
      points: 50, 50, 75, 50, 100, 30, 200, 50, 68, 250
      pointsize: 3
    - type: line
      points: 0, 0, 100, 100, 100, 200
      color: 00ff00
      thickness: 10
      close: true
    - type: bezier
      points: 400, 300, 100, 100, 400, 0
      color: pink
      thickness: 5
    - type: triangle
      points: 400, 300, 200, 300, 400, 200
      color: red
    - type: quad
      points: 50, 50, 55, 70, 100, 75, 110, 45
      color: lightblue
    - type: ellipse
      width: 100
      height: 100
      color: purple
      angle_start: 0
      angle_end: 45
    - type: rectangle
      x: 250
      y: 125
      width: 200
      height: 100
      color: orange
      corner_radius: 30
    - type: rectangle
      x: 350
      y: 50
      width: 50
      height: 100
      color: blue

slide_player:
  slide1: slide1
```

shots (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 328: your_machine_folder/shots/config/test_shots.yaml

```
#config_version=5

modes:
  - base2
  - base3
  - mode1
  - mode2

switches:
  switch_1:
    number:
  switch_2:
    number:
  switch_3:
    number:
  switch_4:
    number:
  switch_5:
    number:
  switch_6:
    number:
  switch_7:
    number:
  switch_8:
    number:
  switch_9:
    number:
  switch_10:
    number:
  s_delay:
    number:
  switch_11:
    number:
  switch_12:
    number:
  switch_13:
    number:
  switch_14:
    number:
  switch_15:
    number:
  switch_16:
    number:
  switch_17:
    number:
  switch_18:
    number:
  switch_19:
    number:
  switch_20:
```

(continues on next page)

(continued from previous page)

```
    number:
  switch_21:
    number:
  switch_22:
    number:
  switch_26:
    number:
  switch_27:
    number:
  switch_28:
    number:

lights:
  light_1:
    number:
    tags: tag1
    subtype: matrix
  light_2:
    number:
    tags: tag2
    subtype: matrix
  light_3:
    number:
    subtype: matrix
  light_4:
    number:
    subtype: matrix
  light_5:
    number:
    subtype: matrix
  light_6:
    number:
    subtype: matrix
  led_1:
    number:
  led_2:
    number:
  led_3:
    number:
  led_4:
    number:
  led_5:
    number:
  led_6:
    number:
  led_11:
    number:
  led_12:
    number:
  led_13:
    number:
  led_14:
    number:
  led_15:
```

(continues on next page)

(continued from previous page)

```
    number:
  led_16:
    number:
  led_17:
    number:
  led_18:
    number:
  led_19:
    number:
  led_20:
    number:
  led_21:
    number:
  led_23:
    number:
  led_24:
    number:
  led_25:
    number:
  led_26:
    number:
  led_27:
    number:
  led_28:
    number:
  led_29:
    number:

shows:
    rainbow:
      - lights:
          (leds): red
      - lights:
          (leds): orange
      - lights:
          (leds): yellow
      - lights:
          (leds): green
      - lights:
          (leds): blue
      - lights:
          (leds): purple
    rainbow_stay_on:
      - lights:
          (leds): red
      - lights:
          (leds): orange
      - lights:
          (leds): yellow
      - lights:
          (leds): green
      - lights:
          (leds): blue
      - lights:
```

(continues on next page)

(continued from previous page)

```
        (leds): purple
      duration: -1
  rainbow2:
    - lights:
        (leds): aliceblue
    - lights:
        (leds): antiquewhite
    - lights:
        (leds): aquamarine
    - lights:
        (leds): azure
  rainbow3:
    - lights:
        (leds): beige
    - lights:
        (leds): blueviolet
    - lights:
        (leds): brown
    - lights:
        (leds): burlywood
```

Listing 329: your_machine_folder/shots/config/test_shot_groups.yaml

```
#config_version=5

modes:
  - base

switches:
  switch_1:
    number:
  switch_2:
    number:
  switch_3:
    number:
  switch_4:
    number:
  s_rotate_l:
    number:
  s_rotate_r:
    number:
  switch_10:
    number:
  switch_11:
    number:
  switch_30:
    number:
  switch_31:
    number:
  switch_32:
    number:
  switch_33:
    number:
  switch_34:
```

(continues on next page)

(continued from previous page)

```
    number:
  switch_35:
    number:
  switch_36:
    number:
  switch_37:
    number:
  switch_38:
    number:
  switch_39:
    number:
  switch_40:
    number:
  switch_41:
    number:
  switch_42:
    number:
  switch_43:
    number:
  switch_44:
    number:
  switch_45:
    number:
  switch_46:
    number:
  s_GAS_G:
    number:
  s_GAS_A:
    number:
  s_GAS_S:
    number:
  s_special_left:
    number:
  s_special_right:
    number:

lights:
  led_10:
    number:
  led_11:
    number:
  led_30:
    number:
  led_31:
    number:
  led_32:
    number:
  led_33:
    number:
  led_34:
    number:
  led_35:
    number:
  led_36:
```

(continues on next page)

(continued from previous page)

```
    number:
  led_37:
    number:
  led_38:
    number:
  led_39:
    number:
  led_40:
    number:
  led_41:
    number:
  led_42:
    number:
  l_GAS_G:
    number:
  l_GAS_A:
    number:
  l_GAS_S:
    number:
  l_special_right:
    number:
    subtype: matrix
  l_special_left:
    number:
    subtype: matrix

shows:
    rainbow:
      - lights:
          (leds): off
      - lights:
          (leds): red
      - lights:
          (leds): orange
      - lights:
          (leds): yellow
      - lights:
          (leds): green
    leds_off:
      - lights:
          (led): off
    leds_on:
      - lights:
          (led): white
```

Listing 330: your_machine_folder/shots/config/test_shot_group_rotate_with_exclude.yaml

```
#config_version=5

modes:
  - rotate_with_exclude

switches:
  s_rotate_l:
```

(continues on next page)

(continued from previous page)

```
    number:
  s_rotate_r:
    number:
  switch_1:
    number:
  switch_2:
    number:
  switch_3:
    number:
  switch_4:
    number:
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 331: your_machine_folder/shots/modes/mode2/config/mode2.yaml

```
#config_version=5

mode:
  priority: 200

shots:
  mode2_shot_rainbow:
    switch: switch_27
    show_tokens:
      leds: led_27
    profile: rainbow
  mode2_shot_rainbow_start_step:
    switch: switch_28
    show_tokens:
      leds: led_28
    profile: rainbow_start_step
  mode2_shot_2:
    switch: switch_2
    show_tokens:
      leds: light_2
    profile: rainbow_start_step
  mode2_shot_show_tokens:
    hit_events: mode2_shot_show_tokens_advance
    enable_events: mode2_shot_show_tokens_enable
    reset_events: mode2_shot_show_tokens_reset
    disable_events: mode2_shot_show_tokens_disable
    show_tokens:
      leds: (machine.leds)
    profile: show_tokens_profile
  mode2_shot_changing_profile:
    profile: changing_profile_one
```

(continues on next page)

(continued from previous page)

```
shows:
  show_with_tokens:
    - lights:
        (leds): (color)

shot_profiles:
  show_tokens_profile:
    states:
      - name: one
        show: show_with_tokens
        show_tokens:
          color: (machine.color1)
      - name: two
        show: show_with_tokens
        show_tokens:
          color: (machine.color2)
      - name: three
        show: show_with_tokens
        show_tokens:
          color: (machine.color3)
  mode2_shot_21:
    states:
      - name: mode2_one
      - name: mode2_two
      - name: mode2_three
  mode2_shot_22:
    states:
      - name: mode2_one
      - name: mode2_two
      - name: mode2_three
  rainbow_start_step:
    states:
      - name: red
        show: rainbow
        start_step: 1
        manual_advance: True
      - name: orange
        show: rainbow
        start_step: 2
        manual_advance: True
      - name: yellow
        show: rainbow
        start_step: 3
        manual_advance: True
      - name: green
        show: rainbow
        start_step: 4
        manual_advance: True
      - name: blue
        show: rainbow
        start_step: 5
        manual_advance: True
      - name: purple
        show: rainbow
```

(continues on next page)

(continued from previous page)

```
      start_step: 6
      manual_advance: True
  changing_profile_one:
    states:
      - name: first
        show: show_with_tokens
        show_tokens:
          leds: led_20
          color: yellow
  changing_profile_two:
    states:
      - name: first
        show: show_with_tokens
        show_tokens:
          leds: led_20
          color: purple
  mode2_shot_26:
    states:
    - name: mode2_one
      show: rainbow3
    - name: mode2_two
      show: rainbow3
    - name: mode2_three
      show: rainbow3
```

Listing 332: your_machine_folder/shots/modes/base/config/base.yaml

```
#config_version=5
mode:
    start_events: player_turn_started
    stop_events: player_turn_stopped
    priority: 100

shots:
  shot_1:
    switch: switch_1
  shot_2:
    switch: switch_2
  shot_3:
    switch: switch_3
  shot_4:
    switch: switch_4
  shot_10:
    switch: switch_10
    show_tokens:
      leds: led_10
  shot_11:
    switch: switch_11
    show_tokens:
      leds: led_11
  shot_30:
    switch: switch_30
    show_tokens:
      leds: led_30
```

(continues on next page)

(continued from previous page)

```
  profile: rainbow
shot_31:
  switch: switch_31
  show_tokens:
    leds: led_31
  profile: rainbow
shot_32:
  switch: switch_32
  show_tokens:
    leds: led_32
  start_enabled: False
  profile: rainbow
shot_33:
  switch: switch_33
  show_tokens:
    leds: led_33
  start_enabled: False
  profile: rainbow
shot_34:
  switch: switch_34
  show_tokens:
    leds: led_34
  enable_events: None
shot_35:
  switch: switch_35
  show_tokens:
    leds: led_35
  enable_events: None
shot_36:
  switch: switch_36
  show_tokens:
    leds: led_36
  enable_events: None
shot_37:
  switch: switch_37
  show_tokens:
    leds: led_37
  enable_events: None
shot_38:
  switch: switch_38
  show_tokens:
    leds: led_38
  enable_events: None
shot_39:
  switch: switch_39
  show_tokens:
    leds: led_39
  enable_events: None
shot_40:
  switch: switch_40
  show_tokens:
    leds: led_40
  profile: shot_profile_40
shot_41:
```

(continues on next page)

(continued from previous page)

```
    switch: switch_41
    show_tokens:
      leds: led_41
    profile: shot_profile_40
  shot_42:
    switch: switch_42
    show_tokens:
      leds: led_42
    profile: shot_profile_40
  shot_43:
    switch: switch_43
  shot_44:
    switch: switch_44
  shot_45:
    switch: switch_45
    profile: rainbow
  shot_46:
    switch: switch_46
    profile: rainbow
  lane_special_left:
      switch: s_special_left
      show_tokens:
          light: l_special_left
      profile: prof_toggle
  lane_special_right:
      switch: s_special_right
      show_tokens:
          light: l_special_right
      profile: prof_toggle

shot_profiles:
    rainbow:
      show: rainbow
      states:
        - name: unlit
        - name: red
        - name: orange
        - name: yellow
        - name: green
    rainbow_no_hold:
      show: rainbow
      states:
        - name: unlit
        - name: red
        - name: orange
        - name: yellow
        - name: green
    shot_profile_40:
      show: rainbow
      rotation_pattern: r, r, l, l
      states:
        - name: unlit
        - name: red
        - name: orange
```

(continues on next page)

(continued from previous page)

```
      - name: yellow
      - name: green
  prof_toggle:
    states:
      - name: unlit_toggle
        show: off
      - name: lit_toggle
        show: on
    loop: true

shot_groups:
  test_group:
    shots: shot_1, shot_2, shot_3, shot_4
    rotate_left_events: s_rotate_l_active
    rotate_right_events: s_rotate_r_active
    debug: True
  test_group_2:
    shots: shot_10, shot_11
    rotate_left_events: rotate_11_left
  shot_group_30:
    shots: shot_30, shot_31
  shot_group_32:
    shots: shot_32, shot_33
    enable_events: group32_enable
    disable_events: group32_disable
    reset_events: group32_reset
    restart_events: group32_restart
    rotate_left_events: group32_rotate_left
    rotate_right_events: group32_rotate_right
    enable_rotation_events: group32_enable_rotation
    disable_rotation_events: group32_disable_rotation
  shot_group_34:
    shots: shot_34, shot_35, shot_36
  shot_group_37:
    shots: shot_37, shot_38, shot_39
  shot_group_40:
    shots: shot_40, shot_41, shot_42
  shot_group_43:
    shots: shot_43, shot_44
  shot_group_45:
    shots: shot_45, shot_46
  special:
    shots: lane_special_left
```

Listing 333: your_machine_folder/shots/modes/base3/config/base3.yaml

```
#config_version=5
mode:
    start_events: player_turn_started
    stop_events: player_turn_stopped
    priority: 50

shots:
  shot_state_1:
```

(continues on next page)

(continued from previous page)

```
    switch: switch_1
    show_tokens:
      light: light_1
    profile: state_toggle
    advance_events: advance_event1
    reset_events: reset_event1
    control_events:
      - events: state_event1, state_event10
        state: 1
  shot_state_2:
    switch: switch_2
    show_tokens:
      light: light_2
    profile: state_loop_3
    advance_events: advance_event2
    reset_events: reset_event2
    enable_events: enable_event2
    disable_events: disable_event2
    control_events:
      - events: state_event2
        state: 0
        force: false
      - events: state_event3
        state: 0
      - events: state_event4
        state: 2

shot_profiles:
    state_toggle:
        states:
        - name: unlit
        - name: lit
    state_loop_3:
      loop: True
      states:
        - name: one
        - name: two
        - name: three
```

Listing 334: your_machine_folder/shots/modes/rotate_with_exclude/config/rotate_with_exclude.yam

```
#config_version=5

mode:
    start_events: ball_started
    priority: 100

shot_profiles:
    profile_state_names_to_not_rotate:
        state_names_to_not_rotate: unlit
        states:
            - name: unlit
            - name: red
            - name: orange
```

(continues on next page)

(continued from previous page)

```
shots:
  shot_1:
    switch: switch_1
    profile: profile_state_names_to_not_rotate
  shot_2:
    switch: switch_2
    profile: profile_state_names_to_not_rotate
  shot_3:
    switch: switch_3
    profile: profile_state_names_to_not_rotate
  shot_4:
    switch: switch_4
    profile: profile_state_names_to_not_rotate

shot_groups:
  test_group:
    shots: shot_1, shot_2, shot_3, shot_4
    rotate_left_events: s_rotate_l_active
    rotate_right_events: s_rotate_r_active
    debug: True
```

Listing 335: your_machine_folder/shots/modes/base2/config/base2.yaml

```
#config_version=5
mode:
    start_events: player_turn_started
    stop_events: player_turn_stopped
    priority: 50

shots:
  shot_1:
    switch: switch_1
    show_tokens:
      light: light_1
  shot_2:
    switch: switch_2
    show_tokens:
      light: light_2
    profile: three_states_loop
  shot_3:
    switch: switch_3
    show_tokens:
      light: tag1
  shot_4:
    switch: switch_1
  led_1:
    switch: switch_1
    show_tokens:
      led: led_1
  shot_delay:
    switch: switch_1
    delay_switch:
      s_delay: 2s
```

(continues on next page)

(continued from previous page)

```
shot_delay_same_switch:
  switch: switch_15
  delay_switch:
    switch_15: 2s
default_show_light:
  switch: switch_5
  show_tokens:
    light: light_4
default_show_lights:
  switch: switch_6
  show_tokens:
    lights: light_5, light_6
default_show_led:
  switch: switch_7
  show_tokens:
    led: led_4
default_show_leds:
  switch: switch_8
  show_tokens:
    leds: led_5, led_6
show_in_profile_root:
  switch: switch_9
  show_tokens:
    leds: led_3
  profile: rainbow
shot_11:
  switch: switch_11
  show_tokens:
    leds: led_11
  profile: profile_11
shot_12:
  switch: switch_12
  show_tokens:
    leds: led_12
  profile: profile_12
shot_13:
  switch: switch_13
  show_tokens:
    leds: led_13
  profile: profile_13
shot_14:
  switch: switch_14
  show_tokens:
    leds: led_14
  profile: profile_14
shot_15:
  switches: switch_13, switch_14
shot_16:
  switch: switch_16
  enable_events: custom_enable_16
  disable_events: custom_disable_16
  reset_events: custom_reset_16
  hit_events: custom_hit_16
  advance_events: custom_advance_16
```

(continues on next page)

(continued from previous page)

```
    restart_events: custom_restart_16
  shot_17:
    switch: switch_17
    profile: profile_17
  shot_19:
    switch: switch_19
    profile: profile_19
    start_enabled: False
    show_tokens:
      leds: led_19
  shot_20:
    switch: switch_20
    profile: profile_20
    start_enabled: False
    show_tokens:
      leds: led_20
  shot_21:
    switch: switch_21
    profile: profile_21
  shot_22:
    switch: switch_22
    profile: profile_22
  shot_23:
    show_tokens:
      leds: led_23
    profile: profile_23
  shot_24:
    show_tokens:
      leds: led_24
    profile: profile_24
  shot_25:
    show_tokens:
      leds: led_25
    profile: profile_25
  shot_26:
    switch: switch_26
    show_tokens:
      leds: led_26
    profile: profile_26
  shot_27:
    switch: switch_1
  shot_28:
    hit_events: event1


shot_profiles:
    prof_toggle2:
        states:
            - name: unlit2
              show: off
            - name: lit2
              show: on
        loop: true
```

(continues on next page)

(continued from previous page)

```
three_states_loop:
  loop: True
  states:
    - name: one
    - name: two
    - name: three
rainbow:
  show: rainbow
  states:
    - name: red
    - name: orange
    - name: yellow
    - name: green
    - name: blue
    - name: purple
profile_11:
  loop: true
  states:
    - name: step1
      show: rainbow
    - name: step2
      show: rainbow2
profile_12:
  show: rainbow
  states:
    - name: one
    - name: two
    - name: three
      show: rainbow2
      loops: -1
    - name: four
    - name: five
profile_13:
  states:
    - name: one
      show: rainbow
    - name: two
    - name: three
      show: rainbow2
profile_14:
  states:
    - name: one
      show: rainbow_stay_on
      loops: 0
    - name: two
profile_17:
  advance_on_hit: false
  states:
    - name: one
    - name: two
    - name: three
    - name: four
    - name: five
profile_19:
```

(continues on next page)

(continued from previous page)

```
      show_when_disabled: true
      states:
        - name: one
          show: rainbow
        - name: two
          show: rainbow2
    profile_20:
      show_when_disabled: false
      states:
        - name: one
          show: rainbow
        - name: two
          show: rainbow2
    profile_21:
      states:
        - name: base_one
        - name: base_two
        - name: base_three
    profile_22:
      states:
        - name: base_one
        - name: base_two
        - name: base_three
    profile_23:
        states:
        - name: base_one
          show: rainbow
        - name: base_two
          show: rainbow
        - name: base_three
          show: rainbow
    profile_24:
        states:
        - name: base_one
          show: rainbow_stay_on
          loops: 0
        - name: base_two
          show: rainbow_stay_on
    profile_25:
        states:
        - name: base_one
          show: rainbow
          loops: 0
        - name: base_two
          show: rainbow
    profile_26:
        states:
        - name: base_one
          show: rainbow
        - name: base_two
          show: rainbow
        - name: base_three
          show: rainbow
```

Listing 336: your_machine_folder/shots/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  priority: 100

shots:
  mode1_shot_1:
    switch: switch_3
    start_enabled: True
    enable_events: custom_enable_1
    disable_events: custom_disable_1
  mode1_shot_17:
    switch: switch_17
    enable_events: custom_enable_17
    disable_events: custom_disable_17
    reset_events: custom_reset_17
    hit_events: custom_hit_17
  mode1_shot_2:
    switch: switch_2
    show_tokens:
      leds: light_2
    start_enabled: True
    profile: mode1_shot_2
  mode1_shot_3:
    switch: switch_3
    profile: mode1_shot_3

shot_profiles:
  mode1_shot_2:
    show: rainbow2
    states:
    - name: mode1_one
    - name: mode1_two
    - name: mode1_three
  mode1_shot_3:
    show: rainbow2
    block: True
    states:
    - name: mode1_one
    - name: mode1_two
    - name: mode1_three
```

shows (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 337: your_machine_folder/shows/config/test_show_pools.yaml

```
#config_version=5

lights:
    led_01:
        number: 0
        tags: tag1, row0
    led_02:
        number: 1
        tags: tag1, row0
    led_03:
        number: 2
        tags: row1
    led_04:
        number: 3
        tags: row2
    light_01:
        number: 0
        label: Test 0
        tags: tag1
        subtype: matrix
        debug: True
    light_02:
        number: 1
        label: Test 1
        tags: tag1
        subtype: matrix
        debug: True
    light_03:
        number: 2
        label: Test 1
        fade_ms: 1s
        subtype: matrix
        debug: True
    gi_01:
        number: 0
        subtype: gi
    flasher_01:
        platform: drivers
        number: flasher_01

coils:
    coil_01:
        number: 1
        default_pulse_ms: 30
    flasher_01:
        number: 2
        label: Test flasher
        default_pulse_ms: 40
        max_hold_power: 1.0

shows:
```

(continues on next page)

(continued from previous page)

```
leds_name_token:
  - time: 0
    lights:
      (leds): red
leds_single_color:
  - time: 0
    lights:
      led_01: (color)
leds_color_token:
  - time: 0
    lights:
      led_01: (color1)
  - time: +1
    lights:
      led_02: (color2)
  - time: +1
leds_extended:
  - time: 0
    lights:
      (leds):
        color: red
        fade: 1s
lights_basic:
  - time: 0
    lights:
      (lights): ff
multiple_tokens:
  - time: 0
    lights:
      (leds): blue
      (lights): ff
show_assoc_tokens:
  - time: 0
    lights:
      (line1num): (line1color)
show_with_time_and_duration:
  - time: +1s
  - time: 5s
  - time: +1s
    duration: 1s
  - lights:
      led_02: red
  - time: 10s
    duration: 3s
leds_color_token_and_fade:
  - time: 0
    lights:
      led_01: (color1)
  - time: +1
    lights:
      led_02: (color2)-f900ms
  - time: +1
manual_advance:
  - duration: -1
```

(continues on next page)

(continued from previous page)

```
      lights:
        (leds): red
    - duration: -1
      lights:
        (leds): lime
    - duration: -1
      lights:
        (leds): blue
  event_show:
    - duration: 1
      events:
        - step1
    - duration: 1
      events:
        - step2
    - duration: 1
      events:
        - step3

show_pools:
  pool_random:
    shows:
      - leds_name_token
      - leds_single_color
      - leds_color_token
      - leds_extended
    type: random
  pool_sequence:
    shows:
      - multiple_tokens
      - show_assoc_tokens
      - leds_color_token_and_fade
    type: sequence
  pool_rfn:
    shows:
      - lights_basic
      - show_with_time_and_duration
      - manual_advance
      - event_show
    type: random_force_next
  pool_rfa:
    shows:
      - leds_name_token
      - leds_single_color
      - leds_color_token
      - leds_extended
      - multiple_tokens
      - show_assoc_tokens
      - leds_color_token_and_fade
      - lights_basic
      - show_with_time_and_duration
      - manual_advance
      - event_show
    type: random_force_all
```

(continues on next page)

(continued from previous page)

```
show_player:
  play_pool_random:
    pool_random:
      show_tokens:
        leds: led_01
        color: blue
        color1: green
        color2: yellow
  stop_pool_random:
    pool_random: stop
  play_pool_sequence:
    pool_sequence:
      show_tokens:
        leds: led_01
        lights: light_01
        line1num: led_01
        line1color: red
        color1: violet
        color2: orange
  play_pool_rfn:
    pool_rfn:
      show_tokens:
        lights: light_01
        leds: led_01
  play_pool_rfa:
    pool_rfa:
      show_tokens:
        leds: led_01
        color: blue
        color1: green
        color2: yellow
        lights: light_01
        line1num: led_01
        line1color: red
```

Listing 338: your_machine_folder/shows/config/test_show_player_queue.yaml

```
#config_version=5


shows:
  show1:
    - duration: 1
      events:
        - step1_1
    - duration: 1
      events:
        - step1_2
    - duration: 1
      events:
        - step1_3
  show2:
    - duration: 1
```

(continues on next page)

(continued from previous page)

```
      events:
        - step2_1
    - duration: 1
      events:
        - step2_2
    - duration: 1
      events:
        - step2_3
  show3:
    - duration: 1
      events:
        - step3_1
    - duration: 1
      events:
        - step3_2
    - duration: 1
      events:
        - step3_3

show_queues:
  queue1:
    label: Queue 1
  queue2:
    label: Queue 2

show_player:
  play_show1_on_queue1:
    show1:
      action: queue
      show_queue: queue1
      loops: 0
  play_show2_on_queue1:
    show2:
      action: queue
      show_queue: queue1
      loops: 0
  play_show3_on_queue1:
    show3:
      action: queue
      show_queue: queue1
      loops: 0
  play_show1_on_queue2:
    show1:
      action: queue
      show_queue: queue2
      loops: 0
  play_show2_on_queue2:
    show2:
      action: queue
      show_queue: queue2
      loops: 0
```

Listing 339: your_machine_folder/shows/config/test_shows.yaml

```
#config_version=5

modes:
  - mode1
  - mode2
  - mode3
  - mode4

lights:
    led_01:
        number: 0
        tags: tag1, row0
    led_02:
        number: 1
        tags: tag1, row0
    led_03:
        number: 2
        tags: row1
    led_04:
        number: 3
        tags: row2
    light_01:
        number: 0
        label: Test 0
        tags: tag1
        subtype: matrix
        debug: True
    light_02:
        number: 1
        label: Test 1
        tags: tag1
        subtype: matrix
        debug: True
    light_03:
        number: 2
        label: Test 1
        fade_ms: 1s
        subtype: matrix
        debug: True
    gi_01:
        number: 0
        subtype: gi
    flasher_01:
        platform: drivers
        number: flasher_01

coils:
    coil_01:
        number: 1
        default_pulse_ms: 30
    flasher_01:
        number: 2
```

(continues on next page)

(continued from previous page)

```
      label: Test flasher
      default_pulse_ms: 40
      max_hold_power: 1.0

shows:
  leds_name_token:
    - time: 0
      lights:
        (leds): red
  leds_single_color:
    - time: 0
      lights:
        led_01: (color)
  leds_color_token:
    - time: 0
      lights:
        led_01: (color1)
    - time: +1
      lights:
        led_02: (color2)
    - time: +1
  leds_extended:
    - time: 0
      lights:
        (leds):
          color: red
          fade: 1s
  lights_basic:
    - time: 0
      lights:
        (lights): ff
  multiple_tokens:
    - time: 0
      lights:
        (leds): blue
        (lights): ff
  show_assoc_tokens:
    - time: 0
      lights:
        (line1num): (line1color)
  show_with_time_and_duration:
    - time: +1s
    - time: 5s
    - time: +1s
      duration: 1s
    - lights:
        led_02: red
    - time: 10s
      duration: 3s
  leds_color_token_and_fade:
    - time: 0
      lights:
        led_01: (color1)
    - time: +1
```

(continues on next page)

(continued from previous page)

```
    lights:
      led_02: (color2)-f900ms
  - time: +1
manual_advance:
  - duration: -1
    lights:
      (leds): red
  - duration: -1
    lights:
      (leds): lime
  - duration: -1
    lights:
      (leds): blue
event_show:
  - duration: 1
    events:
      - step1
  - duration: 1
    events:
      - step2
  - duration: 1
    events:
      - step3
flash_multiple:
  - duration: -1
    shows:
      flash_color:
        show_tokens:
          leds: "{led1}, {led2}, {led3}"
          color: "{color}"
        speed: 4

show_player:
  flash_multiple_leds:
    flash_multiple:
      show_tokens:
        led1: led_01
        led2: led_02
        led3: led_03
        color: red
  play_on_led1:
    on:
      key: on_led_01
      show_tokens:
        lights: led_01
  play_on_led2:
    on:
      key: on_led2
      show_tokens:
        lights: led_02
  stop_on_led1:
    on_led_01: stop
  stop_on_led2:
    on_led2: stop
```

(continues on next page)

(continued from previous page)

```
play_test_show1: test_show1
play_with_priority:
  test_show1:
    priority: 15
play_with_speed:
  test_show1:
    speed: 2
play_with_start_step:
  test_show1:
    start_step: 2
play_with_neg_start_step:
  test_show1:
    start_step: -2
play_with_loops:
  test_show1:
    loops: 2
play_with_sync_ms_1000:
  test_show1:
    sync_ms: 1000
play_with_sync_ms_500:
  test_show1:
    sync_ms: 500
play_with_manual_advance:
  test_show1:
    manual_advance: True
pause_test_show1:
  test_show1:
    action: pause
resume_test_show1:
  test_show1:
    action: resume
stop_test_show1:
  test_show1: stop
play_show_assoc_tokens:
  show_assoc_tokens:
    speed: 1
    show_tokens:
       line1num: tag1
       line1color: red
stop_show_assoc_tokens:
  show_assoc_tokens:
    action: stop
test_mode_started:
  8linesweep:
    loops: 0
    speed: 1
    show_tokens:
       line1num: row0
       line1color: red
       line2num: row1
       line2color: orange
       line3num: row2
       line3color: yellow
       line4num: row2
```

(continues on next page)

(continued from previous page)

```
        line4color: green
        line5num: row2
        line5color: blue
        line6num: row2
        line6color: indigo
        line7num: row2
        line7color: violet
        line8num: row2
        line8color: midnightblue
test_mode_stopped:
  8linesweep:
    action: stop
play_manual_advance:
  manual_advance:
    show_tokens:
      leds: led_01
advance_manual_advance:
  manual_advance: advance
advance_manual_step_back:
  manual_advance: step_back
queue_play:
  event_show:
    block_queue: True
    action: play
    loops: 0
play_with_emitted_events:
  test_show1:
    events_when_played: test_show1_played, test_show1_played2
    events_when_stopped: test_show1_stopped
    events_when_looped: test_show1_looped
    events_when_paused: test_show1_paused
    events_when_resumed: test_show1_resumed
    events_when_advanced: test_show1_advanced
    events_when_stepped_back: test_show1_stepped_back
    events_when_completed: test_show1_completed
stop_emitted_events_show:
  test_show1: stop
pause_emitted_events_show:
  test_show1: pause
resume_emitted_events_show:
  test_show1: resume
advance_emitted_events_show:
  test_show1: advance
step_back_emitted_events_show:
  test_show1: step_back
play_with_completed_event:
  test_show1:
    events_when_completed: test_show1_completed
    events_when_stopped: test_show1_stopped
    loops: 0
play_show_with_token_in_key:
  test_show_key_token:
    show_tokens:
      num: "01"
```

(continues on next page)

(continued from previous page)

```
        color: red
  play_show_with_placeholder_in_token:
    test_show_key_token:
      show_tokens:
        num: (machine.test_num)
        color: (machine.test_color)
  play_show_with_condition_in_event{green==False}:
    leds_single_color:
      action: play
      show_tokens:
        color: purple
  play_show_with_condition_in_event{green==True}:
    leds_single_color:
      action: play
      show_tokens:
        color: green
  play_show_with_condition_in_show:
    leds_single_color{not blue}:
      action: play
      show_tokens:
        color: red
    leds_single_color{blue}:
      action: play
      show_tokens:
        color: blue
  play_show_with_placeholder_in_token_and_event_args:
    test_show_key_token:
      show_tokens:
        num: (test_num)
        color: (test_color)
```

Listing 340: your_machine_folder/shows/config/test_sync_ms.yaml

```
#config_version=5
lights:
  light:
    number: 1
  led_01:
    number:
  led_02:
    number:
  light_01:
    number:
  light_02:
    number:
  gi_01:
    number:
  gi_02:
    number:


shows:
  my_show1:
    - duration: -1
```

(continues on next page)

(continued from previous page)

```
      lights:
        light: red
  my_show2:
    - duration: -1
      lights:
        light: blue

show_player:
  play_show_sync_ms1:
    my_show1:
      key: sync_show
      sync_ms: 250
  play_show_sync_ms2:
    my_show2:
      key: sync_show
      sync_ms: 250
  stop_show:
    sync_show: stop
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 341: your_machine_folder/shows/modes/mode2/config/mode2.yaml

```
#config_version=5
mode:
  start_events: start_mode2
  stop_events: stop_mode2
  priority: 300
  start_priority: 1
  stop_on_ball_end: false
  game_mode: False

show_player:
  mode_mode2_started: test_show2

  mode_mode2_stopped:
    test_show2:
      action: stop
```

Listing 342: your_machine_folder/shows/modes/mode3/config/mode3.yaml

```
#config_version=5
mode:
  start_events: start_mode3
  stop_events: stop_mode3
  priority: 100
  start_priority: 1
```

(continues on next page)

(continued from previous page)

```
  stop_on_ball_end: false
  game_mode: False

show_player:
  mode_mode3_started: test_show3

  mode_mode3_stopped:
    test_show3:
      action: stop
```

Listing 343: your_machine_folder/shows/modes/mode1/config/mode1.yaml

```
#config_version=5
mode:
  start_events: start_mode1
  stop_events: stop_mode1
  priority: 200
  start_priority: 1
  game_mode: False
  stop_on_ball_end: false

show_player:
  mode_mode1_started:
    test_show1:
      loops: -1

  mode_mode1_stopped:
    test_show1:
      action: stop
  "{machine.test == 42}": show_from_mode
  "{machine.test == 23}":
    show_from_mode2:
      key: test_key1
    show_from_mode3:
      key: test_key2

shows:
  show_from_mode2:
    - duration: -1
  show_from_mode3:
    - duration: -1
  show_from_mode:
    - time: 0
      lights:
        (leds): red
    - time: 1
```

Listing 344: your_machine_folder/shows/modes/mode4/config/mode4.yaml

```
#config_version=5
mode:
  priority: 100
  game_mode: False
```

(continues on next page)

(continued from previous page)

```
show_player:
  test_token:
    test_show4:
      show_tokens:
        fade_time: 100

shows:
  test_show4:
    - lights:
        led_01:
          color: red
          fade: (fade_time)
```

Show file examples

Here are some example show files that go along with the above config(s).

Note that there are multiple shows here.

Listing 345: your_machine_folder/shows/shows/8linesweep.yaml

```
#show_version=5
- time: 0
  lights:
    (line1num): (line1color)
- time: +1s
  lights:
    (line1num): black
    (line2num): (line2color)
- time: +1s
  lights:
    (line2num): black
    (line3num): (line3color)
- time: +1s
  lights:
    (line3num): black
    (line4num): (line4color)
- time: +1s
  lights:
    (line4num): black
    (line5num): (line5color)
- time: +1s
  lights:
    (line5num): off
    (line6num): (line6color)
- time: +1s
  lights:
    (line6num): off
    (line7num): (line7color)
- time: +1s
  lights:
    (line7num): off
```

(continues on next page)

(continued from previous page)

```
    (line8num): (line8color)
- time: +1s
  lights:
    (line8num): off
- time: +1s
```

Listing 346: your_machine_folder/shows/shows/test_show_key_token.yaml

```
#show_version=5
- duration: -1
  lights:
    led_(num): (color)
```

Listing 347: your_machine_folder/shows/shows/test_show1.yaml

```
#show_version=5
- time: 0
  lights:
    led_01: 006400
    led_02: CCCCCC
    light_01: CC
    light_02: 78
    gi_01: FF
- time: 1
  lights:
    led_01: DarkGreen
    led_02: Black
- time: 2
  lights:
    led_01: DarkSlateGray
    led_02: Tomato
    light_01: FF
    light_02: 33
    gi_01: 99
- time: +1
  lights:
    led_01: MidnightBlue-f500 ms
    led_02: DarkOrange-f0.5 s
    gi_01: 33
- time: 4
  lights:
    led_01: Off-f800
    led_02: Off-f800
    light_01: 00-f800
    light_02: 00-f800
    gi_01: 00
- time: 6
```

Listing 348: your_machine_folder/shows/shows/test_show2.yaml

```
#show_version=5
- time: 0
  events:
```

(continues on next page)

(continued from previous page)

```
    test_event:
    test_event2:
    play_sound: {"sound": "test_1", "volume": 0.5, "loops": -1}
- time: 1
  events:
    play_sound: {"sound": "test_2"}
- time: 2
  events:
    play_sound: {"sound": "test_3", "volume": 0.35, "loops": 1}
- time: 3
```

Listing 349: your_machine_folder/shows/shows/myparentshow.yaml

```
#show_version=5
- duration: -1
  shows:
    mychildshow:
      speed: 1
      loops: 0
```

Listing 350: your_machine_folder/shows/shows/test_variable_show.yaml

```
#show_version=5
- time: 0
  variables:
    foo:
      action: set_machine
      int: 0
- time: 1
  variables:
    foo:
      action: add_machine
      int: 1
```

Listing 351: your_machine_folder/shows/shows/test_show3.yaml

```
#show_version=5
- time: 0
  flashers: flasher_01
- time: 1
  coils:
      coil_01: pulse
- time: 2
  coils:
      coil_01:
          pulse_power: .45
- time: 3
```

Listing 352: your_machine_folder/shows/shows/on_demand/mychildshow.yaml

```
#show_version=5
- time: 0
  events: test
```

slide (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 353: your_machine_folder/slide/config/test_slides.yaml

```
#config_version=5

modes:
 - mode1

displays:
  display1:
    width: 401
    height: 301
  display2:
    width: 402
    height: 302
    default: true


slides:
  slide1:
  - type: text
    text: SLIDE TEST 1-1
    y: -50
    color: ff0000
    font_size: 50
  - type: text
    text: SLIDE TEST 1-2
    color: 00ff00
    font_size: 50
  - type: text
    text: SLIDE TEST 1-3
    y: 50
    color: 0000ff
    font_size: 50
  slide2:
    type: text
    text: SLIDE TEST 2-1
    color: 00ffff
    font_size: 50
```

(continues on next page)

(continued from previous page)

```
slide3:
  widgets:
    type: text
    text: SLIDE TEST 3-1
    color: 00ff00
    font_size: 50
slide4:
  widgets:
    type: text
    text: SLIDE TEST 4-1
    color: ffff00
    font_size: 50
  transition: move_in
slide5:
  widgets:
    type: text
    text: SLIDE TEST 5-1
    color: ffaa00
    font_size: 50
  transition:
    type: move_in
    direction: right
slide6:
  background_color: ff0000ff
  opacity: 0.5
  widgets:
    type: text
    text: TEST BACKGROUND COLOR & OPACITY
slide7:
- type: text
  text: TEST Z-ORDER 50-1
  y: -50
  z: 50
  color: ff0000
  font_size: 50
- type: text
  text: TEST Z-ORDER 100
  z: 100
  color: 00ff00
  font_size: 50
- type: text
  text: TEST Z-ORDER 0
  y: 50
  color: 0000ff
  font_size: 50
- type: text
  text: TEST Z-ORDER 50-2
  y: 75
  z: 50
  color: ffff00
  font_size: 50
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 354: your_machine_folder/slide/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  priority: 500

slides:
  mode1_slide1:
    type: text
    text: MODE 1 SLIDE 1
    x: 25%
    color: ffaa00
    font_size: 100
```

slide_player (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 355: your_machine_folder/slide_player/config/test_slide_player.yaml

```
#config_version=5
modes:
 - mode1

displays:
  display1:
    height: 400
    width: 300
  display2:
    height: 400
    width: 300
  display3:
    height: 400
    width: 300
    enabled: false

slides:
  slide_with_var:
    - type: text
      text: SLIDE WITH VAR (test)
  slide_condition_foo:
    - type: text
      text: Conditional Slide (FOO)
  slide_condition_bar:
```

(continues on next page)

(continued from previous page)

```
  - type: text
    text: Conditional Slide (BAR)
machine_slide_1:
  - type: text
    text: TEST SLIDE PLAYER - SLIDE 1
    color: ff0000
    font_size: 100
  - type: rectangle
    width: 400
    height: 300
    color: blue
machine_slide_2:
  - type: text
    text: TEST SLIDE PLAYER - SLIDE 2
    color: ffaa00
    font_size: 100
  - type: rectangle
    width: 400
    height: 300
    color: purple
machine_slide_3:
  - type: text
    text: TEST SLIDE PLAYER - SLIDE 3
    color: 00ff00
    font_size: 100
  - type: rectangle
    width: 400
    height: 300
    color: yellow
machine_slide_4:
  - type: text
    text: TEST SLIDE PLAYER - SLIDE 4
    color: 0000ff
    font_size: 100
  - type: rectangle
    width: 400
    height: 300
    color: pink
machine_slide_5:
  - type: text
    text: TEST SLIDE PLAYER - SLIDE 5
    color: ff00ff
    font_size: 100
  - type: rectangle
    width: 400
    height: 300
    color: green
machine_slide_6:
  - type: text
    text: BASE SLIDE
  - type: rectangle
    width: 400
    height: 300
    color: blue
```

(continues on next page)

(continued from previous page)

```
  machine_slide_7:
    widgets:
      - type: text
        text: EXPIRE 1s
        color: red
      - type: rectangle
        width: 400
        height: 300
        color: yellow
    expire: 1s
  machine_slide_8:
    widgets:
    - type: text
      text: EXPIRE 1s
      color: purple
      y: 66%
    - type: text
      text: WITH TRANSITION OUT
      color: purple
      y: 33%
    - type: rectangle
      width: 400
      height: 300
      color: orange
    expire: 1s
    transition_out: wipe
  machine_slide_9:
    widgets:
    - type: text
      text: TRANSITION IN
    - type: rectangle
      width: 400
      height: 300
      color: lime
    transition: move_in
  machine_slide_10:   # used for test_SlidePlayer::test_animation_triggers
    widgets:
    - type: text
      text: WIDGET 1
      animations:
        flash_widget_1:
        - property: opacity
          value: 1
          duration: .25s
        - property: opacity
          value: 0
          duration: .25s
          repeat: yes

slide_player:
  show_slide_1: machine_slide_1
  show_slide_2:
    machine_slide_2:
      target: display1
```

(continues on next page)

(continued from previous page)

```
show_slide_3:
  machine_slide_3:
    target: display2
show_slide_4: machine_slide_4
show_slide_5: machine_slide_5
show_slide_4_p200:
  machine_slide_4:
    priority: 200
show_slide_1_force:
  machine_slide_1:
    force: true
show_slide_display3:
  machine_slide_1:
      target: display3
anon_slide_dict:
  slide_6:
    type: text
    text: TEXT FROM SLIDE_PLAYER DICT
    color: ff00ff
    font_size: 15
anon_slide_list:
  slide_7:
    - type: text
      text: TEXT FROM SLIDE_PLAYER LIST
      color: red
      font_size: 15
      y: 66%
    - type: text
      text: WIDGET 2
      color: purple
      font_size: 15
      y: 33%
anon_slide_widgets:
  slide_8:
    widgets:
    - type: text
      text: TEXT FROM SLIDE_PLAYER WIDGET LIST
      color: green
      font_size: 15
      y: 66%
    - type: text
      text: WIDGET 2
      color: lime
      font_size: 15
      y: 33%
    target: display1
    transition: move_in
anon_slide_widgets2:
  slide_8:
    widgets:
    - type: text
      text: Another text
      color: green
      font_size: 15
```

(continues on next page)

(continued from previous page)

```
      y: 66%
    target: display1
    transition: none
base_slide_no_expire: machine_slide_6
new_slide_expire:
  machine_slide_1:
    expire: 1s
show_slide_7: machine_slide_7
show_slide_8: machine_slide_8
show_slide_9: machine_slide_9
show_slide_5_with_transition:
  machine_slide_5:
    transition: fade
show_slide_9_with_transition:
  machine_slide_9:
    transition: fade
slide_2_dont_show:
  machine_slide_2:
    show: no
remove_slide_4:
  machine_slide_4:
    action: remove
remove_slide_4_with_transition:
  machine_slide_4:
    action: remove
    transition: wipe
remove_slide_8:
  machine_slide_8:
    action: remove
remove_slide_8_fade:
  machine_slide_8:
    action: remove
    transition: fade
slide1_expire_1s:
  machine_slide_1:
    expire: 1s
slide2_expire_1s:
  machine_slide_2:
    expire: 1s
random_player_with_animations:  # used for test_SlidePlayer::test_animation_triggers
  random_slide:
    widgets:
    - type: text
      text: WIDGET 1
      animations:
        flash_widget_2:
        - property: opacity
          value: 1
          duration: .25s
        - property: opacity
          value: 0
          duration: .25s
          repeat: yes
show_slide_with_animations:
```

(continues on next page)

(continued from previous page)

```
    my_slide:
      widgets:
      - type: text
        text: WIDGET 1
        animations:
          pre_show_slide:
          - property: opacity
            value: 1
            duration: .25s
          - property: opacity
            value: 0
            duration: .25s
            repeat: yes
  remove_slide_with_animations:
    my_slide: remove
  show_slide_with_var:
    slide_with_var:
      tokens:
        test: asd
  show_conditional_slide:
    slide_condition_foo{var=="foo"}:
      action: play
    slide_condition_bar{var=="bar"}:
      action: play
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 356: your_machine_folder/slide_player/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  priority: 500

slides:
  mode1_slide:
    type: text
    text: MODE 1 SLIDE 1
    color: 00ff00
    font_size: 100
  mode1_slide_2:
    type: text
    text: MODE 1 SLIDE 2
    color: 00ffff
    font_size: 150

slide_player:
  show_mode1_slide: mode1_slide
  remove_mode1_slide:
    mode1_slide: remove
```

(continues on next page)

(continued from previous page)

```
  show_mode1_slide_2:
    mode1_slide_2:
      priority: -350
```

smart_matrix (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 357: your_machine_folder/smart_matrix/config/config.yaml

```
#config_version=5
hardware:
  rgb_dmd: smartmatrix

smartmatrix:
  smartmatrix_1:
    port: com4
    baud: 3400000
  smartmatrix_2:
    port: com5
    baud: 3400000
    old_cookie: true

displays:
  dmd:
    width: 128
    height: 32

rgb_dmds:
  smartmatrix_1:
    hardware_brightness: .5
  smartmatrix_2:
    hardware_brightness: .5
```

Listing 358: your_machine_folder/smart_matrix/config/old_cookie.yaml

```
#config_version=5
config:
  - config.yaml
```

smart_virtual_platform (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 359: your_machine_folder/smart_virtual_platform/config/test_smart_virtual_initial.yaml

```
#config_version=5

config: test_smart_virtual.yaml

virtual_platform_start_active_switches:
  - device1_s1
```

Listing 360: your_machine_folder/smart_virtual_platform/config/test_entrance_switch.yaml

```
#config_version=5

switches:
  s_trough_1:
    number:
  s_trough_2:
    number:
  s_trough_3:
    number:
  s_entrance:
    number:

coils:
  c_trough_eject:
    number:

ball_devices:
  trough:
    tags: trough, home, drain
    entrance_switch: s_entrance
    ball_capacity: 3
    eject_coil: c_trough_eject

playfields:
    playfield:
        default_source_device: trough
        tags: default
```

Listing 361: your_machine_folder/smart_virtual_platform/config/test_coil_fired_plunger.yaml

```
#config_version=5

machine:
  balls_installed: 5
  min_balls: 1

```

(continues on next page)

(continued from previous page)

```
virtual_platform_start_active_switches: s_trough_1, s_trough_2, s_trough_3, s_trough_4, s_trough_5

switches:
  s_start:
    number: s13
    label:
    tags: start, player, high_score_select
  s_ball_launch:
    number: s11
    label:
    tags: plunger, player
  s_shooter_lane:
    number: s27
    label:
    tags:
  s_trough_1:
    number: s31
    label:
    tags:
    type: NC
  s_trough_2:
    number: s32
    label:
    tags:
    type: NC
  s_trough_3:
    number: s33
    label:
    tags:
    type: NC
  s_trough_4:
    number: s34
    label:
    tags:
    type: NC
  s_trough_5:
    number: s35
    label:
    tags:
    type: NC
  s_trough_jam:
    number: s36
    label:
    tags:
    type: NC
  s_standup:
    number: s38
    label:
    tags: playfield_active

coils:
  c_trough_eject:
    number: c01
    label:
```

(continues on next page)

(continued from previous page)

```
    tags:
    default_pulse_ms: 25
  c_plunger_lane:
    number: c03
    label:
    tags:
    default_pulse_ms: 25

ball_devices:
  trough:
    tags: trough, home, drain
    ball_switches: s_trough_1, s_trough_2, s_trough_3, s_trough_4, s_trough_5, s_trough_jam
    eject_coil: c_trough_eject
    confirm_eject_type: target
    eject_targets: shooter_lane
    jam_switch: s_trough_jam

  shooter_lane:
    ball_switches: s_shooter_lane
    eject_coil: c_plunger_lane
    player_controlled_eject_event: sw_plunger

playfields:
    playfield:
        default_source_device: shooter_lane
        tags: default
```

Listing 362: your_machine_folder/smart_virtual_platform/config/test_smart_virtual.yaml

```
#config_version=5

virtual_platform_start_active_switches:
  - trough1
  - trough2
  - trough3

smart_virtual:
    simulate_manual_plunger: True
    simulate_manual_plunger_timeout: 3s

coils:
    outhole:
        number: C09
        default_pulse_ms: 20
    trough:
        number: C10
        default_pulse_ms: 20
    trough2:
        number:
    plunger:
        number: 1
    device1:
      number: 2
    device2:
```

(continues on next page)

(continued from previous page)

```
      number: 3
    coil1:
      number:
    coil3:
      number:
    coil4:
      number:
    device3_c:
      number:
    device4_c:
      number:

switches:
    switch1:
      number:
    switch2:
      number:
    switch3:
      number:
    start:
        number: 1
        tags: start
    outhole:
        number: 2
    trough1:
        number: 3
    trough2:
        number: 4
    trough3:
        number: 5
    plunger:
        number: 6
    playfield:
        number: 7
        tags: playfield_active
    device1_s1:
      number: 8
    device1_s2:
      number: 9
    device2_s1:
      number: 10
    device2_s2:
      number: 11
    device3_s:
      number: 12
    device4_s:
      number: 13
    trough2_1:
      number:
    trough2_2:
      number:
    trough2_3:
      number:
    plunger2:
```

(continues on next page)

(continued from previous page)

```
      number:

drop_targets:
   left1:
     switch: switch1
   left2:
     switch: switch2
   left3:
     switch: switch3
     reset_coil: coil3
     knockdown_coil: coil4

drop_target_banks:
   left_bank:
     drop_targets: left1, left2
     reset_coils: coil1
     reset_events:
       drop_target_bank_left_bank_down: 1s

ball_devices:
    outhole:
        tags: drain
        ball_switches: outhole
        eject_coil: outhole
        eject_targets: trough
        confirm_eject_type: target
        debug: true
    trough:
        tags: trough, home
        ball_switches: trough1, trough2, trough3
        eject_coil: trough
        eject_targets: plunger
        confirm_eject_type: target
        debug: true
    plunger:
        tags: home
        ball_switches: plunger
        eject_coil: plunger
        debug: true
    device1:
        ball_switches: device1_s1, device1_s2
        eject_coil: device1
        eject_targets: device2
        confirm_eject_type: target
        tags: home # has to be home or attract will collect the balls
    device2:
        ball_switches: device2_s1 #, device2_s2
        confirm_eject_type: target
        mechanical_eject: true
    device3:
        tags: home
        entrance_switch: device3_s
        eject_coil: device3_c
        ball_capacity: 3
```

(continues on next page)

(continued from previous page)

```
        auto_fire_on_unexpected_ball: False
        debug: true
    device4:
        tags: home
        entrance_switch: device4_s
        eject_coil: device4_c
        ball_capacity: 3
        entrance_switch_full_timeout: 500ms
        auto_fire_on_unexpected_ball: False
        debug: true

    trough2:
        tags: drain, trough, home
        ball_switches: trough2_1, trough2_2, trough2_3
        eject_coil: trough2
        eject_targets: plunger2
        confirm_eject_type: target
        debug: true
    plunger2:
        ball_switches: plunger2
        mechanical_eject: True
        debug: true
```

smbus2 (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 363: your_machine_folder/smbus2/config/config.yaml

```
#config_version=5

hardware:
    platform: smbus2
```

snux (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 364: your_machine_folder/snux/config/config.yaml

```
#config_version=5
```

(continues on next page)

(continued from previous page)

```
hardware:
    platform: virtual
    driverboards: wpc
    coils: snux
    switches: snux

system11:
    ac_relay_delay_ms: 75
    ac_relay_driver: c_ac_relay

snux:
    diag_led_driver: c_diag_led_driver

coils:
    c_diag_led_driver:
        number: c24
        default_hold_power: 1.0
    c_flipper_enable_driver:
        number: c23
        default_hold_power: 1.0
    c_ac_relay:
        number: c25
        default_hold_power: 1.0
    c_side_a1:
        number: c11a
    c_side_a2:
        number: c12a
        default_hold_power: 0.5
    c_side_c1:
        number: c11c
    c_side_c2:
        number: c12c
        default_hold_power: 0.5
    c_flipper_left_main:
        number: FLLM
    c_flipper_left_hold:
        number: FLLH
        allow_enable: true

switches:
    s_flipper_left:
        number: sf01
    s_test:
        number: s77

flippers:
    f_test_single:
        main_coil: c_flipper_left_main
        hold_coil: c_flipper_left_hold
        activation_switch: s_flipper_left
```

spi_bit_bang (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 365: your_machine_folder/spi_bit_bang/config/config.yaml

```
#config_version=5

hardware:
    platform: virtual, spi_bit_bang

spi_bit_bang:
    miso_pin: s_miso
    cs_pin: o_cs
    clock_pin: o_clock


digital_outputs:
    o_cs:
        number: 1
        type: driver
    o_clock:
        number: 2
        type: driver

switches:
    s_trough_0:
        number: 0
        platform: spi_bit_bang
    s_trough_1:
        number: 1
        platform: spi_bit_bang
    s_trough_2:
        number: 2
        platform: spi_bit_bang
    s_trough_3:
        number: 3
        platform: spi_bit_bang
    s_trough_4:
        number: 4
        platform: spi_bit_bang
    s_trough_5:
        number: 5
        platform: spi_bit_bang
    s_trough_6:
        number: 6
        platform: spi_bit_bang
    s_trough_7:
        number: 7
        platform: spi_bit_bang
    s_miso:
```

(continues on next page)

(continued from previous page)

```
        number: 10
        platform: virtual
```

spike (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 366: your_machine_folder/spike/config/config.yaml

```
#config_version=5

hardware:
  platform: spike

spike:
   port: /dev/ttyUSB0
   baud: 115200
   debug: True
   nodes: 0, 1, 8, 9, 10, 11
   poll_hz: 10
   node_config:
     1:
        num_leds: 16
        num_inputs: 22
     8:
        coil_priorities: 0, 5, 6, 7, 1, 4, 3, 2
        num_leds: 56
        num_inputs: 16
     11:
        coil_priorities: 0, 1, 3, 5, 6, 7, 2, 4

coils:
  c_test:
    number: 1-0
    default_pulse_ms: 100
    default_hold_power: 0.625
  c_flipper_main:
    number: 8-1
    default_hold_power: 0.625
  c_flipper_hold:
    number: 8-3
    default_hold_power: 1.0
  c_pop:
    number: 8-10
    default_pulse_power: 0.5

lights:
  backlight:
```

(continues on next page)

(continued from previous page)

```
    number: 0-0
  l_1_3:
    number: 1-3
  l_8_3:
    number: 8-3
  l_8_30:
    number: 8-40
  l_rgb_insert:
    channels:
      red:
        number: 1-10
      green:
        number: 1-11
      blue:
        number: 1-12

switches:
  s_service:
    number: 0-13
  s_start:
    number: 1-11
  s_8_3:
    number: 8-3
  s_flipper:
    number: 8-13
  s_flipper_eos:
    number: 8-15
  s_pop:
    number: 8-4
  s_pop2:
    number: 8-5
    type: NC
  s_stepper_home:
    number: 10-1

autofire_coils:
  ac_pops:
    coil: c_pop
    switch: s_pop
  ac_pops2:
    coil: c_pop
    switch: s_pop2

flippers:
    f_test_single:
        main_coil: c_flipper_main
        activation_switch: s_flipper

    f_test_hold:
        main_coil: c_flipper_main
        hold_coil: c_flipper_hold
        activation_switch: s_flipper

    f_test_hold_eos:
```

(continues on next page)

(continued from previous page)

```
        main_coil: c_flipper_main
        hold_coil: c_flipper_hold
        activation_switch: s_flipper
        use_eos: True
        eos_switch: s_flipper_eos

    f_test_single_eos:
        main_coil: c_flipper_main
        activation_switch: s_flipper
        use_eos: True
        eos_switch: s_flipper_eos

dmds:
  spike_dmd:
    fps: 5

steppers:
  stepper1:
    number: 10-0
    homing_mode: switch
    homing_switch: s_stepper_home
    platform_settings:
      speed: 20
      light_number: 10-10
    named_positions:
      100: test_00
      200: test_01
      500: test_10
```

spinners (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 367: your_machine_folder/spinners/config/test_spinners.yaml

```
#config_version=5

switches:
  switch1:
    number:
  switch2:
    number:
  switch3:
    number:
  switch4:
    number:

spinners:
```

(continues on next page)

(continued from previous page)

```
  spin1:
    switch: switch1
    active_ms: 800
    idle_ms: 0
  spin2:
    switches: switch2, switch3
    labels: foo, bar
    active_ms: 400
    idle_ms: 800
  spin3:
    switches: switch4
    active_ms: 400
    idle_ms: 800
    reset_when_inactive: false
```

state_machine (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 368: your_machine_folder/state_machine/config/config.yaml

```
#config_version=5

modes:
  - game_mode
  - non_game_mode

state_machines:
  my_state:
    states:
      start:
        label: Start state
      step1:
        label:
        show_when_active:
          show: on
          show_tokens: None
        events_when_started: step1_start
        events_when_stopped: step1_stop
      step2:
        label:
    transitions:
      - source: start
        target: step1
        events: state_machine_proceed
      - source: step1
        target: step2
        events: state_machine_proceed2
```

(continues on next page)

(continued from previous page)

```
        events_when_transitioning: going_to_step2
      - source: step2
        target: start
        events: state_machine_proceed3
      - source: step1, step2
        target: start
        events: state_machine_reset
  second_state:
    starting_state: foo
    states:
      bar:
        label: Bar
      foo:
        label: Foo
    transitions:
      - source: foo
        target: bar
        events: state_machine_outoforder
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 369: your_machine_folder/state_machine/modes/non_game_mode/config/non_game_mode.yaml

```
#config_version=5
mode:
  start_events: machine_reset_phase_3
  game_mode: false

state_machines:
  non_game_mode_state_machine:
    persist_state: false
    states:
      start:
        label: Start state
      done:
        label: Done state
        events_when_started: non_game_mode_state_machine_done
    transitions:
      - source: start
        target: done
        events: non_game_mode_state_machine_proceed
```

Listing 370: your_machine_folder/state_machine/modes/game_mode/config/game_mode.yaml

```
#config_version=5
mode:
  start_events: ball_started
```

(continues on next page)

(continued from previous page)

```
state_machines:
  game_mode_state_machine:
    persist_state: true
    states:
      start:
        label: Start state
      done:
        label: Done state
        events_when_started: game_mode_state_machine_done
    transitions:
      - source: start
        target: done
        events: game_mode_state_machine_proceed
```

step_stick (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 371: your_machine_folder/step_stick/config/config.yaml

```
#config_version=5

hardware:
    platform: virtual
    stepper_controllers: step_stick


digital_outputs:
  c_direction:
    number: 1
    type: driver
  c_step:
    number: 2
    type: driver
  c_enable:
    number: 3
    type: driver

switches:
  s_home:
    number: 1

steppers:
  stepper1:
    number: c_direction:c_step:c_enable
    homing_mode: switch
    homing_switch: s_home
```

(continues on next page)

(continued from previous page)

```
    named_positions:
      10: test_00
      20: test_01
      50: test_10
```

stepper (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 372: your_machine_folder/stepper/config/config.yaml

```
#config_version=5

steppers:
    linearAxis_stepper:
        number: 1
        pos_min:   -5 #user units (negative is behind home flag)
        pos_max: 1000 #user units
        homing_direction: clockwise
        homing_mode: hardware
        reset_position: 0
        reset_events: test_reset
        debug: True
        named_positions:
            -5: test_00
            999: test_01
            500: test_10


# this is needed to test ball search
coils:
  coil1:
    number: 1

switches:
  switch1:
    number: 1

autofire_coils:
  ac_test:
    coil: coil1
    switch: switch1
```

switch_controller (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 373: your_machine_folder/switch_controller/config/config.yaml

```
#config_version=5

switches:
    s_test:
        number: 1
    s_test_events:
        number: 2
        events_when_activated: test_active|100ms, test_active2
        events_when_deactivated: test_inactive, test_inactive2|2s
    s_test_window_ms:
        number: 3
        ignore_window_ms: 100ms
    s_test_invert:
        number: 4
        type: 'NC'
```

switch_player (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 374: your_machine_folder/switch_player/config/config.yaml

```
#config_version=5

switches:
    s_test1:
        number:
        x: 0.4
        y: 0.5
        z: 0
    s_test2:
        number:
        x: 0.6
        y: 0.7
    s_test3:
        number:

plugins: switch_player

switch_player:
    start_event: test_start
    steps:
```

(continues on next page)

(continued from previous page)

```
    - time: 100ms
      switch: s_test1
      action: activate
    - time: 600ms
      switch: s_test3
      action: hit
    - time: 100ms
      switch: s_test1
      action: deactivate
    - time: 1s
      switch: s_test2
      action: activate
    - time: 1s
      switch: s_test3
      action: hit
    - time: 100ms
      switch: s_test2
      action: deactivate
    - time: 1s
      switch: s_test3
      action: hit
```

text (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 375: your_machine_folder/text/config/test_text.yaml

```
#config_version=5

modes:
  - mode1

displays:
  default:
    width: 400
    height: 300

slides:
  static_text:
    - type: text
      text: TEST
    - type: text
      text: STATIC TEXT
      y: 200
  text_from_event_param1:
    - type: text
      text: (param1)
```

(continues on next page)

(continued from previous page)

```
    - type: text
      text: TEXT FROM EVENT PARAMETER
      y: 200
      color: red
  text_from_event_param2:
    - type: text
      text: HI (param1)
    - type: text
      text: MIX STATIC AND DYNAMIC FROM EVENT
      y: 200
      color: orange
  text_from_event_param3:
    - type: text
      text: MIX (param1) MATCH
    - type: text
      text: MIX STATIC SURROUNDING DYNAMIC
      y: 200
      color: yellow
  text_from_event_param4:
    - type: text
      text: NO(param1)
    - type: text
      text: MIX WITH NO SPACE
      y: 200
      color: green
  text_from_event_param5:
    - type: text
      text: NUMBER (param1)
    - type: text
      text: DYNAMIC INTEGER
      y: 200
      color: lightblue
  text_from_event_param6:
    - type: text
      text: (param1)
    - type: text
      text: PURELY DYNAMIC NO MIX
      y: 200
      color: blue
  text_from_event_param7:
    - type: text
      text: 1)
    - type: text
      text: PARENTHESIS IN STRING
      y: 200
      color: pink
  text_from_event_param8:
    - type: text
      text: ((param1))
    - type: text
      text: COMBINE PARENTHESIS AND DYNAMIC
      y: 200
      color: purple
```

(continues on next page)

(continued from previous page)

```
text_with_player_var1:
  - type: text
    text: (test_var)
    font_size: 100
  - type: text
    text: TESTING WIDGET AUTO UPDATE
    y: 90
    color: pink
  - type: text
    text: FROM PLAYER VAR
    y: 70
    color: pink
text_with_player_var2:
  - type: text
    text: (player|test_var)
  - type: text
    text: DEFAULT PLAYER
    y: 200
    color: red
text_with_player_var3:
  - type: text
    text: (player1|test_var)
  - type: text
    text: NAMED PLAYER
    y: 200
    color: blue
text_with_player_var4:
  - type: text
    text: (player2|test_var)
  - type: text
    text: NAMED PLAYER THAT DOESN'T EXIST
    y: 200
    color: brown
text_with_player_var_and_event:
  - type: text
    text: (player_var) (test_param)
  - type: text
    text: MIX EVENT PARAM AND PLAYER VAR
    y: 200
    color: orange

text_string1:
  - type: text
    text: $greeting
  - type: text
    text: TEST text_string
    y: 200
    color: green
text_string2:
  - type: text
    text: $greeting $name
  - type: text
    text: TEST 2 text_strings
    y: 200
```

(continues on next page)

(continued from previous page)

```
    color: purple
text_string3:
  - type: text
    text: $money
  - type: text
    text: TEST text_string without dollar sign
    y: 200
    color: red
text_string4:
  - type: text
    text: $$dollar
  - type: text
    text: TEST text_string with dollar sign
    y: 200

number_grouping:
  - type: text
    text: 0
    min_digits: 2
    number_grouping: yes
  - type: text
    text: TEST NUMBER GROUPING & DOUBLE ZEROS
    y: 200

text_nocase:
  - type: text
    text: sAmPlE tExT caSiNg
  - type: text
    text: TEST CASING none
    y: 200
text_lower:
  - type: text
    text: sAmPlE tExT caSiNg
    casing: lower
  - type: text
    text: TEST CASING lower
    y: 200
text_upper:
  - type: text
    text: sAmPlE tExT caSiNg
    casing: upper
  - type: text
    text: TEST CASING upper
    y: 200
text_title:
  - type: text
    text: sAmPlE tExT caSiNg
    casing: title
  - type: text
    text: TEST CASING title
    y: 200
text_capitalize:
  - type: text
    text: sAmPlE tExT caSiNg
```

(continues on next page)

(continued from previous page)

```
      casing: capitalize
    - type: text
      text: TEST CASING capitalize
      y: 200
  text_line_break:
    - type: text
      text: "line\nbreak"
  text_bad_line_break:
    - type: text
      text: no line\nbreak

  mpfmc_font:
    - type: text
      text: MPF-MC FONT TEST
      font_name: pixelmix

  machine_font:
    - type: text
      text: TEST FONT FROM MACHINE FOLDER
      font_name: big_noodle_titling
  baseline:
    - type: text
      text: aaa
      x: 50
      y: 100
      anchor_y: bottom
    - type: text
      text: aaa
      x: 150
      y: 100
      anchor_y: baseline
    - type: text
      text: yyy
      x: 250
      y: 100
      anchor_y: bottom
    - type: text
      text: yyy
      x: 350
      y: 100
      anchor_y: baseline
    - type: line
      points: 0, 100, 800, 100
      color: red

slide_player:
  static_text: static_text
  text_from_event_param1: text_from_event_param1
  text_from_event_param2: text_from_event_param2
  text_from_event_param3: text_from_event_param3
  text_from_event_param4: text_from_event_param4
  text_from_event_param5: text_from_event_param5
  text_from_event_param6: text_from_event_param6
  text_from_event_param7: text_from_event_param7
```

(continues on next page)

(continued from previous page)

```
  text_from_event_param8: text_from_event_param8
  text_with_player_var1: text_with_player_var1
  text_with_player_var2: text_with_player_var2
  text_with_player_var3: text_with_player_var3
  text_with_player_var4: text_with_player_var4
  text_with_player_var_and_event: text_with_player_var_and_event
  number_grouping: number_grouping
  text_nocase: text_nocase
  text_lower: text_lower
  text_upper: text_upper
  text_title: text_title
  text_capitalize: text_capitalize
  text_string1: text_string1
  text_string2: text_string2
  text_string3: text_string3
  text_string4: text_string4
  mpfmc_font: mpfmc_font
  machine_font: machine_font
  baseline: baseline
  text_line_break: text_line_break
  text_bad_line_break: text_bad_line_break

text_strings:
  greeting: HELLO
  ball: (ball)
  name: PLAYER
  dollar: 100
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 376: your_machine_folder/text/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  priority: 100

text_strings:
  greeting: HELLO FROM MODE 1

slides:
  text_string1_mode1:
    - type: text
      text: $greeting

slide_player:
  text_string1_mode1: text_string1_mode1
```

text_input (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 377: your_machine_folder/text_input/config/text_input.yaml

```
#config_version=5

displays:
  default:
    width: 400
    height: 300

slides:
  slide1:
  - type: text
    text: TEXT HIGH SCORE ENTRY
    color: red
    y: top-5
    anchor_y: top
  - type: text_input
    initial_char: C
    key: key1
    style: score_entry
    animations:
      show_slide:
      - property: opacity
        value: 1
        duration: .25s
      - property: opacity
        value: 0
        duration: .25s
        repeat: yes
    block_events: test_block
    release_events: test_release
  - type: text
    text: ""
    key: key1
    style: score_entry

widget_styles:
  score_entry:
    font_size: 50

slide_player:
  slide1: slide1
```

tilt (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 378: your_machine_folder/tilt/config/config_system_11_trough.yaml

```
#config_version=5

config:
    - settings.yaml

game:
    balls_per_game: 3
    allow_start_with_ball_in_drain: True

modes:
    - tilt
    - base

playfields:
    playfield:
        default_source_device: bd_plunger
        tags: default

coils:
    c_outhole:
        number:
        default_pulse_ms: 20
    c_trough:
        number:
        default_pulse_ms: 20

switches:
    s_start:
        number:
        tags: start
    s_outhole:
        number:
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch3:
        number:
    s_plunger:
        number:
    s_playfield:
        number:
        tags: playfield_active
    s_tilt:
        number:
```

(continues on next page)

(continued from previous page)

```
        tags: tilt
    s_tilt_warning:
        number:
        tags: tilt_warning
    s_slam_tilt:
        number:
        tags: slam_tilt

ball_devices:
    bd_outhole:
        tags: drain
        ball_switches: s_outhole
        eject_coil: c_outhole
        eject_targets: bd_trough
        confirm_eject_type: target
        debug: true
    bd_trough:
        tags: trough, home
        ball_switches: s_ball_switch1, s_ball_switch2, s_ball_switch3
        eject_coil: c_trough
        eject_targets: bd_plunger
        confirm_eject_type: target
        debug: true
    bd_plunger:
        ball_switches: s_plunger
        mechanical_eject: true
        eject_timeouts: 4s
        debug: true
```

Listing 379: your_machine_folder/tilt/config/config.yaml

```
#config_version=5

config:
    - settings.yaml

modes:
    - tilt
    - base

game:
    balls_per_game: 2

playfields:
    playfield:
        default_source_device: bd_launcher
        tags: default

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:
    c_flipper:
```

(continues on next page)

(continued from previous page)

```
        number:
        default_hold_power: 0.125

switches:
    s_start:
        number:
        tags: start
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch_launcher:
        number:
    s_tilt:
        number:
        tags: tilt
    s_tilt_warning:
        number:
        tags: tilt_warning
    s_slam_tilt:
        number:
        tags: slam_tilt
    s_flipper:
        number:

ball_devices:
    bd_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2
        debug: true
        confirm_eject_type: target
        eject_targets: bd_launcher
        tags: trough, drain, home
    bd_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
        debug: true
        confirm_eject_type: target
        eject_timeouts: 6s, 10s

flippers:
    f_test:
        main_coil: c_flipper
        activation_switch: s_flipper
```

Listing 380: your_machine_folder/tilt/config/settings.yaml

```
#config_version=5

settings:
  warnings_to_tilt:
     label: Number of tilt warnings
     values:
        0: "no warnings"
```

(continues on next page)

(continued from previous page)

```
      1: "1"
      2: "2"
      3: "3"
      5: "5"
      10: "10"
    default: 3
    key_type: int
    sort: 600
  settle_time:
    label: Time to wait on tilt to settle bob
    values:
      3000: "3s"
      5000: "5s"
      10000: "10s"
    default: 5000
    key_type: int
    sort: 610
  multiple_hit_window:
    label: Tilt sensitivity
    values:
      150: "sensitive"
      300: "normal"
      500: "insensitive"
      1000: "very insensitive"
    default: 300
    key_type: int
    sort: 620
  shoot_again:
    label: Multiball Ball Save Timeout
    values:
      10: "10 Seconds (default)"
      20: "20 Seconds"
      30: "30 Seconds"
    default: 10000
    key_type: int
    sort: 630
```

Listing 381: your_machine_folder/tilt/config/config_mechanical_eject.yaml

```
#config_version=5

config:
    - settings.yaml

modes:
    - tilt
    - base

game:
    balls_per_game: 2

playfields:
    playfield:
        default_source_device: bd_launcher
```

(continues on next page)

(continued from previous page)

```
        tags: default

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:

switches:
    s_start:
        number:
        tags: start
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch_launcher:
        number:
    s_tilt:
        number:
        tags: tilt
    s_tilt_warning:
        number:
        tags: tilt_warning
    s_slam_tilt:
        number:
        tags: slam_tilt

ball_devices:
    bd_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2
        debug: true
        confirm_eject_type: target
        eject_targets: bd_launcher
        tags: trough, drain, home
    bd_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
        debug: true
        confirm_eject_type: target
        eject_timeouts: 6s, 10s
        mechanical_eject: True
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 382: your_machine_folder/tilt/modes/base/config/base.yaml

```
#config_version=5

mode:
  start_events: ball_starting
  priority: 100

variable_player:
  test_scoring:
    score: 100
```

Listing 383: your_machine_folder/tilt/modes/tilt/config/tilt.yaml

```
#config_version=5

tilt:
  reset_warnings_events: tilt_reset_warnings
  tilt_events: tilt_event
  multiple_hit_window: settings.multiple_hit_window
  settle_time: settings.settle_time
  warnings_to_tilt: settings.warnings_to_tilt
```

tilt_defaults (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 384: your_machine_folder/tilt_defaults/config/config.yaml

```
#config_version=5

modes:
    - tilt

playfields:
    playfield:
        default_source_device: bd_launcher
        tags: default

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:
    c_flipper:
        number:
        default_hold_power: 0.125
```

(continues on next page)

(continued from previous page)

```
switches:
    s_start:
        number:
        tags: start
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch_launcher:
        number:
    s_tilt:
        number:
        tags: tilt
    s_tilt_warning:
        number:
        tags: tilt_warning
    s_slam_tilt:
        number:
        tags: slam_tilt
    s_flipper:
        number:

ball_devices:
    bd_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2
        debug: true
        confirm_eject_type: target
        eject_targets: bd_launcher
        tags: trough, drain, home
    bd_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
        debug: true
        confirm_eject_type: target
        eject_timeouts: 6s, 10s

flippers:
    f_test:
        main_coil: c_flipper
        activation_switch: s_flipper
```

timed_switches (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 385: your_machine_folder/timed_switches/config/timed_switches.yaml

```
#config_version=5

modes:
 - mode1

switches:
  switch1:
    number:
  switch2:
    number:
  switch3:
    number:
  switch4:
    number:
    tags: left_flipper
  switch5:
    number:
    tags: right_flipper

timed_switches:
  group1:
    switches: switch1, switch2
    time: 2s
  another_one:
    switches: switch3
    time: 2000ms
    state: inactive
    events_when_active: active_event
    events_when_released: release_event
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 386: your_machine_folder/timed_switches/modes/mode1/config/mode1.yaml

```
#config_version=5
mode:
  game_mode: False

timed_switches:
  mode_switch:
    switches: switch2, switch3
    time: 2s
```

timer (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 387: your_machine_folder/timer/config/test_timer.yaml

```
#config_version=5

modes:
  - mode_with_timers
  - mode_with_timers2
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 388: your_machine_folder/timer/modes/mode_with_timers2/config/mode_with_timers2.yaml

```
#config_version=5
mode:
    start_events: player_turn_started
    stop_events: stop_mode_with_timers2
timers:
    timer_start_with_game:
        debug: True
        start_value: 0
        end_value: 10
        direction: up
        tick_interval: 1s
        start_running: yes
    timer_with_player_var_control_events:
        start_value: 0
        control_events:
            - action: start
              event: start_player_var_timer
            - action: add
              event: add_player_var_timer
              value: current_player.timer_amount
            - action: subtract
              event: subtract_player_var_timer
              value: current_player.timer_amount
```

Listing 389: your_machine_folder/timer/modes/mode_with_timers/config/mode_with_timers.yaml

```
#config_version=5
mode:
    start_events: start_mode_with_timers
    stop_events: stop_mode_with_timers
    game_mode: false
timers:
```

(continues on next page)

(continued from previous page)

```
timer_down:
    debug: True
    bcp: True
    start_value: 5
    end_value: 0
    direction: down
    tick_interval: 1.5s
    start_running: no
    control_events:
        - event: start_timer_down
          action: start
        - event: reset_timer_down
          action: reset
        - event: pause_timer_down
          action: pause
          value: 2
        - event: add_timer_down
          action: add
          value: 2
        - event: subtract_timer_down
          action: subtract
          value: 2
timer_start_running:
    debug: True
    start_value: 0
    end_value: 10
    direction: up
    tick_interval: 1s
    start_running: yes
timer_restart_on_complete:
    debug: True
    start_value: 0
    end_value: 5
    direction: up
    tick_interval: 1s
    start_running: yes
    restart_on_complete: yes
timer_up:
    bcp: True
    debug: True
    start_value: 0
    end_value: 10
    max_value: 15
    direction: up
    tick_interval: 1s
    start_running: no
    control_events:
        - event: start_timer_up
          action: start
        - event: reset_timer_up
          action: reset
        - event: stop_timer_up
          action: stop
        - event: restart_timer_up
```

(continues on next page)

(continued from previous page)

```
            action: restart
          - event: jump_timer_up
            action: jump
            value: 5
          - event: jump_over_max_timer_up
            action: jump
            value: 20
          - event: add_timer_up
            action: add
            value: 2
          - event: change_tick_interval_timer_up
            action: change_tick_interval
            value: 4
          - event: set_tick_interval_timer_up
            action: set_tick_interval
            value: 2
          - event: reset_tick_interval
            action: reset_tick_interval
    timer_player_var:
        debug: True
        start_value: current_player.start
        end_value: current_player.end
        direction: up
        tick_interval: 1s
        start_running: yes
    timer_change_tick:
        start_value: 30
        end_value: 0
        tick_interval: 1s
        direction: down
        start_running: no
        control_events:
          - event: timer_change_tick_start
            action: start
          - event: timer_change_tick_event
            action: change_tick_interval
            value: 0.1
          - event: timer_set_tick_event_fixed
            action: set_tick_interval
            value: 0.2
          - event: timer_set_tick_event_kwarg
            action: set_tick_interval
            value: event_value
```

transitions (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 390: your_machine_folder/transitions/config/test_transitions.yaml

```
#config_version=5

displays:
  default:
    width: 400
    height: 300

slides:
  slide1:
    - type: text
      text: TRANSITION TEST
      y: 33%
      color: ff0000
      font_size: 50
    - type: text
      text: ========== SLIDE 1 ===========
      y: 66%
      color: ff0000
      font_size: 50
    - type: rectangle
      width: 400
      height: 300
      color: 330000

  slide2:
    - type: text
      text: TRANSITION TEST
      color: 00ff00
      font_size: 50
      y: 33%
    - type: text
      text: ---------- SLIDE 2 -----------
      color: 00ff00
      font_size: 50
      y: 66%
    - type: rectangle
      width: 400
      height: 300
      color: 003300

slide_player:
  show_slide1: slide1
  show_slide2:
    slide2:
      transition:
        type: push
        easing: out_bounce
        duration: 2s
        direction: right
  push_left:
    slide2:
      transition:
```

(continues on next page)

(continued from previous page)

```
      type: push
      direction: left
push_right:
  slide2:
    transition:
      type: push
      direction: right
push_up:
  slide2:
    transition:
      type: push
      direction: up
push_down:
  slide2:
    transition:
      type: push
      direction: down
move_in_left:
  slide2:
    transition:
      type: move_in
      direction: left
move_in_right:
  slide2:
    transition:
      type: move_in
      direction: right
move_in_top:
  slide2:
    transition:
      type: move_in
      direction: top
move_in_bottom:
  slide2:
    transition:
      type: move_in
      direction: bottom
move_out_left:
  slide2:
    transition:
      type: move_out
      direction: left
move_out_right:
  slide2:
    transition:
      type: move_out
      direction: right
move_out_top:
  slide2:
    transition:
      type: move_out
      direction: top
move_out_bottom:
  slide2:
```

(continues on next page)

(continued from previous page)

```
      transition:
        type: move_out
        direction: bottom
  wipe:
    slide2:
      transition:
        type: wipe
  swap:
    slide2:
      transition:
        type: swap
  fade:
    slide2:
      transition:
        type: fade
  fade_back:
    slide2:
      transition:
        type: fade_back
  rise_in:
    slide2:
      transition:
        type: rise_in
#  no_transition_1:
#    slide2
#    transition: None
#  no_transition_2:
#    slide2
#    transition: false
#  no_transition_3:
#    slide2
#    transition:
#      type: none

  show_slide1_with_push:
    slide1:
      transition:
        type: push
        direction: right
  show_slide2_no_transition: slide2
```

trinamics_steprocker (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 391: your_machine_folder/trinamics_steprocker/config/trinamics_steprocker.yaml

```
#config_version=5

hardware:
  platform: virtual
  driverboards: virtual
  stepper_controllers: trinamics_steprocker

trinamics_steprocker:
  port: /dev/ttyACM0

steppers:
    #Scenario: 1.8 stepper that spins something continuously when activated; that you want to control in units of degrees
    #TODO: Add limit switches
#    velocityStepper:
#        number: 0
#        mode: velocity
#        move_current:  25 #percent
#        hold_current:  5 #percent
#        microstep_per_fullstep: 16 # 1/16 mode (1 step = 1/16 of a full step)
#        fullstep_per_userunit: 0.55 # UU = 1 Degree =  1 / 1.8 Degrees per Fullstep
#        velocity_limit: 360 #user units/sec   (so, 360 degrees per Sec)
#        acceleration_limit: 720 #user units/sec^2  (so, 720 degrees per Sec^2)


    # Scenario: 1.8 degree stepper attached to a 7:1 gear ratio with homing flag that you want to control in units of revolutions
    positionStepper:
        number: 0
        reset_position: 0
        reset_events: test_reset
        homing_direction: clockwise #when facing the shaft
        homing_mode: hardware
        named_positions:
             0.0: test_00
             0.6: test_01
             1.0: test_10
        platform_settings:
             move_current:  25 #percent
             hold_current:  5 #percent
             homing_speed: 0.1 #user units/sec
             microstep_per_fullstep: 16 # 1/16 mode (1 step = 1/16 of a full step)
             fullstep_per_userunit: 1400 # UU=1 Revolution = 200 full steps per rev (1.8 deg stepper) * 7 gear ratio
             velocity_limit: 0.5 #user units/sec   (so, 0.8 RPS of output gear )
             acceleration_limit: 2.0 #user units/sec^2  (so, 2 RPS^S of output gear)
```

twitch_client (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 392: your_machine_folder/twitch_client/config/config.yaml

```
#config_version=5

twitch_client:
  user: jan
  password: MPF
  channel: MPF-rocks
```

utils (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 393: your_machine_folder/utils/config/test_utils.yaml

```
#config_version=5

slides:
  slide1:
    - type: text
      text: widget1
      z: 100
    - type: text
      text: widget2
      z: 50
    - type: text
      text: widget3

slide_player:
  show_slide1:
    slide: slide1
```

variable_player (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 394: your_machine_folder/variable_player/config/config.yaml

```
#config_version=5

switches:
    s_counter_target:
        number:
    s_kills_counter_target:
        number:

modes:
  - mode1
  - mode2
  - mode3
  - mode_for_logic_block
  - non_game_mode
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Note that there are multiple mode config examples here. You might not necessarily use more than one in your machine.

Listing 395: your_machine_folder/variable_player/modes/mode2/config/mode2.yaml

```
#config_version=5
mode:
    start_events: start_mode2
    stop_events: stop_mode2
    priority: 300
    restart_on_next_ball: True

variable_player:
    test_event1:
        score: 1000|block
        var_a: 0|block
        var_b: 1
        var_c: current_player.ramps * 10|block
    test_score_mode:
        score:
           int: 1000
           block: true
```

Listing 396: your_machine_folder/variable_player/modes/mode_for_logic_block/config/mode_for_log

```
#config_version=5
# actived when all 5 drop targets have dropped
# user wants to continue hitting those
# hitting the special kills the mode
mode:
  start_events: counter_target_complete # from logic_block
  # priority higher that mode1 priority
```

(continues on next page)

(continued from previous page)

```
  priority: 300
  stop_events: s_kills_counter_target_active

variable_player:
  s_counter_target_active:
    score: 100|block
  s_kills_counter_target_active:
    score: 500|block
```

Listing 397: your_machine_folder/variable_player/modes/non_game_mode/config/non_game_mode.y

```
#config_version=5
mode:
    start_events: start_non_game_mode
    game_mode: False
    priority: 200

variable_player:
  test_event:
    test:
      string: "123"
      action: set_machine
    test2:
      int: 7
      action: add_machine
  "{machine.test5}":
    test6:
      action: set_machine
      string: "{value}-suffix"
```

Listing 398: your_machine_folder/variable_player/modes/mode3/config/mode3.yaml

```
#config_version=5
mode:
    start_events: start_mode3
    stop_events: stop_mode3
    priority: 400

variable_player:
    score_player1:
        score:
            int: 42
            player: 1
    score_player2:
        score:
            int: 23
            player: 2
    reset_player2:
        score:
            int: 10
            player: 2
            action: set
    score_float2:
```

(continues on next page)

(continued from previous page)

```
        score:
            float: 2.0
    score_float3:
        score: 100 * current_player.multiplier
    set_float:
        multiplier:
            float: 1.5
            action: set
    score_floordiv:
        score: 123456 // 100 * 100
    set_player7:
        score:
            int: 10
            player: 7
            action: set
    add_player7:
        score:
            int: 10
            player: 7
            action: add
```

Listing 399: your_machine_folder/variable_player/modes/mode1/config/mode1.yaml

```
#config_version=5
mode:
    start_events: start_mode1
    stop_events: stop_mode1
    priority: 200

variable_player:
    test_event1:
        score: 100
        var_a: 1
        var_c: current_player.ramps
    test_set_100:
        test1:
          int: 100
          action: set
    test_set_200:
        test1:
          int: 200
          action: set
    test_set_string:
        string_test:
          action: set
          string: HELLO
    test_set_machine_var:
        my_var:
          int: 100
          action: set_machine
    test_add_machine_var:
        my_var:
          int: 23
          action: add_machine
```

(continues on next page)

(continued from previous page)

```
player_score:
    my_var2:
      int: change
      action: add_machine
test_score_mode:
    score: 100
s_counter_target_active:
    score: 10
s_kills_counter_target_active:
    score: 100
```

video (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 400: your_machine_folder/video/config/test_video.yaml

```
#config_version=5

displays:
  default:
    width: 400
    height: 300

modes:
  - mode1

slides:
  video_test:
    - type: video
      video: mpf_video_small_test
    - type: text
      text: Video Test
      y: bottom+20%
    - type: text
      text: ""
      y: bottom+10%
  video_test2:
    - type: video
      video: mpf_video_small_test
      control_events:
        - event: play1
          action: play
        - event: stop1
          action: stop
        - event: pause1
          action: pause
        - event: seek1
```

(continues on next page)

(continued from previous page)

```
        action: seek
        value: .5
      - event: position1
        action: position
        value: 4
      - event: mute
        action: volume
        value: 0
  - type: text
    text: Video Control Events Test
    y: bottom+20%
  - type: text
    text: ""
    y: bottom+10%
video_test3:
  - type: video
    video: mpf_video_small_test
    control_events:
      - event: pre_show_slide
        action: seek
        value: .5
video_test4:
  - type: video
    video: mpf_video_small_test
    control_events:
      - event: show_slide
        action: seek
        value: .5
video_test5:
  - type: video
    video: mpf_video_small_test
    control_events:
      - event: pre_slide_leave
        action: seek
        value: .5
video_test6:
  - type: video
    video: mpf_video_small_test
    control_events:
      - event: slide_leave
        action: seek
        value: .5
video_test7:
  - type: video
    video: mpf_video_small_test
    auto_play: true
    end_behavior: loop
    volume: .2
    control_events:
      - event: seek1
        action: seek
        value: .9
video_test8:
  - type: video
```

(continues on next page)

(continued from previous page)

```
      video: mpf_video_small_test
      auto_play: false
      end_behavior: stop
      volume: 0.8
      control_events:
        - event: play1
          action: play
        - event: seek1
          action: seek
          value: .9
  video_test9:
    - type: text
      text: Machine slide, no video

slide_player:
  show_slide1: video_test
  show_slide2: video_test2
  show_slide3: video_test3
  show_slide4: video_test4
  show_slide5: video_test5
  show_slide6: video_test6
  show_slide7: video_test7
  show_slide8: video_test8
  show_slide9: video_test9

videos:
  mpf_video_small_test:
    width: 100
    height: 70
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 401: your_machine_folder/video/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  priority: 100


slide_player:
  mode_mode1_started:
    mode1_slide1:
      widgets:
      - type: video
        video: mpf_video_small_test
      - type: text
        text: Video from Mode
        y: bottom+20%
```

virtual_pinball (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 402: your_machine_folder/virtual_pinball/config/config.yaml

```
#config_version=5

hardware:
    platform: virtual

switches:
    s_test:
        number: 0-0
    s_test_no_debounce:
        number: 0-1
        debounce: quick
    s_test_nc:
        number: 0-2
        type: 'NC'
    s_flipper:
        number: 0-3
    s_test_card2:
        number: 0-8

coils:
    c_test:
        number: 0-0
        default_pulse_ms: 23
    c_test_allow_enable:
        number: 0-1
        default_pulse_ms: 23
        platform_settings:
            recycle_factor: 3
        default_hold_power: 1.0
    c_flipper_hold:
        number: 0-2
        default_hold_power: 1.0
    c_flipper_main:
        number: 0-3
        default_pulse_ms: 10
        default_hold_power: 0.375
    c_holdpower_16:
        number: 1-12
        default_hold_power: 0.0625

lights:
  test_light1:
    number: 0-16
    subtype: matrix
  test_light2:
```

(continues on next page)

(continued from previous page)

```
    number: 0-17
    subtype: matrix
  test_led1:
    number: 1-0
  test_led2:
    number: 1-1

autofire_coils:
    ac_slingshot_test:
        coil: c_test
        switch: s_test

    ac_slingshot_test2:
        coil: c_test_allow_enable
        switch: s_test_no_debounce

flippers:
    f_test_single:
        debug: true
        #main_coil_overwrite:
        #    pulse_ms: 11
        main_coil: c_flipper_main
        activation_switch: s_flipper

    f_test_hold:
        debug: true
        main_coil: c_flipper_main
        hold_coil: c_flipper_hold
        activation_switch: s_flipper
```

virtual_segment_display_connector (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 403: your_machine_folder/virtual_segment_display_connector/config/config.yaml

```
#config_version=5

segment_displays:
  display1:
    number: 1
  display2:
    number: 2
  display3:
    number: 3

virtual_segment_display_connector:
    segment_displays: display1, display2
```

vpe (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 404: your_machine_folder/vpe/config/config.yaml

```
#config_version=5

hardware:
    platform: visual_pinball_engine

playfields:
  playfield:
    tags: default
    default_source_device: None

vpe:
  debug: True

rgb_dmds:
  test_dmd:
    source_display: rgb_dmd_display

dmds:
  default:
    source_display: dmd_display

segment_displays:
    segment1:
        number: 0
        size: 10

switches:
    s_sling:
        number: 0
    s_flipper:
        number: 3
    s_test:
      number: 6

coils:
    c_sling:
        number: 0
    c_flipper:
        number: 1
        allow_enable: True
    c_test:
        number: 2
        allow_enable: True

lights:
```

(continues on next page)

(continued from previous page)

```
  test_light1:
    number: 0
  test_light2:
    number: 1

autofire_coils:
    ac_slingshot_test:
        coil: c_sling
        switch: s_sling

flippers:
    f_test:
        main_coil: c_flipper
        activation_switch: s_flipper
```

vpx (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 405: your_machine_folder/vpx/config/config.yaml

```
#config_version=5

hardware:
    platform: virtual_pinball

switches:
    s_sling:
        number: 0
    s_flipper:
        number: 3
    s_test:
        number: 6

coils:
    c_sling:
        number: 0
    c_flipper:
        number: 1
        allow_enable: True
    c_test:
        number: 2
        allow_enable: True

lights:
  test_light1:
    number: 0
    subtype: matrix
```

(continues on next page)

(continued from previous page)

```
  test_light2:
    number: 1
  test_flasher:
    number: 0
    subtype: flasher
  test_led1:
    number: 0
    subtype: led
  test_led2:
    number: 1
    subtype: led
  test_gi:
    number: 0
    subtype: gi

autofire_coils:
    ac_slingshot_test:
        coil: c_sling
        switch: s_sling

flippers:
    f_test:
        main_coil: c_flipper
        activation_switch: s_flipper
```

widget_styles (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Listing 406: your_machine_folder/widget_styles/config/test_widget_styles.yaml

```
#config_version=5

modes:
  - mode1

displays:
  default:
    width: 400
    height: 300

widget_styles:
  text_default:
    font_size: 21
    color: red
  bigStyle:
    font_size: 100
    halign: left
```

(continues on next page)

(continued from previous page)

```
  stackedStyle:
    color: blue

slides:
  slide1:
    - type: text
      text: HELLO
      style: bigStyle
      halign: right
    - type: text
      text: Default Style
      y: 100
  slide3:
    - type: text
      font_size: 30
      text: COLOR FROM DEFAULT STYLE
  slide4:
    - type: text
      text: TESTING INVALID STYLE
      style: bogus
  slide5:
    - type: text
      text: HELLO
      style: bigStyle
      font_size: 50
  slide6:
    - type: text
      text: HELLO TOO
      style: bigStyle, stackedStyle
  slide7:
    - type: text
      text: HELLO THREE
      style:
        - text_default
        - stackedStyle

slide_player:
  slide1: slide1
  slide3: slide3
  slide4: slide4
  slide5: slide5
  slide6: slide6
  slide7: slide7
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 407: your_machine_folder/widget_styles/modes/mode1/config/mode1.yaml

```
#config_version=5
```

(continues on next page)

(continued from previous page)

```
mode:
  priority: 100

widget_styles:
  text_default:
    font_size: 50

slides:
  slide2:
    - type: text
      text: MODE1 DEFAULT STYLE
      y: 75
    - type: text
      style: big
      text: BIG FROM BASE
      y: 225

slide_player:
  slide2: slide2
```

widgets (example config files)

Machine config examples

Here are some example machine-wide config files that show real-world examples of how these configs are used.

Note that there are multiple machine config examples here. They're just included to show different options. You wouldn't actually use more than one.

Listing 408: your_machine_folder/widgets/config/test_widgets.yaml

```
#config_version=5

modes:
- mode1

displays:
  default:
    width: 800
    height: 600

widgets:
  widget1:
    type: text
    text: widget1
    color: ffff00
    font_size: 100
    y: top-40%
  widget2:
  - type: text
```

(continues on next page)

(continued from previous page)

```
    text: widget2
    y: 50
    color: ff0000
    font_size: 100
  widget3:
  - type: text
    text: widget3.1
    color: ff0000
    font_size: 100
  - type: text
    text: widget3.2
    color: 00ff66
    font_size: 100
  - type: text
    text: widget3.3
    color: ff00ff
    font_size: 100
  widget4:
  - type: text
    text: widget4.1
    y: 300
    z: 1
    color: ff0000
    font_size: 100
  - type: text
    text: widget4.2
    z: 1000
    y: 250
    color: ffff00
    font_size: 100
  - type: text
    text: widget4.3
    y: 200
    color: 00ff00
    font_size: 100
  - type: text
    text: widget4.4
    z: 1
    y: 150
    color: 00ffff
    font_size: 100
  - type: text
    text: widget4.5
    z: 1000
    y: 100
    color: 0000ff
    font_size: 100
  - type: text
    text: widget4.6
    color: ff00ff
    font_size: 100
    y: 50
  - type: text
    text: widget4.7
```

(continues on next page)

(continued from previous page)

```
    y: 0
    color: 888888
    font_size: 100
  widget5:
    type: text
    text: widget5
    z: 200
    y: 150
    font_size: 100
  widget6:
    type: text
    text: widget6
    z: 100
    color: 774303
    font_size: 100
  widget7:
    type: text
    text: EXPIRES 1S (player|test)
    color: orange
    font_size: 100
    expire: 1s
    animations:
      test_event:
        - property: rotation
          value: 360
          duration: .5s
  widget8:
    type: text
    text: WIDGET 8
    color: orange
    font_size: 100
  box11:
     - type: text
       text: box11
  box12:
     - type: text
       text: box12
  box13:
     - type: text
       text: box13
  box14:
     - type: text
       text: box14
  widget9:
     - type: text
       text: named_widget9
       key: widget9_key
  widget10:
    type: text
    text: (text)
  widget_bezier:
    - type: bezier
      points: 400, 300, 100, 100, 400, 0
      color: red
```

(continues on next page)

(continued from previous page)

```
    thickness: 5
    animations:
      add_to_slide:
        - property: color, points
          value: 0, 1, 0, 1, 200, 200, 50, 100, 100, 250
          duration: 2s
        - property: rotation
          value: -300
          duration: 2s
          timing: with_previous
        - property: color
          value: 0, 0, 1, 1
          duration: 1s
        - property: color
          value: 1, 1, 0, 1
          duration: 1s
widget_ellipse:
  - type: ellipse
    width: 200
    height: 100
    angle_end: 0
    color: magenta
    opacity: 0.5
    animations:
      add_to_slide:
        - property: pos
          value: 100, 100
          relative: true
          duration: 2s
        - property: rotation
          value: 360
          duration: 2s
          timing: with_previous
widget_quad:
  - type: quad
    points: 300, 100, 350, 200, 500, 150, 450, 50
    color: cornflowerblue
    animations:
      add_to_slide:
        - property: points
          value: 50, -50, -50, 50, 50, -50, -50, 50
          duration: 1.5
          relative: true
        - property: rotation, scale
          value: -720, -0.9
          relative: true
          duration: 1.5s
          timing: with_previous
        - property: points
          value: -50, 50, 50, -50, -50, 50, 50, -50
          duration: 1.5
          relative: true
        - property: rotation, scale
          value: 720, 0.9
```

(continues on next page)

(continued from previous page)

```
          relative: true
          duration: 1.5s
          timing: with_previous
widget_rectangle:
  - type: rectangle
    x: 600
    y: 300
    width: 100
    height: 200
    color: gold
    animations:
      add_to_slide:
        - property: rotation, scale, corner_radius
          value: 540, 0.5, 50
          duration: 1.5s
        - property: scale
          value: 0.5
          duration: 1.0s
        - property: rotation, scale, corner_radius
          value: 0, 1.0, 0
          duration: 1.5s
widget_line:
  - type: line
    points: 200, 50, 600, 50
    thickness: 10
    color: darkcyan
    animations:
      add_to_slide:
        - property: rotation, scale
          value: 360, 1.5
          duration: 3s
widget_triangle:
  - type: triangle
    points: 100, 450, 100, 550, 200, 450
    color: lawngreen
    animations:
      add_to_slide:
        - property: rotation, scale
          value: -900, 1.5
          duration: 3s
          easing: in_quint
widget_points:
  - type: points
    points: 100, 450, 100, 550, 200, 450
    pointsize: 2
    color: deeppink
    animations:
      add_to_slide:
        - property: rotation, scale, pointsize
          value: 900, 1.5, 8
          duration: 3s
          easing: in_quint
widget_text:
  - type: text
```

(continues on next page)

(continued from previous page)

```
      text: TEST
      font_size: 50
      bold: true
      color: lightyellow
      animations:
        add_to_slide:
          - property: rotation, scale
            value: 45, 1.5
            duration: 2s
            easing: in_quint
          - property: scale
            value: 0.75
            duration: 1s
            easing: out_quint
  widget_reusable:
    - type: text
      text: Reusable Widget
  widget_placeholder_value1:
    - type: text
      text: Value One
  widget_placeholder_value2:
    - type: text
      text: Value Two
  widget_custom_events1:
    - type: text
      text: Testing events
      events_when_added: custom_events1_added
      events_when_removed: custom_events1_removed
  widget_custom_events2:
    - type: rectangle
      x: 600
      y: 300
      width: 100
      height: 200
      color: gold
      events_when_added: custom_events2_added, custom_events2_added_again
      events_when_removed: custom_events2_removed, custom_events2_removed_again

widget_player:
  add_widget1_to_current: widget1
  add_widget2_to_current: widget2
  add_widget2_to_slide1:
    widget2:
      slide: slide1
  update_widget2:
    widget2:
      action: update
      slide: slide1
  remove_widget2:
    widget2:
      action: remove
  add_widget6:
    widget6:
      widget_settings:
```

(continues on next page)

(continued from previous page)

```
      z: -1
remove_widget1_by_key:
  widget1:
    action: remove
    key: widget1
remove_widget1:
  widget1:
    action: remove
add_widget7: widget7
add_widget8_expire:
    widget8:
      widget_settings:
        expire: 1s
add_widget8_expire_parent:
    widget8:
      widget_settings:
        expire: 1s
        z: -1
add_widget8_custom_settings:
    widget8:
      widget_settings:
        color: red
        font_size: 70
        x: right-10
        anchor_x: right
add_widget8_opacity_50:
    widget8:
      widget_settings:
        opacity: .5
        text: 50% OPACITY
        font_size: 50
    widget1:
      action: add
event_a:
  widget1:
    action: update
    widget_settings:
      text: A
      color: red
event_s:
  widget1:
    action: update
    widget_settings:
      text: S
      color: lime
event_d:
  widget1:
    action: update
    widget_settings:
      text: D
      color: blue
widget_4up:
  box14:
    widget_settings:
```

(continues on next page)

(continued from previous page)

```
      x: 25
      expire: 6s
  box13:
    widget_settings:
      x: 105
      expire: 6s
  box12:
    widget_settings:
      x: 185
      expire: 6s
  box11:
    widget_settings:
      x: 265
      expire: 6s
widget_4up_red:
  box14:
    widget_settings:
      color: red
  box13:
    widget_settings:
      color: red
  box12:
    widget_settings:
      color: red
  box11:
    widget_settings:
      color: red
widget_to_parent:
  box11:
    widget_settings:
      z: -1
  box12:
    widget_settings:
      z: 2
      color: red
      y: middle+2
show_christmas_slide_full:
  widget2:
    widget_settings:
      expire: 5s
    slide: slide1
    key: xmas_intro_keyname
remove_christmas_full:
  widget2:
    action: remove
    key: xmas_intro_keyname
show_widget9:
  widget9:
    key: wigdet9_wp_key
show_widget10:
  widget10:
    action: add
show_bezier_widget: widget_bezier
show_ellipse_widget: widget_ellipse
```

(continues on next page)

(continued from previous page)

```
  show_quad_widget: widget_quad
  show_rectangle_widget: widget_rectangle
  show_line_widget: widget_line
  show_triangle_widget: widget_triangle
  show_points_widget: widget_points
  show_text_widget: widget_text
  show_custom_events1_widget: widget_custom_events1
  show_custom_events2_widget: widget_custom_events2
  remove_custom_events1_widget:
    widget_custom_events1:
      action: remove
  remove_custom_events2_widget:
    widget_custom_events2:
      action: remove

slide_player:
  show_slide_1:
    slide_1:
    - type: text
      text: WIDGET WITH KEY
      key: widget1
      color: red
      y: 33%
    - type: text
      text: WIDGET NO KEY
      color: red
      y: 66%
  show_slide_1_with_expire:
    slide_1:
    - type: text
      text: WIDGET EXPIRE 1s
      expire: 1s
      color: red
      y: 33%
    - type: text
      text: WIDGET NO EXPIRE
      color: red
      y: 66%
  show_slide_2:
    slide_2:
    - type: text
      text: TEST UPDATING EXISTING WIDGET SETTINGS
      y: bottom
      anchor_y: bottom
  show_slide_3:
    slide_3:
      widgets:
      - type: text
        text: WIDGET REPLACEMENT
        y: 25%
  show_slide_with_widgets:
    slide_1:
      - type: text
        text: widget4.1
```

(continues on next page)

(continued from previous page)

```
        y: 300
        z: 1
        color: ff0000
        font_size: 100
      - type: text
        text: widget4.2
        z: 1000
        y: 250
        color: ffff00
        font_size: 100
      - type: text
        text: widget4.3
        y: 200
        color: 00ff00
        font_size: 100
      - type: text
        text: widget4.4
        z: 1
        y: 150
        color: 00ffff
        font_size: 100
      - type: text
        text: widget4.5
        z: 1000
        y: 100
        color: 0000ff
        font_size: 100
      - type: text
        text: widget4.6
        color: ff00ff
        font_size: 100
        y: 50
      - type: text
        text: widget4.7
        y: 0
        color: 888888
        font_size: 100
  show_slide_with_lots_of_widgets: slide_with_lots_of_widgets
  show_new_slide:
    new_slide2:
      widgets:
      - type: text
        text: NEW SLIDE
        y: 0
        anchor_y: bottom
        events_when_added: text_on_new_slide2_added
        events_when_removed: text_on_new_slide2_removed
  remove_new_slide:
    new_slide2:
      action: remove

slides:
    slide_with_lots_of_widgets:
      - type: text
```

(continues on next page)

(continued from previous page)

```
        text: widget4.1
        y: 300
        z: 1
        color: ff0000
        font_size: 100
      - type: text
        text: widget4.2
        z: 1000
        y: 250
        color: ffff00
        font_size: 100
      - type: text
        text: widget4.3
        y: 200
        color: 00ff00
        font_size: 100
      - type: text
        text: widget4.4
        z: 1
        y: 150
        color: 00ffff
        font_size: 100
      - type: text
        text: widget4.5
        z: 1000
        y: 100
        color: 0000ff
        font_size: 100
      - type: text
        text: widget4.6
        color: ff00ff
        font_size: 100
        y: 50
      - type: text
        text: widget4.7
        y: 0
        color: 888888
        font_size: 100
```

Listing 409: your_machine_folder/widgets/config/test_widgets_with_named_colors.yaml

```
#config_version=5

displays:
  default:
    width: 800
    height: 600

named_colors:
   tt_yellow: [255,220, 0]

widgets:
   base_score_widget:
      - type: text
```

(continues on next page)

(continued from previous page)

```
        text: HELLO
        color: tt_yellow


widget_player:
  add_widget1_to_current: base_score_widget
```

Mode config examples

Here are some example mode config files that go along with the machine-wide config above.

Listing 410: your_machine_folder/widgets/modes/mode1/config/mode1.yaml

```
#config_version=5

mode:
  priority: 500

widget_player:
  mode1_add_widgets: widget2
  mode1_add_widget6:
    widget6:
      widget_settings:
        z: -1
  mode1_add_widget_with_key:
    widget2:
      key: newton_crosby
  mode1_update_widget2:
    widget2:
      action: update
      key: newton_crosby
      widget_settings:
        text: UPDATED TEXT
  show_widget_with_placeholder: widget_with_placeholder

widgets:
  widget_with_placeholder:
    - type: text
      text: Placeholder widget
    - widget: widget_placeholder_(value)

slide_player:
  show_slide_with_named_widget: slide_with_named_widget

slides:
  slide_with_named_widget:
    - type: text
      text: One Use Widget
    - widget: widget_reusable
```

CHAPTER 23

Example Machine Projects you can learn from

The mpf-examples project

Contains several examples of MPF configs you can run and learn from, including:

- *Demo Man*
- *MC Demo*
- *Cookbook recipes*
- *Tutorial config files*

Full details about the mpf-examples project and how to download it are *here*.

State Fair Pinball

An upcoming machine we're working on. Details: http://statefairpinball.com. GitHub repo: https://github.com/missionpinball/state_fair

Brooks 'n Dunn

One of the projects we took on in 2015 was to rewire and build and MPF config for Gottlieb's Brooks 'n Dunn. (BnD). BnD was the machine that Gottlieb was working on when they shut down in 1996.

This config is probably the most complete of any MPF project that's publicly available. However it contains lots of licensed assets (music, videos, images, etc.) that are not in the public repo. This means you won't be able to actually run it, but you can look through the configs (which are well commented) to see how we do things.

The BnD repo is at https://github.com/gabeknuth/bnd

Mass Effect 2

An extensive project to build a complete MPF game from scratch and play on a re-skinned Game of Thrones cabinet (Spike platform), inspired by the video game Mass Effect 2. With the exception of audio tracks extracted from the Mass Effect 2 data files, all of the game code is available to clone from the repo and run. MPF monitor is supported so you can simulate gameplay without the Spike GoT hardware.

All of the project code is at https://github.com/avanwinkle/masseffect2

How to download the mpf-examples bundle

We maintain a GitHub repo called mpf-examples which contains a few different example MPF configs and some templates you can use.

The mpf-examples repo doesn't have an installer, rather, you just download it and unzip it and start using the stuff it contains.

Each software repo in GitHub has several "branches". (Think of branches kind of like versions.) The mpf-examples repo has multiple branches that each match a specific version of MPF. For example, the 0.21 branch of the mpf-examples repo is for MPF 0.21, the 0.30 branch is for MPF 0.30, etc.

Here are the direct links (to ZIP files) for the various branches of mpf-examples that you can download based on your version of MPF:

- **|branch|**0.56

If you are looking for another version please switch to the documentation of that MPF version.

Unzip the file to any location you want, and then browse the files to see what's there, or open a console window to launch MPF and/or MPF MC in each folder.

`demo_man`

> Williams Demolition Man. This config is pretty basic, but you can play complete games and it has some simple shots, scoring, and modes. It also contains custom code to run the Cryro Claw. See details which explain how to "play" this game on your computer *here*.

`mc_demo`

> A machine config that demonstrates several capabilities of the MPF Media Controller (MPF-MC). Details *here*.

`tutorial` (and several `tutorial_step_XX` folders)

> Contains the config files used in the *MPF Tutorial*.

`wpc_template`

> A template config you can use for WPC machines (with either a P-ROC or FAST WPC controller).

How to run "Demo Man", an MPF example game

One of the development machines we have for MPF is a 1994 Williams *Demolition Man*, and we have a simple MPF configuration built for it that you can run to see MPF in action.

Even if you don't have a physical *Demolition Man* machine (which we assume you don't), you can run our "Demo Man" config using MPF's *smart virtual* platform.

1. Download the MPF examples bundle

Instructions *here*.

2. Run *Demo Man*, a sample game that comes with MPF

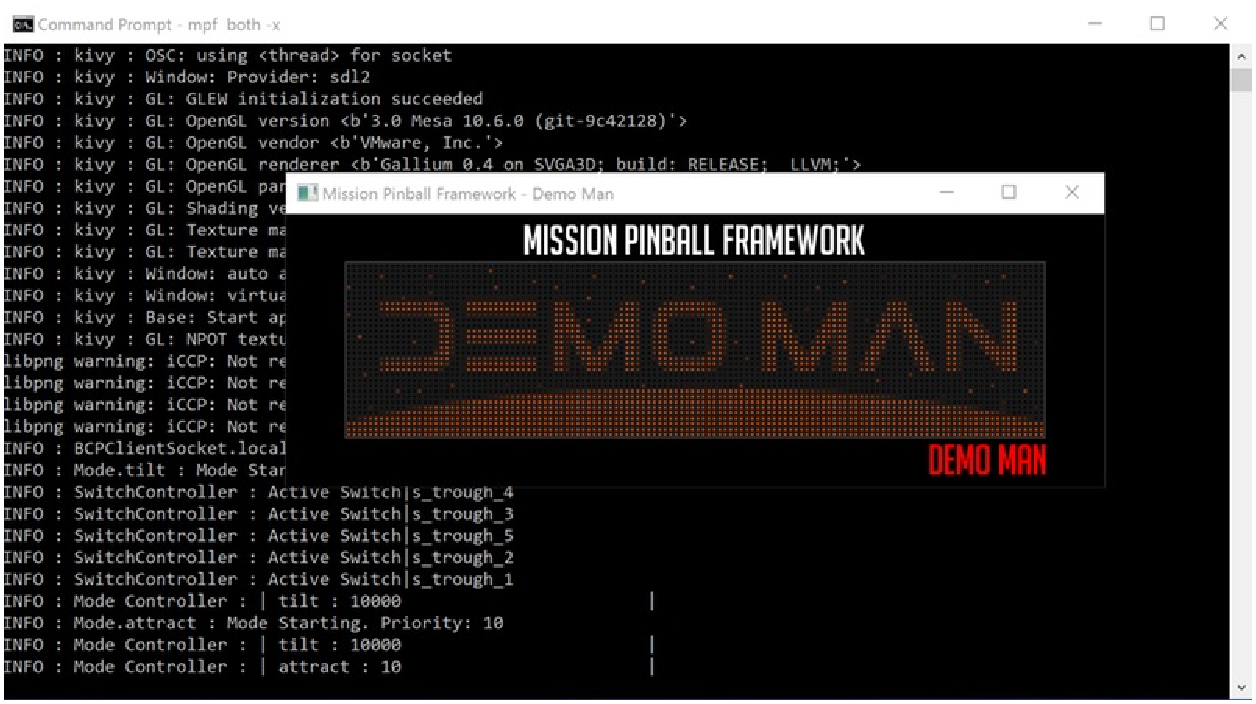
Open a command prompt (like you did when you installed MPF) and switch to the folder where you unzipped the mpf-examples ZIP file, then change to the `demo_man` folder and run:

```
mpf both -X
```

(Note that's an uppercase "X")

The `mpf both` command launches both the MPF game engine and media controller at the same time, the `-X` command line option tells MPF to use the "Smart Virtual" platform (instead of the P-ROC platform that the Demo Man files are configured for) since you most likely don't have a Demolition Man machine connected to your computer right now.

You should see a bunch of stuff scroll by and a pop up window which shows the Demo Man DMD, like this:



If you don't see the DMD window pop up, make sure it isn't hiding behind another window.

3. "Play" your first game

Since you don't have physical hardware attached, you can use the keyboard to simulate machine switch changes.

The *Demo Man* configuration files have the "S" key mapped to start, so if you click in the graphical window with the DMD in it (to give it focus) and push the `S` key, then you should see the DMD attract mode stop and it change to a score screen showing a score of *00* and *BALL 1 FREE PLAY*:

If your speakers are on you should also hear a music loop playing. (Depending on your system, you might not hear the music when the DMD window doesn't have focus.)

At this point you can "play" the game via your keyboard. Hit the `L` key to launch the ball into play. You should hear the music loop change to the main background music.

You can hit the `X` key to simulate the left slingshot hit which should play a sound effect on top of the music as well as show a score. You can hit the `1` key to simulate the ball draining and entering the trough. Then you can hit the `L` key again to launch the ball into play again. You can also press the `S` key additional times during Ball 1 to add additional players.

When you play through a complete game (3 balls per player), the machine should go back into attract mode (or possibly the high score entry mode).

You can quit the game by making sure the *Demo Man* popup window is in focus and hitting the *Esc* key.

To summarize the instructions for "playing" a game from the paragraphs above:

1. Launch both the MPF core engine and the media controller and make sure you see the the popup graphical window with the DMD in it.
2. Click the mouse into the DMD window so that it has "focus"
3. Press the `S` key to start a game. You should hear the music loop start.
4. Press the `L` key to launch a ball into play. You should her the music switch to the main background theme for the game.
5. Press the `X` key a few times to simulate hitting the left slingshot. You should see the score change each time you do this.
6. Press the `1` key to drain the ball.
7. Repeat Steps 4-6 until you finish your game or get bored.
8. If you get a high score, the `Z` and `/` keys are mapped to the left and right flipper buttons to highlight a letter, and the `S` key (start) selects it.
9. Press the `Esc` key to exit

What if it did not work?

In the following we list some common problems and solutions. If you got another problem please ask in our MPF User Forum.

Keyboard does not work

If your keyboard does not work first make sure that the MC window has focus. Afterward, please check if numlock is enabled. This seems to be common issue on Windows 10. Disable numlock in this case.

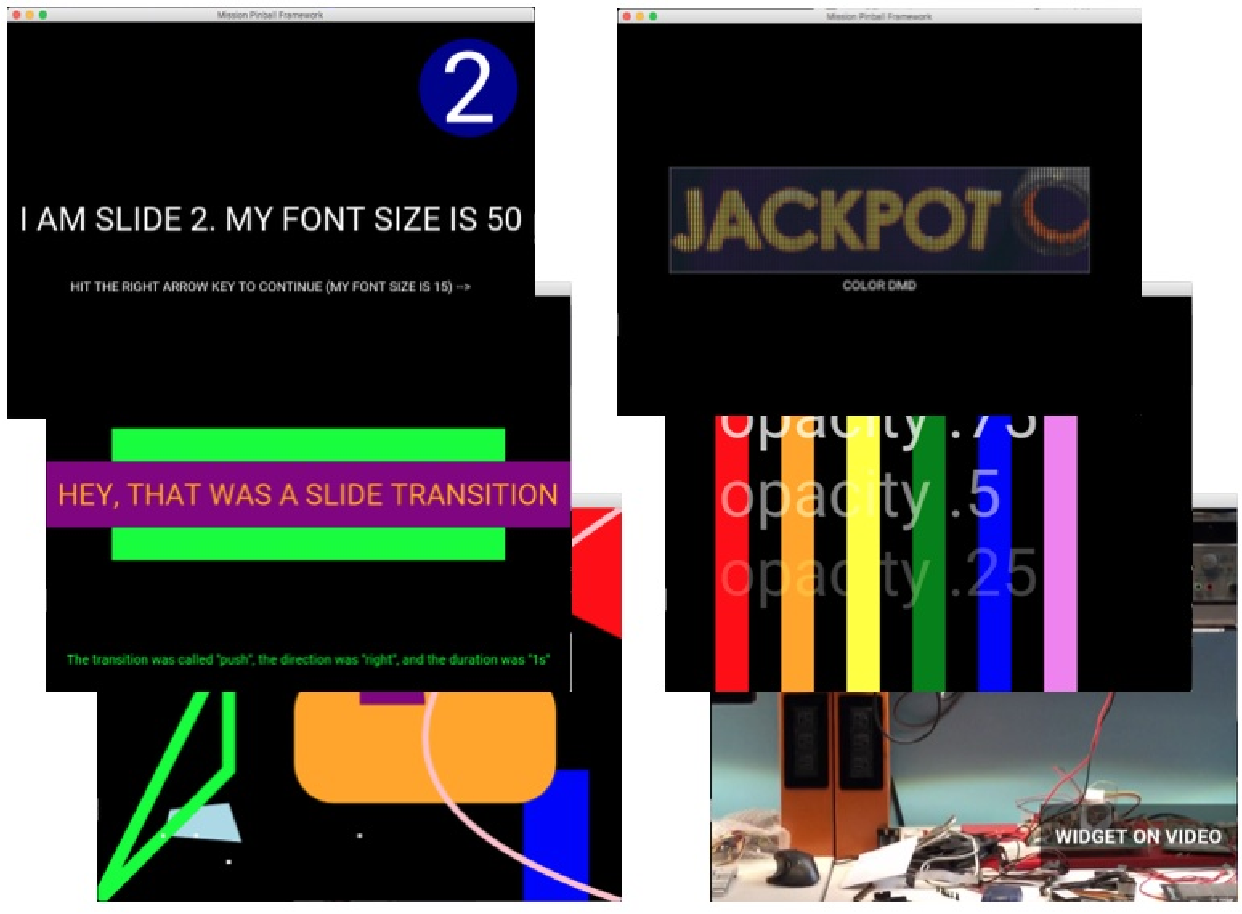
Game does not start errors

Version mismatch - Make sure you are using the exact version of demo_man for your MPF, MPF-MC and MPF monitor (optional). For instance use `0.30.x` if you are on MPF `0.30.2`. MPF-MC and (if installed) MPF monitor need to have the same major version (`0.30` in this case). You can check installed versions using `pip3 list`.

FT_ListDevices(2) or WARNING : P-Roc : Failed to instantiate pinproc.PinPROC(3) error - By default the Demo Man example expects to be connected to P-Roc hardware. If you see this you probably forgot to add the `-X` switch to your `mpf both` command which tells MPF to override the hardware setting to the Smart Virtual platform. This sometimes happens when trying to debug another error and you follow the guide about verbose logging using the `-v -V` switches.

How to run the "MC Demo" example

The *MPF Examples* GitHub repository includes a machine configuration called "MC Demo" which is a demo of many different features of the MPF media controller. Here are a few random screen shots of it:



You can run it and use the arrow keys on your computer to step through different slides, and then you can look at the source config file to see how it all works.

It's designed to both show you what's possible and to show you how to do different things with the MC.

1. Download the MPF examples bundle

Instructions *here*.

2. Run it

Open a command prompt or terminal window, and change to the "mc_demo" folder in the "mpf-examples" package you downloaded. Then run:

```
mpf both
```

When you run the demo, use the left and right arrow keys to step through the different slides.

3. Check out the config (with notes!)

You can browse the complete config in the `mc_demo/config/config.yaml` file. Or check it out online here.

CHAPTER 24

The MPF Cookbook

The MPF cookbook contains recipes (how to guides) which show you how you would implement complex features in your pinball machines using MPF.

Here are the recipes that are done:

- *The Addams Family: Mansion Awards*
- *Attack From Mars: Super Jets*
- *Indiana Jones: Rollover Lanes (with Lane Change)*
- *Batman 66: Gadgets Targets*
- *Modifying the Game mode: Dual launch devices*
- *Skillshots with Lane Change*
- *Skillshots with Auto-Rotate*
- *Sequential Drop Bank Targets*

If you've ever played a game and wondered, "How would I do that?" then let us know and we'll write a recipe for it! And here's what's on our to-do list:

- *Attack from Mars:* 5-Way Combo
- *Red & Ted's RoadShow:* Bulldozer hits to ball lock & multiball
- *Red & Ted's RoadShow:* City modes
- *Centigrade 37:* Flip-flopping groups of lit targets
- *Judge Dredd* DeadWorld ball lock and multiball
- *Demolition Man* Crane elevator & unloader

Recipe: The Addams Family Mansion Awards

This guide shows you how to build an MPF config for *The Addams Family's* Mansion Awards and Tour the Mansion feature. The idea is you can use this as a guide to implement a similar feature in your machine.

Note: This recipe requires MPF 0.33 or newer.

This guide uses the following concepts in MPF that you should be familiar with:

- *Modes*
- *Achievements*
- *Achievement Groups*
- *Shows*

This guide will also show you how to do a few tricky things, including:

- From a group of 12 achievements, ensure that the randomly selected one when the game starts is 1 of 2, not random from all 12.
- Have two shots that light the achievements, but one of the shots lights the achievements indefinitely and the other only lights them for 3 seconds.

You can find the complete runnable machine config for this recipe in the `cookbook/TAF_mansion_awards` folder of the mpf-examples repository on GitHub.

What are the Mansion Awards & Tour the Mansion?

In The Addams Family, the Mansion Awards are the name for the 12 "goals" which each have a light in the mansion on the playfield just above the flippers.

Tour the Mansion is a wizard mode (associated with the question mark insert at the top of the mansion) that can be started after all 12 mansion awards have been collected.

Here are the specific rules we need to implement:

Mansion Awards

- Lights for incomplete awards are off.
- Complete awards are on solid.
- The currently selected award's light is flashing.
- Hitting any pop bumper will change the currently selected award to another random from the awards that are not yet complete.
- When the game starts, either "Hit Cousin It" or "Mamushka" are selected.
- The selected award is awarded / collected when the electric chair is lit (yellow and red lights on the chair toy) and either the electric chair or swamp shot is hit. (The swamp is technically an operator setting, but we'll use it since that's what the default it.)
- Some of the awards start modes, and others are instant awards with a short show. Collecting an

award immediately turns its light on solid and selects another random uncollected award.

- If 3 Mil is awarded, 6 Mil is spotted (automatically set to complete) as well, and vice-versa. (This differs in the Gold Edition of the game, and is also an operator setting, but we're just going to hard code this behavior for this recipe.)
- The electric chair is lit for 3 seconds after the right inlane is hit.
- The electric chair is lit indefinitely after either ramp is hit.
- The electric chair is lit at the beginning of each ball
- For awards that start modes, the chair can be relit and another award awarded even while the prior award's mode is running.
- Accumulating 15, 25, 35, 45, 55, 65, 75, 85, 95 bear kicks (center ramp) collects the currently selected award (except Tour the Mansion), even if the chair is not lit.
- Each award collected adds 500k to the bonus.

Tour the Mansion

- Once all 12 Mansion Awards have been collected, the Tour the Mansion light (the question mark at the top of the mansion) is selected.
- The electric chair must be lit in the same way as before, and then the shot must be made to the electric chair or the swamp as before.
- This starts the Tour the Mansion mode
- When Tour the Mansion completes, all the mansion awards are reset and a new random one is selected.
- If Tour the Mansion ends before the ball ends, no mansion award can be awarded until the next ball.

Step 1. The machine-wide prerequisites

Before we dig into how to handle the mansion itself, we need to create a machine-wide config that has all the devices we'll need, including the lights for the mansion, switches for the shots we need, the ramps, the right inlane, and the switches, coils, and ball devices we need to glue it all together.

Here's what our machine config looks like. (Note that this is complete in terms of what we need to make this recipe work, but if you have a real Addams Family then you'll probably have a lot more than this in your machine config file.)

```
#config_version=5
modes:
  - mansion_awards
  - chair_lit
  - chair_lit_3s
switches:
  start:
    number: S13
    tags: start
  drain:
    number:
  trough1:
```

(continues on next page)

(continued from previous page)

```
    number: S15
  trough2:
    number: S16
  trough3:
    number: S17
  plunger_lane:
    number: S27
  swamp_kickout:
    number: S74
  electric_chair:
    number: S43
  left_ramp:
    number: S66
  center_ramp:
    number: S65
  right_inlane:
    number: S25
  upper_left_jet:
    number: S31
    tags: jet
  upper_right_jet:
    number: S32
    tags: jet
  center_left_jet:
    number: S33
    tags: jet
  center_right_jet:
    number: S34
    tags: jet
  lower_jet:
    number: S35
    tags: jet
virtual_platform_start_active_switches: trough1, trough2, trough3
coils:
  drain:
    number: "05"
  trough:
    number: "04"
  swamp_kickout:
    number: "08"
  electric_chair:
    number: "01"
lights:
  9_mil:
    number: L66
    subtype: matrix
  6_mil:
    number: L54
    subtype: matrix
  3_mil:
    number: L68
    subtype: matrix
  thing:
    number: L51
```

(continues on next page)

(continued from previous page)

```
    subtype: matrix
  quick_multiball:
    number: L55
    subtype: matrix
  graveyard_at_max:
    number: L67
    subtype: matrix
  raise_the_dead:
    number: L52
    subtype: matrix
  festers_tunnel_hunt:
    number: L56
    subtype: matrix
  lite_extra_ball:
    number: L53
    subtype: matrix
  seance:
    number: L57
    subtype: matrix
  hit_cousin_it:
    number: L58
    subtype: matrix
  mamushka:
    number: L45
    subtype: matrix
  mansion_question:
    number: L65
    subtype: matrix
  electric_chair_yellow:
    number: L64
    subtype: matrix
  electric_chair_red:
    number: L47
    subtype: matrix
ball_devices:
  drain:
    ball_switches: drain
    eject_coil: drain
    eject_targets: trough
    tags: drain
  trough:
    ball_switches: trough1, trough2, trough3
    eject_coil: trough
    eject_targets: plunger_lane
    tags: trough, home
  plunger_lane:
    ball_switches: plunger_lane
    mechanical_eject: true
    eject_timeouts: 3s
    tags: home
  electric_chair:
    ball_switches: electric_chair
    eject_coil: electric_chair
  swamp_kickout:
```

(continues on next page)

(continued from previous page)

```
    ball_switches: swamp_kickout
    eject_coil: swamp_kickout
playfields:
  playfield:
    default_source_device: plunger_lane
    tags: default
##! mode: mansion_awards
# mode will be defined below
##! mode: chair_lit
# mode will be defined below
##! mode: chair_lit_3s
# mode will be defined below
```

Step 2. Add the achievements

Each mansion award will be an achievement. We decided to create a separate mode called "mansion_awards" just so we can keep everything separate. (This isn't required, it's just to help us keep it clear in our minds, and it's ok to have lots and lots of modes in MPF.)

We'll configure this mode to start on the *ball_starting* event so it's always running when a ball is in play. We won't configure a stop event which means this mode will automatically stop when the ball ends.

Next we add an `achievements:` section and then subsections for our 12 mansion achievements.

You'll notice that most of them are almost identical. For example, here's the entry for Thing Multiball:

```
##! mode: mansion_awards
achievements:
  thing_multiball:
    show_tokens:
      lights: thing
    show_when_selected: flash
    show_when_completed: on
    events_when_started: award_thing_multiball    # starts thing_multiball mode
    enable_events: initialize_mansion, reset_mansion
    complete_events: award_thing_multiball
    reset_events: reset_mansion
```

Stepping through how we're using each setting:

show_tokens: link this achievement to it's light on the playfield.

show_when_selected: flash Plays the show called "flash" when this achievement is selected. Note that the default "flash" show is 1 sec on / 1 sec off. While you can play it faster, the original Addams Family flashed the lights more like .75s on / .25 off, so you'd probably want to create a custom version of the "flash" show for TAF that flashed them more like the original version.

show_when_completed: on Plays the show called "on" when this achievement is complete

events_when_started: award_thing_multiball Posts an event called *award_thing_multiball* when this achievement is started. We'll use this as the start event for the Thing Multiball mode.

enable_events: initialize_mansion, reset_mansion Enables this achievement when either of the events *initialize_mansion* or *reset_mansion* is posted. Prior to that, this achievement will be disabled.

`complete_events: award_thing_multiball` Watches for the event *award_thing_multiball*, and when it sees it, it marks this achievement as complete. Notice this is the same event that this achievement posts when it starts. In other words, we've configured it so the achievement is complete as soon as it starts! This is by design, because the rules state that once an achievement is awarded, the chair can be relit immediately, and it's possible to receive the next award even while the mode from the prior award is still running.

`reset_events: reset_mansion` Watches for an event called *reset_mansion* that will reset this achievement back to its initial (disabled) state.

This achievements configuration takes care of the following rules:

- Lights for incomplete awards are off.
- Complete awards are on solid.
- The currently selected award's light is flashing.

Step 3. Create an achievement group

Next we need to create an achievement group called "mansion_awards" which will group the 12 mansion achievements together. That will look like this:

```
##! mode: mansion_awards
achievement_groups:
  mansion_awards:
    achievements:
      - 9_mil
      - 6_mil
      - 3_mil
      - thing_multiball
      - quick_multiball
      - graveyard_at_max
      - raise_the_dead
      - festers_tunnel_hunt
      - lite_extra_ball
      - seance
      - hit_cousin_it
      - mamushka
    show_tokens:
      lights: electric_chair_yellow, electric_chair_red
    auto_select: true
    events_when_all_completed: select_tour_mansion
    enable_while_no_achievement_started: false
    show_when_enabled: on
    select_random_achievement_events: sw_jet
    allow_selection_change_while_disabled: true
    disable_while_achievement_started: false
    start_selected_events: balldevice_electric_chair_ball_enter, balldevice_swamp_kickout_ball_enter,
↪award_mansion_from_bear
    enable_events: light_chair
    disable_events: unlight_chair
```

Let's look at each of these settings:

`achievements:` This is just the list of the 12 achievements that make up this group.

`show_tokens:` These are the show tokens for the group itself. In this case they're the two lights on the electric chair, since those lights turn on and off to indicate whether the chair or swamp can be shot to award the currently selected item.

`auto_select: yes` This is used to make sure that one achievement is selected at all times. If the currently selected achievement is completed, the achievement group will notice that there is no currently selected achievement and it will pick one from random from the remaining achievements (those that are "enabled").

`events_when_all_completed: select_tour_mansion` Posts an event called *select_tour_mansion* once all 12 achievements in this group in complete. We'll use this later to light the "tour mansion" award.

`enable_while_no_achievement_started: no` In our case, we do not want to automatically enable the achievement group when no achievement is started, because the rules for Addams Family say that the player has to shoot the center ramp or right inlane to light the chair (which is enabling this achievement group).

`show_when_enabled: on` This plays the show called "on" when the achievement group is in the enabled state. This will have the effect of turning on the red and yellow chair lights (from the `show_tokens:` section) when the achievement group is enabled and the selected item can be awarded.

`select_random_achievement_events: sw_jet` In Addams Family, each pop bumper hit changes the currently selected mansion award. To make this happen, we added a tag called "jet" to the five pop bumper switches. (That will post an event called *sw_jet* any time one of these switches is hit. Then we add that event name here which will cause this achievement group to change the currently selected award.

`allow_selection_change_while_disabled: yes` The pop bumper hits to change the current selection happens regardless of whether the group is enabled (e.g. the chair is lit) or not, so we use this setting to allow that selection change to happen at any time.

`start_selected_events: balldevice_electric_chair_ball_enter, balldevice_swamp_kickout_ball_enter, award_mans` A shot to either the electric chair or the swamp kickout will award the selected achievement.

`enable_events: light_chair` When an event called *light_chair* is posted, this achievement group will be enabled (which will turn on the chair lights and allow the selected achievement to be started via the `start_selected_events:`.

`disable_events: unlight_chair` When an event called *light_chair* is posted, this achievement group will be disabled. The chair lights will turn off, and the `start_selected_events:` will not cause the current selected achievement to start.

This step takes care of:

- Hitting any pop bumper will change the currently selected award to another random from the awards that are not yet complete.
- The selected award is awarded / collected when the electric chair is lit (yellow and red lights on the chair toy) and either the electric chair or swamp shot is hit.

Step 4. Light the electric chair

Now that we have the basic achievements and achievement group structure laid out, let's focus on getting the chair lit. We'll look at the following four rules:

- The electric chair is lit for 3 seconds after the right inlane is hit.
- The electric chair is lit indefinitely after either ramp is hit.
- The electric chair is lit at the beginning of each ball
- For awards that start modes, the chair can be relit and another award awarded even while the prior award's mode is running.

At first this seems pretty straightforward. If the center ramp is shot, post an event to enable the achievement group. If the right inlane is hit, post an event to enable the achievement group and also set a timer that will disable it 3 seconds later. The problem with this is that if the chair was previously lit from the ramp when the inlane is hit, we don't want the inlane timer to disable the chair after 3 seconds.

There are several ways in MPF to achieve this. In our case, we're going to use modes. (We really like *using modes for game logic*.)

The two modes we're going to create are:

- chair_lit_3s
- chair_lit

The chair_lit_3s mode

Let's look at the config for the "chair_lit_3s" mode:

```
##! mode: chair_lit_3s
#config_version=5
mode:
  priority: 101
  start_events: right_inlane_active
  stop_events: unlight_chair balldevice_electric_chair_ball_enter balldevice_swamp_kickout_ball_enter␣
↪cancel_chair_timer
event_player:
  mode_chair_lit_3s_started: light_chair
  timer_unlight_chair_complete: unlight_chair
timers:
  unlight_chair:
    end_value: 3
    start_running: true
```

Notice that this mode started when the *right_inlane_active* switch is hit, which means it starts when the right inlane is hit. Pretty simple.

When it comes to stop events, we have four of them. First is *unlight_chair*. This mode has a timer (for 3 seconds) which starts when the mode starts, so when that completes, it posts *timer_unlight_chair_complete* which the event player uses to post *unlight_chair* which will stop the mode. (The *unlight_chair* event is also used by the mansion achievement group to disable itself.

There are also stop events for *balldevice_electric_chair_ball_enter* and *balldevice_swamp_kickout_ball_enter* which stop this mode if either of those shots are hit. Notice those are also `start_selected_events:` for the achievement group, so hitting either one of those will start the selected achievement (if the group is enabled) and also stop this mode.

You may be wondering why we have both of those ball enter events listed here? Why not just use an "events_when_started" setting in the achievement group to stop this mode? The reason is for this rule here:

- Accumulating 15, 25, 35, 45, 55, 65, 75, 85, 95 bear kicks (center ramp) collects the currently selected award (except Tour the Mansion), even if the chair is not lit.

This shot will "start" an award, but if the chair is lit, we do not want it to unlight, so that's why we need to stop the chair_lit_3s mode based on the actual chair or swamp being hit, not just any time the selected award is started.

Finally, notice there's also an event called *cancel_chair_timer* which will stop this mode. We'll talk about that in a bit.

The only other thing to discuss in this mode is the `event_player:`. We talked about the timer being used to post the *unlight_chair* event. But notice there's also an entry `mode_chair_lit_3s_started: light_chair` which posts the *light_chair* event when the mode starts. (This event is listed in the achievement group as the event which enables it.) These settings, in combination, mean that when the chair_lit_3s mode is running, the mansion achievement group will be enabled (e.g. the chair is lit).

The chair_lit mode

The second mode we're going to create will be like the chair_lit_3s mode, except instead of having a timer that stops the mode after 3 seconds, this mode will stay active until the chair or swamp is hit. (Well, or until the ball ends, as by default, all modes end when the ball ends automatically.)

Here's the config for this mode:

```
##! mode: chair_lit
#config_version=5
mode:
  priority: 102
  start_events: center_ramp_active, ball_starting
  stop_events: balldevice_electric_chair_ball_enter balldevice_swamp_kickout_ball_enter
event_player:
  mode_chair_lit_stopping: unlight_chair
  mode_chair_lit_started: light_chair, cancel_chair_timer
  mode_chair_lit_3s_started: cancel_chair_timer
counters:
  initialize_mansion:
    count_events: mode_chair_lit_started
    events_when_complete: initialize_mansion
    count_complete_value: 1
    persist_state: true
```

The `start_events:` are pretty straightforward. We start the mode when the center ramp is hit, and also on *ball_starting* since the Addams Family rules state that the chair is lit at the beginning of every ball.

This mode has an event_player to help with the logic. When this mode stops, we also post the *unlight_chair* event which is one of the disable events for the mansion achievement group. We also post the *light_chair* event when the mode starts to enable the group.

The final two event player settings help us with the interaction between this mode and the 3 second timed version. We have *cancel_chair_timer* as an event that's fired when this mode starts too. Notice that that event is one of the `stop_events` for the other mode. The reason for this is that if the ball hits the right inline and the chair is lit for 3 seconds, and then the ball hits the center ramp within those 3 seconds, we need to make sure the chair stays lit indefinitely, meaning we need to stop the 3s mode so it doesn't shut the chair off. So that's what this event is doing.

Similarly if the player had previously hit the center ramp (which starts this mode to light the chair), and then the player hits the right inline, we also need to kill that 3s mode to make sure it doesn't turn off the chair, so we do that with the event player setting `mode_chair_lit_3s_started: cancel_chair_timer`. Basically this setting means that if this mode sees the 3s mode, it shuts it down. :) And obviously this shut down only happens if this mode is running.

What about that logic block? Let's discuss that in the next step. . .

Step 5. Select the proper award at game start

One of the twists of the Addams Family mansion awards is that when the game first starts, it always starts with either "Hit Cousin It" or "Mamuska" selected. So we have to figure out a way to randomly pick from one of those two (instead of all 12) at the start of the game, but then every random choice after that has to be from all 12 (well, of the ones that have not yet been awarded out of all 12.

We'll tackle this in two parts.

First, take a look at the Hit Cousin It and Mamuska achievements:

```
##! mode: mansion_awards
achievements:
  hit_cousin_it:
    show_tokens:
      lights: hit_cousin_it
    show_when_selected: flash
    show_when_completed: on
    events_when_started: award_hit_cousin_it  # starts hit_cousin_it mode
    complete_events: award_hit_cousin_it
    reset_events: reset_mansion
  mamushka:
    show_tokens:
      lights: mamushka
    show_when_selected: flash
    show_when_completed: on
    events_when_started: award_mamushka   # starts mamushka mode
    complete_events: award_mamushka
    reset_events: reset_mansion
```

Notice that they're slightly different than the other 10 mansion awards in that they do NOT have enable events.

The reason for this is that devices in MPF that have enable_events in their configurations are NOT automatically enabled when they're created. (This is because MPF thinks, "Hey, you have enable events, so you have some way to enable them, so you can enable them whenever you want." But if there are no enable events, like these two, then MPF will enable them immediately.)

This means that when this mode first starts and these 12 mansion achievements are created, the `hit_cousin_it` and `mamuska` achievements are enabled immediately (since they don't have enable events), and the other 10 mansion awards are disabled (since they do have enable events). Since the achievement group is configured for `auto_select: yes`, it will automatically (and immediately) pick one of the enabled achievements which will change into the selected state (and start it's select show, etc.). This means that the initial selection will always be one of those two.

However, once the initial selection is made, we need a way to enable the remaining 10 mansion awards. For this we'll use a counter logic block:

```
##! mode: chair_lit
# This is in the chair_lit mode config, NOT machine-wide config
counters:
  initialize_mansion:
    count_events: mode_chair_lit_started
    events_when_complete: initialize_mansion
    count_complete_value: 1
    persist_state: true
```

This is a simple counter that "counts" the *mode_chair_lit_started* event (which is posted by this mode once it's fully started and done initializing). The count complete value is one, meaning that once it sees this event once, it's done. We tell it to persist its state so that it remembers where it was from ball-to-ball (meaning it will only run once ever in the game) and when it's done (which is after it sees that event once) it will post the event *initialize_mansion*.

(Remember that logic block states are stored on a per-player basis, so everything we say happens "once" here is really "once per player".)

Note also that in the 10 "other" mansion achievements, we have *initialize_mansion* listed as one of their enable events. This means that when this counter completes its count (of 1) that it will post that event which will enable the other 10 achievements.

At this point you'll have 1 achievement selected (which will be either Hit Cousin It or Mamushka), and you'll have the other 11 in the "enabled" state.

Hitting a pop bumper will pick a new random selected achievement.

Step 6. Kick off the award

Next up we have an easy thing: Starting the modes and/or kicking off the shows for each mansion award.

In this case, note that our 12 mansion achievements each have an `events_when_started:` setting with a unique event name, like *award_seance* or *award_lite_extra_ball*. So just use that event to either start a mode or to play a show. Simple!

- Some of the awards start modes, and others are instant awards with a short show. Collecting an award immediately turns its light on solid and selects another random uncollected award.

Step 7. Collect the selected award via the bear kick

Todo: Need to explain this fully

- Accumulating 15, 25, 35, 45, 55, 65, 75, 85, 95 bear kicks (center ramp) collects the currently selected award (except Tour the Mansion), even if the chair is not lit.

Step 8. Setup the 3 Mil / 6 Mil linking

- If 3 Mil is awarded, 6 Mil is spotted (automatically set to complete) as well, and vice-versa.

This is pretty simple. Just add the events posted when one achievement is started to the complete events for the other. Here are the examples:

```
##! mode: mansion_awards
achievements:
  6_mil:
    show_tokens:
      lights: 6_mil
    show_when_selected: flash
    show_when_completed: on
    events_when_started: award_6_mil    # instant points award & plays shows, also spots 3 mil
    enable_events: initialize_mansion, reset_mansion
    complete_events: award_6_mil, award_3_mil
    reset_events: reset_mansion
  3_mil:
    show_tokens:
      lights: 3_mil
    show_when_selected: flash
    show_when_completed: on
    events_when_started: award_3_mil    # instant points award & plays shows, also spots 6 mil
    enable_events: initialize_mansion, reset_mansion
    complete_events: award_3_mil, award_6_mil
    reset_events: reset_mansion
```

Notice that the 6_mil's `complete_events:` includes *award_3_mil* and vice-versa.

Step 8. Add 500k to the bonus for each award collected

Todo: Need to explain this fully

- Each award collected adds 500k to the bonus.

Step 9. Move on to Tour the Mansion after all 12 awards have been completed

Todo: Need to explain this fully

- Once all 12 Mansion Awards have been collected, the Tour the Mansion light (the question mark at the top of the mansion) is selected.
- The electric chair must be lit in the same way as before, and then the shot must be made to the electric chair or the swamp as before.
- This starts the Tour the Mansion mode

Step 10. Reset everything when Tour the Mansion is complete

Todo: Need to explain this fully

- When Tour the Mansion completes, all the mansion awards are reset and a new random one is selected.
- If Tour the Mansion ends before the ball ends, no mansion award can be awarded until the next ball.

Recipe: Attack From Mars Super Jets

This guide shows you how to build an MPF config for *Attack From Mar's* Super Jets feature. The idea is you can use this as a guide to implement a similar feature in your machine.

Note: This recipe requires MPF 0.33 or newer.

This guide uses the following concepts in MPF that you should be familiar with:

- *Modes*
- *Counter Logic Blocks*
- *Shows*

You can find the complete runnable machine config for this recipe in the `cookbook/AFM_super_jets` folder of the mpf-examples repository on GitHub.

What is a Super Jets mode?

In Attack From Mars, Super Jets occur when the player hits the jet bumpers in the top right of the playfield 100 times in the course of a game. The effect of Super Jets is that once the mode is active, each jet bumper hit is worth 3,000,000 points instead of 1,000,000. The mode stops when the ball drains, but once achieving it, it only takes 25 more jet bumper hits to restart it.

Here are the specific rules we need to implement:

Super Jets

- Jet Bumper hits are initially 1,000,000 per hit.
- Each completion of the two inlanes above the jet bumpers add 50,000 to each jet bumper hit (1,050,000, 1,100,000, and so on.)
- Lit inlanes are movable with the flippers.
- One the inlane value reaches 2,000,000 per it, the inlanes stop adding 50,000 when completed.
- Super Pops occur when 100 jet bumper hits occue in the game. The amount of hits carry over to the next ball.
- Once the Super Pops mode has been made, the mode is active until the ball drains.
- Super Pops can be restarted by hitting the jet bumpers 25 more times.
- Once Super Pops have been made, the Super Pops insert on the playfield turns on and stays on.

Step 1. The machine-wide prerequisites

Before we dig into how to handle the mode itself, we need to create a machine-wide config that has all the devices we'll need, including the switches for the jet bumpers and lanes.

Here's what our machine config looks like. (Note that this is complete in terms of what we need to make this recipe work, but if you have a real Attack From Mars then you'll probably have a lot more than this in your machine config file. Also, the coil, switch, and light numbers are generic and need to be changes for a real machine.)

Notice the "player_vars" section. It has a player variable named "sj_active". We will explain this later on, but for now we'll just say that it is how we will tell if we are starting Super Jets for the first time or resuming it after starting it but draining.

```
#config_version=5

player_vars:
  sj_active:
    value_type: int
    initial_value: 0

modes:
  - super_jets_setup
  - super_jets

switches:
  s_left_flipper:
    number: 0
    tags: left_flipper
  s_right_flipper:
    number: 71
    tags: right_flipper
  s_credit:
    number: 6
    tags: start
  s_outhole:
    number: 8
    tags:
  s_left_bumper:
    number: 17
    tags:  jets
  s_middle_bumper:
    number: 18
    tags: jets
  s_right_bumper:
    number: 19
    tags: jets
  s_right_rollover:
    number: 22
    tags: playfield_active, right_rollover
  s_left_rollover:
    number: 23
    tags: playfield_active, left_rollover
  s_trough_5:
    number: 36
```

(continues on next page)

(continued from previous page)

```
    tags:
  s_trough_4:
    number: 37
    tags:
  s_trough_3:
    number: 38
    tags:
  s_trough_2:
    number: 39
    tags:
  s_trough_1:
    number: 40
    tags:

virtual_platform_start_active_switches: s_trough_1 s_trough_2 s_trough_3 s_trough_4 s_trough_5

coils:
 c_flipper_left_main:
   number: 0
   default_pulse_ms: 20
 c_flipper_left_hold:
   number: 1
   allow_enable: true
 c_flipper_right_main:
   number: 2
   default_pulse_ms: 20
 c_flipper_right_hold:
   number: 3
   allow_enable: true
 c_trough_eject:
   number: 4
   allow_enable: true
 c_left_bumper:
   number: c01
   label:
   tags:
   default_pulse_ms: 25
 c_middle_bumper:
   number: c02
   label:
   tags:
   default_pulse_ms: 25
 c_right_bumper:
   number: c03
   label:
   tags:
   default_pulse_ms: 25
 c_ball_eject:
   number: c12
   label:
   tags:
   default_pulse_ms: 20
 c_outhole:
   number: c14
```

(continues on next page)

(continued from previous page)

```
   label:
   tags:
   default_pulse_ms: 20

lights:
  l_right_rollover:
    number: 5
  l_left_rollover:
    number: 7
  l_super_jets:
    number: 9

ball_devices:
  bd_drain:
    ball_switches: s_outhole
    eject_coil: c_outhole
    eject_targets: bd_trough
    tags: drain, outhole
  bd_trough:
    ball_switches: s_trough_1, s_trough_2, s_trough_3, s_trough_4, s_trough_5
    eject_coil: c_ball_eject
    tags: trough, home

  autofire_coils:
    left_jet:
     coil: c_left_bumper
     switch: s_left_bumper
   mid_jet:
     coil: c_middle_bumper
     switch: s_middle_bumper
   right_jet:
     coil: c_right_bumper
     switch: s_right_bumper

playfields:
    playfield:
        default_source_device: bd_trough
        tags: default

##! mode: super_jets_setup
# mode will be defined below
##! mode: super_jets
# mode will be defined below
```

Step 2. Add Super Jets values

We'll start off with the easier mode first as all the heavy lifting is handled by the setup mode for Super Jets. In super_jets.yaml, we set up our starting events for the mode itself, the values of the jet bumpers when Super Jets are active, and also a call to show a slide stating that Super Jets are active.

```
##! mode: super_jets
mode:
```

(continues on next page)

(continued from previous page)

```
  start_events: Super_Jets_Go, Super_Jets_Resume_Go
  priority: 300
variable_player:
  s_left_bumper_active:
    score: 3000000|block
  s_middle_bumper_active:
    score: 3000000|block
  s_right_bumper_active:
    score: 3000000|block


show_player:
  mode_super_jets_started:
    super_jets_startup:
      loops: 0
    Super_Jets_on:
      show_tokens:
        lights: l_super_jets
```

Stepping through how we're using each setting:

```
start_events: Super_Jets_Go, Super_Jets_Resume_Go
```

The way the super_jets mode is called is if either "Super_Jets_Go" or "Super_Jets_Resume_Go" are posted.

```
s_left_bumper_active:
  score: 3000000|block
```

Everytime "s_left_bumper_active" is seen, the score has 3,000,000 points added onto it. The *|block* is used to prevent any other instances that awards points for hitting "s_left_bumper_active" from adding points as well.

This code is used for all three jets.

```
show_player:
  mode_super_jets_started:
    super_jets_startup:
      loops: 0
```

The Show Player shows the slide names "super_jets_started" at the start of the mode. The settings in super_jets_started.yaml dictate the size, font, and duration of the slide being used.

```
Super_Jets_on:
  show_tokens:
    lights: l_super_jets
```

Plays the show called "Super_Jets_on" when this mode starts, lighting the Super Jets light on the playfield.

Step 3. Create an setup mode for Super Jets

Next we need to create a mode called "super_jets_setup" to control when to call the "super_jets" mode. There's lot going on here, but we'll go through it step by step.

```
##! mode: super_jets_startup
#config_version=5

    mode:
      start_events: ball_starting
      priority: 200

    shots:
      jets:
        switch: s_right_bumper, s_left_bumper, s_middle_bumper
      right_rollover:
        switch: s_right_rollover
        show_tokens:
          light: l_right_rollover
      left_rollover:
        switch: s_left_rollover
        show_tokens:
          light: l_left_rollover

    shot_groups:
      rollover_lanes:
        shots: right_rollover, left_rollover
        rotate_left_events: s_left_flipper_active
        rotate_right_events: s_right_flipper_active
        reset_events:
          rollover_lanes_lit_complete: 1s

    counters:
      lb_jets_count:
        count_events: jets_hit
        starting_count: 0
        count_complete_value: 100
        count_interval: 1
        direction: up
        persist_state: true
        events_when_complete: Super_Jets_Go
        debug: true
      lb_jets_resume:
        enable_events: mode_base_started{current_player.sj_active>0}
        count_events: jets_hit
        starting_count: 0
        count_complete_value: 25
        count_interval: 1
        direction: up
        persist_state: false
        events_when_complete: Super_Jets_Resume_Go
        debug: true
        reset_on_complete: true
      lb_rollover_complete_count:
        count_events: rollover_lanes_complete
        events_when_hit: rollover_lanes_done
        starting_count: 0
        count_complete_value: 40
        reset_on_complete: false
        direction: up
```

(continues on next page)

(continued from previous page)

```
      persist_state: false

  event_player:
    Super_Jets_Go:
      start_mode_super_jets
    Super_Jets_Go_Again:
     start_mode_super_jets

  variable_player:
    s_left_bumper_active:
      score: 1000000 + (device.counters.lb_rollover_complete_count.value * 50000)
    s_middle_bumper_active:
      score: 1000000 + (device.counters.lb_rollover_complete_count.value * 50000)
    s_right_bumper_active:
      score: 1000000 + (device.counters.lb_rollover_complete_count.value * 50000)
    rollover_lanes_complete:
      score: 1000
    mode_super_jets_started:
      sj_active:
        int: 1
        action: set

 show_player:
   mode_super_jets_setup_started{current_player.sj_active>0}:
     Super_Jets_on:
       show_tokens:
         lights: l_super_jets
```

Let's look at each of these settings:

```
start_events: ball_starting
```

Here, we are saying that we want "super_jets_setup" to start as soon as the game starts a ball, including extra balls.

```
shots:
  jets:
    switch: s_right_bumper, s_left_bumper, s_middle_bumper
  right_rollover:
    switch: s_right_rollover
    show_tokens:
      light: l_right_rollover
  left_rollover:
    switch: s_left_rollover
    show_tokens:
      light: l_left_rollover
```

This section establishes our shots. Any time "s_right_bumper", "s_left_bumper", or "s_middle_bumper" is activated, the shot "jet" will register a hit.

"right_rollover" and "left_rollover" will show a hit any time their associated switch is made. Also, when their shots are made, their corresponding insert will also light up because we have a "show_tokens" section listing a light.

```
shot_groups:
  rollover_lanes:
    shots: right_rollover, left_rollover
    rotate_left_events: s_left_flipper_active
    rotate_right_events: s_right_flipper_active
    reset_events:
      rollover_lanes_lit_complete: 1s
```

This section is to set up the behavior of our rollover lanes. First, we list our shots, “right_rollover” and “left_rollover”. Then we designate our left flipper as a switch to rotate our shots left, and the right flipper to rotate the shots right. This is how we can use the flippers to move a lit rollover to the other lane to try and get the ball to go into an unlit rollover lane and complete the rollovers. “reset_events” is used to pause the shot group for 1 second as the rollover lanes are reset so they are both off again.

```
counters:
  lb_jets_count:
    count_events: jets_hit
    starting_count: 0
    count_complete_value: 100
    count_interval: 1
    direction: up
    persist_state: true
    events_when_complete: Super_Jets_Go
    debug: true
  lb_jets_resume:
    enable_events: mode_base_started{current_player.sj_active>0}
    count_events: jets_hit
    starting_count: 0
    count_complete_value: 25
    count_interval: 1
    direction: up
    persist_state: false
    events_when_complete: Super_Jets_Resume_Go
    debug: true
    reset_on_complete: true
  lb_rollover_complete_count:
    count_events: rollover_lanes_complete
    events_when_hit: rollover_lanes_done
    starting_count: 0
    count_complete_value: 40
    reset_on_complete: false
    direction: up
    persist_state: false
```

This is the heart of the mode. There are three counters here to help control the program.

“lb_jets_count” is the main counter. It is set up to increment from 0 to 100 every time the jets shot registers a hit, which is set up to include all the jet bumpers. By using “persist_state: true” we have the program not reset the count to 0 if the ball drains. If it takes all 3 balls for the player to hit 100 hits, they can still get Super Jets to start. When the counter hits 100, it causes the event “Super_Jets_Go” to post, and the event player later in the code handles what needs to be done now that it has posted.

“lb_jets_resume” is a similar counter, but it has a few very important differences. First, it has an “enable_events” requirement. If “sj_active” is not greater than 0, this counter will not run. That means that the previous counter, "lb_jets_count", had to start the super_jets mode first, and that the variavle "sj_active" has to somehow be set to greater than 0. When it is active, the counter counts up from 0 to 25. At 25, the counter stops and posts the "Super_Jets_Resume_Go" event. Another important difference is that we use "persist_state: false" to reset the counter on every ball. For example, a player can't get 12 hits in the jets, drain, and then expect Super Jets to start by hitting the jets just 13 more times. It has to be 25 without draining.

The final counter is for the rollover lanes, "lb_rollover_complete_count". We use this to track how many times the rollovers are comeplete, form 0 to 40. We use 40, because 50,000 * 40 = 2,000,000 which is the maximum addition of points we can add to the jets if not in Super Jets mode. By using "persist_state: false" we reset the count on the end of every ball back to 0.

```
event_player:
  Super_Jets_Go:
    start_mode_super_jets
  Super_Jets_Go_Again:
    start_mode_super_jets
```

Here is where we call our modes depending on what events are posted by the mode. Both events, "Super_Jets_Go" and "Super_Jets_Resume_Go" call the same mode to start, "super_jets", but because we have two different counters calling the mode under different conditions, we have to set it up like this.

```
variable_player:
  s_left_bumper_active:
    score: 1000000 + (device.counters.lb_rollover_complete_count.value * 50000)
  s_middle_bumper_active:
    score: 1000000 + (device.counters.lb_rollover_complete_count.value * 50000)
  s_right_bumper_active:
    score: 1000000 + (device.counters.lb_rollover_complete_count.value * 50000)
  rollover_lanes_complete:
    score: 1000
  mode_super_jets_started:
    sj_active:
      int: 1
      action: set
```

This is how the scoring is handled before Super Pops is active. Each jet bumper hit is worth 1,000,000 at the start. But, we also have to add 50,000 points for each time the rollovers are completed. To do that, we take the value of the counter, "lb_rollover_complete_count" and multiply it by 50000. Then we add that value to the standard 1,000,000. Remember in "super_jets" that we added *|block* to the end of the scoring? That was in part to keep these lines from continuing to add to the score, and to just have the scoring from "super_jets.yaml" to appear.

We also have a small score for when the rollover lanes are completed.

What is "sj_active"? This is a player variable set up previously to help with determining when to use which counter to activate super jets. Initially, the game sets "sj_active" to an integer value of 0. But, when Super Pops are activated by "lb_jets_count" because we hit the target of 100 hit, the variable player sets "sj_active" to an integer of 1 as the mode starts. Now, if the ball drains, and a new ball is launched, "lb_jets_resume" will be enabled to start counting, and because its count ends at 25 instead of 100, it will call super_jets before "lb_jets_count". "sj_active" will also stay at a value of 1 because every time the super_jets mode is called, we set "sj_active" is set to 1.

```
show_player:
  mode_super_jets_setup_started{current_player.sj_active>0}:
    Super_Jets_on:
      show_tokens:
        lights: l_super_jets
```

When “sj_active” has been set to 1, it is greater than 0. Now, the light for Super Jets will stay on from now on whenever a ball starts, and the super_jets_setup mode starts.

Step 4. Set up your Super Jets Slide

Here we set up a quick slide that pops up on the DMD when we’ve started Super Pops.

```
##! mode: super_jets_startup
    - duration: 2s
    slides:
      super_jets_startup:
        widgets:
        - type: text
          text: SUPER JETS
          font_size: 20
          y: 60%
          priority: 200
```

Step 5. Add the light for Super Jets

And finally, we set up a lightshow for turning on the Super Jets insert on the playfield.

```
##! mode: Super_Jets_on
    - time: 0
    lights:
      l_super_jets: ff
```

At this point you should have a working Super Pop mode. If any of this feels unclear or I’ve muddied up the explanation, feel free to join the discussion in the forums at https://groups.google.com/forum/#!topic/mpf-users/oVwBRQOgodY .

Recipe: Rollover Lanes (with Lane Change)

This guide shows you how to build an MPF config for rotating rollover lanes, as found in *Indiana Jones*, *Attack From Mars*, *Medieval Madness*, and many, many more.

What are Rollover Lanes?

Rollover lanes are found where pinball machines have a series of parallel lanes the ball can roll through. These are commonly found at the top of the playfield, often above pop bumpers and accessed via the outer orbit loop. Some games, like *Medieval Madness*, also use the outlanes and return lanes together as a group of rollover lanes

Each lane in the rollover lane group has a switch and a light. To start, all the lights are off. When the ball passes through a lane, that lane's light turns on. When the player turns on all the lights, they are awarded some prize and the lanes reset to off.

What is a Lane Change?

Games that use rollover lanes usually incorporate a "lane change" feature to make the completion easier. Lane changes use the flipper buttons to rotate the lit and unlit lane shots, shifting them left and right according to which button is pressed.

If a ball is about to enter a lane that's already been lit, the player can use the flipper buttons to shift the lanes so that the lane with the ball is unlit when the ball rolls over. By changing the lanes ahead of the ball, the player can complete the lane set more frequently—and it also gives the player more to do while the ball is away from the flippers!

Step 1. Create a lane change mode

Lane changes are typically available at all times during a game, so it's wise to create a separate mode for them. This mode can be run at the same time as other modes (but stopped any time, maybe during wizard modes if you want).

The first thing our mode needs is *shots:*. Each lane will count as a shot, and for this example we'll use the I-N-D-Y lanes from the Indiana Jones pinball game. We'll assume that the machine has switches defined in the `switches:` config section for each of the top lanes, called `s_top_lane_1` through `s_top_lane_4`.

```
##! mode: top_lanes
mode:
  start_events: start_mode_top_lanes
  stop_events: stop_mode_top_lanes, ball_will_end

shots:
  top_lane_i:
    switch: s_top_lane_1
  top_lane_n:
    switch: s_top_lane_2
  top_lane_d:
    switch: s_top_lane_3
  top_lane_y:
    switch: s_top_lane_4
```

Step 2. Creating a profile for the lanes

We can create a *shot_profile* for the top lanes that starts with the light on, and turns it off after the shot is hit.

```
##! mode: top_lanes
shot_profiles:
  top_lane_profile:
    states:
      - name:
```

(continues on next page)

(continued from previous page)

```
      show: off
    - name: hit
      show: on
```

Note: In common pinball parlance, a shot is "lit" if the player should try an hit it. In almost all cases this means the light for the shot is on (i.e. "lit"), but rollover lane shots are the opposite: the light is **off** when the shot is lit, and **on** after the shot is hit.

We can apply our shot profile to each of the `shots` we defined earlier. Each lane has its own light, which we can specify using `show_tokens`. This tells MPF that when it plays the show (in this case, the "on" show) for a specific shot, use the light that corresponds to that shot.

We'll assume the machine has four lights defined in the `lights:` config section, called `l_top_lane_1` through `l_top_lane_4`

```
##! mode: top_lanes
shots:
  top_lane_i:
    switch: s_top_lane_1
    profile: top_lane_profile
    show_tokens:
      led: l_top_lane_1
  top_lane_n:
    switch: s_top_lane_2
    profile: top_lane_profile
    show_tokens:
      led: l_top_lane_2
  top_lane_d:
    switch: s_top_lane_3
    profile: top_lane_profile
    show_tokens:
      led: l_top_lane_2
  top_lane_y:
    switch: s_top_lane_4
    profile: top_lane_profile
    show_tokens:
      led: l_top_lane_2
```

Step 3. Creating a shot_group for the lanes

To tell MPF that the four lane shots are related to each other, we create a *shot_group* with all the shots in it.

Shot groups are powerful because they control behavior of all the shots together. In this case, we'll use our shot group to:

- Rotate the lit and hit shots
- Trigger an event when all the shots are hit
- Reset all the shots to be lit

```
##! mode: top_lanes
shot_groups:
  top_lane_group:
    shots: top_lane_i, top_lane_n, top_lane_d, top_lane_y
    reset_events: top_lane_group_hit_complete
    rotate_left_events: s_flipper_left_active
    rotate_right_events: s_flipper_right_active
```

The `rotate_left_events` and `rotate_right_events` allow the lane changes based on the flipper events.

A shot group tracks the profile state of each shot, and will post an event *(shot_group_name)_(state_name)_complete* event whenever all shots in the group are the same state. In the profile "top_lane_profile" we said that the second state is called "hit", so we can use the *top_lane_group_hit_complete* event to know that all the shots are hit. The name of the state is up to you.

When the *top_lane_group_hit_complete* event is triggered, the shot group will reset all the shots to their initial state: the "lit" state of the profile with the light off. Now the lanes are ready for the player to complete again!

Step 4. Rewards for rollover lane completion

Presumably when the player completes the rollover lanes, they should get some reward: a bonus multiplier, a counter advance, some points. . . it can be anything.

In this example, we'll use the *variable_player:* to award the player 10,000 points for completing the rollover lanes, and also increase a the bonus multiplier for the end-of-game bonus.

```
variable_player:
  top_lane_group_hit_complete:
    score: 10000
    bonus_multiplier: 1
```

See *End of Ball Bonus* for details on *bonus_multiplier*.

The full mode config code

```
##! mode: top_lanes
mode:
  start_events: start_mode_top_lanes
  stop_events: stop_mode_top_lanes, ball_will_end

shots:
  top_lane_i:
    switch: s_top_lane_1
    profile: top_lane_profile
    show_tokens:
      led: l_top_lane_1
  top_lane_n:
    switch: s_top_lane_2
    profile: top_lane_profile
    show_tokens:
```

(continues on next page)

(continued from previous page)

```
      led: l_top_lane_2
  top_lane_d:
    switch: s_top_lane_3
    profile: top_lane_profile
    show_tokens:
      led: l_top_lane_2
  top_lane_y:
    switch: s_top_lane_4
    profile: top_lane_profile
    show_tokens:
      led: l_top_lane_2

shot_groups:
  top_lane_group:
    shots: top_lane_i, top_lane_n, top_lane_d, top_lane_y
    reset_events: top_lane_group_hit_complete
    rotate_left_events: s_flipper_left_active
    rotate_right_events: s_flipper_right_active

shot_profiles:
  top_lane_profile:
    states:
      - name:
        show: off
      - name: hit
        show: on

variable_player:
  top_lane_group_hit_complete:
    score: 10000
    bonus_multiplier: 1
```

Related Docs

- *shots:*
- *shot_groups:*
- *shot_profiles:*
- *variable_player:*

Recipe: GADGET Targets from Stern Batman '66

This guide shows you how to build an MPF config for *Batman 66's* GADGET targets. The idea is you can use this as a guide to implement a similar feature in your machine.

Note: This recipe requires MPF 0.53 or newer.

This guide uses the following concepts in MPF:

- *Modes*
- *Event player*
- *Accrual Logic Blocks*
- *Conditional Events*
- *Show player*
- *Shows*

TODO You can find the complete runnable machine config for this recipe in the `cookbook/B66_Gadget` folder of the mpf-examples repository on GitHub.

What is GADGET mode?

In Bataman '66, a player may hit each of the 6 stand-up targets representing the letters of the word "GADGET". When all letters have been hit, the player is awarded a "Gadget" which gives the players special ability in the game.

Here are the specific rules we need to implement:

GADGET

- Each letter begin unlit
- Letters become lit when hit individually
- When an already-lit letter is hit, award one adjacent unlit letter. (Friendly Neighbor)
- After all letters are hit, award a gadget and reset the letters to the beginning.
- Players may earn multiple gadgets
- Light the lockdown bar to indicate to the player that they have earned a gadget.

Using an earned Gadget is outside the scope of this document. This cookbook only covers earning gadgets.

Step 1. The machine-wide prerequisites

Before we dig into how to handle the mode itself, we need to create a machine-wide config that has all the devices we'll need, including the switches for the targets.

Here's what our machine config looks like. (Note that this is complete in terms of what we need to make this recipe work, but if you have a real Batman '66 then you'll probably have a lot more than this in your machine config file. Also, the coil, switch, and light numbers are generic and need to be changes for a real machine.)

Notice the "player_vars" section. It has a two player variables named "gadgets_available" & "gadgets_earned". This exists outside of the mode to 'protect' earned, but unused gadgets from being reset in the rare cases when we may need to stop the mode that allows players to earn gadgets.

```
#config_version=5

modes:
  - gadget
```

(continues on next page)

(continued from previous page)

```
player_vars:
  gadgets_available:
    initial_value: 0
  gadgets_earned:
    initial_value: 0

switches:
  s_left_flipper:
    number: 0
    tags: left_flipper, playfield_active
  s_right_flipper:
    number: 71
    tags: right_flipper
  s_credit:
    number: 6
    tags: start
  s_outhole:
    number: 8
    tags:
  s_gadget_g1:
    number: 17
    tags: gadget_targets
  s_gadget_a:
    number: 18
    tags: gadget_targets
  s_gadget_d:
    number: 19
    tags: gadget_targets
  s_gadget_g2:
    number: 22
    tags: gadget_targets
  s_gadget_e:
    number: 23
    tags: gadget_targets
  s_gadget_t:
    number: 24
    tags: gadget_targets
  s_trough_6:
    number: 33
    tags:
  s_trough_5:
    number: 36
    tags:
  s_trough_4:
    number: 37
    tags:
  s_trough_3:
    number: 38
    tags:
  s_trough_2:
    number: 39
    tags:
  s_trough_1:
    number: 40
```

(continues on next page)

(continued from previous page)

```
    tags:
  s_start_button:
    number: 99
    tags: start, playfield_active

keyboard:
  s:
    switch: s_start_button

virtual_platform_start_active_switches: s_trough_1, s_trough_2, s_trough_3, s_trough_4, s_trough_5, s_
↪trough_6

coils:
  c_flipper_left_main:
    number: 0
    default_pulse_ms: 20
  c_flipper_left_hold:
    number: 1
    allow_enable: true
  c_flipper_right_main:
    number: 2
    default_pulse_ms: 20
  c_flipper_right_hold:
    number: 3
    allow_enable: true
  c_trough_eject:
    number: 4
    allow_enable: true
  c_ball_eject:
    number: c12
    label:
    tags:
    default_pulse_ms: 20
  c_outhole:
    number: c14
    label:
    tags:
    default_pulse_ms: 20

lights:
  l_gadget_g1:
    number: 5
    tags: gadget_letter
  l_gadget_a:
    number: 6
    tags: gadget_letter
  l_gadget_d:
    number: 7
    tags: gadget_letter
  l_gadget_g2:
    number: 8
    tags: gadget_letter
  l_gadget_e:
    number: 9
```

(continues on next page)

(continued from previous page)

```
    tags: gadget_letter
  l_gadget_t:
    number: 10
    tags: gadget_letter
  l_lockdown_bar:
    number: 11

ball_devices:
  bd_drain:
    ball_switches: s_outhole
    eject_coil: c_outhole
    eject_targets: bd_trough
    tags: drain, outhole
  bd_trough:
    ball_switches: s_trough_1, s_trough_2, s_trough_3, s_trough_4, s_trough_5
    eject_coil: c_ball_eject
    tags: trough, home

playfields:
  playfield:
    default_source_device: bd_trough
    tags: default

##! mode: gadget
```

Step 2. Create the Gadget Mode Config File

Next, we can start setting up our gadget mode; below you see the contents of `gadget.yaml`

```
##! mode: gadget
config:
  - logic_blocks.yaml
  - event_player.yaml
  - show_player.yaml
  - variable_player.yaml

mode:
  #this mode starts when the ball starts
  start_events: ball_started

  priority: 500
```

Stepping through how we're using each setting:

```
##! mode: gadget
config:
  - logic_blocks.yaml
```

The config section imports other config files; this is often easier to manage than on long config file.

```
##! mode: gadget
  priority: 500
```

The Gadget mode in Batman '66 is nearly always running and rarely blocked, so we have assigned it a very high priority, but one that can still be superceded if the need arises.

Step 3. Create the Accrual Logic Block

Also in our mode config folder, we will add `logic_blocks.yaml` to hold our mode-specific logic_blocks. In this case, we're using an *Accrual Logic Blocks* to track when all of the letters have been hit.

```
##! mode: gadget
accruals:
  gadget_accrual:
    events:
      - gadget_g1_complete # index [0]
      - gadget_a_complete # index [1]
      - gadget_d_complete # index [2]
      - gadget_g2_complete # index [3]
      - gadget_e_complete # index [4]
      - gadget_t_complete # index [5]
    reset_on_complete: true
    disable_on_complete: false
    reset_events: mode_gadget_started
    events_when_complete: award_gadget, reset_gadget_lights
```

Stepping through once again:

```
##! mode: gadget
accruals:
  gadget_accrual:
```

These two lines simply tell MPF that we have an accrual and we've named it "gadget_accrual".

```
##! mode: gadget
    events:
      - gadget_g1_complete # index [0]
      - gadget_a_complete # index [1]
      - gadget_d_complete # index [2]
      - gadget_g2_complete # index [3]
      - gadget_e_complete # index [4]
      - gadget_t_complete # index [5]
```

Next, we have a list of events for the accrual to track. Accruals behave like arrays, so I added a comment after each event to help me remember the index of each event. We'll need to reference these events and their index later.

```
##! mode: gadget
    reset_on_complete: true
```

Once the player has hit all of the letters, we want the accrual to reset so that they can earn more Gadgets.

```
##! mode: gadget
    disable_on_complete: false
```

We also have to tell MPF to leave our accrual enabled, even after it's completed.

```
##! mode: gadget
    events_when_complete: award_gadget, reset_gadget_lights
```

When the accrual is complete, we want it to fire the two events in the list. We'll see what these events actually do a bit later.

Step 4. Create the 'Friendly Neighbor' Behavior

The Gadget targets exhibit a player-friendly behavior that makes them easier to complete. If the player hits a letter that is already complete, the game will award one of the neigbhoring targets if they are incomplete. To accomplish this, we'll use conditional events in our event player.

```
##! mode: gadget
event_player:
  #plus one gadget when accrual is complete
  award_gadget:
    - gadgets_earned
    - gadgets_available

  s_gadget_g1_active:
    #if the g is hit, and unlit
    - gadget_g1_complete{device.accruals.gadget_accrual.value[0]==False}
    #award a if we already have g1
    - gadget_a_complete{device.accruals.gadget_accrual.value[0]==True}
  s_gadget_a_active:
    #if a is hit and unlit
    - gadget_a_complete{device.accruals.gadget_accrual.value[1]==False}
    #award g1 if we already have a
    - gadget_g1_complete{device.accruals.gadget_accrual.value[0]==False and device.accruals.gadget_
↪accrual.value[1]==True}
    #award d if we already have a and g1
    - gadget_d_complete{device.accruals.gadget_accrual.value[0]==True and device.accruals.gadget_
↪accrual.value[1]==True and device.accruals.gadget_accrual.value[2]==False}
  s_gadget_d_active:
    - gadget_d_complete{device.accruals.gadget_accrual.value[2]==False}
    - gadget_a_complete{device.accruals.gadget_accrual.value[1]==False and device.accruals.gadget_
↪accrual.value[2]==True}
    - gadget_g2_complete{device.accruals.gadget_accrual.value[1]==True and device.accruals.gadget_
↪accrual.value[2] and device.accruals.gadget_accrual.value[3]==False}
  s_gadget_g2_active:
    - gadget_g2_complete{device.accruals.gadget_accrual.value[3]==False}
    - gadget_d_complete{device.accruals.gadget_accrual.value[2]==False and device.accruals.gadget_
↪accrual.value[3]==True}
    - gadget_e_complete{device.accruals.gadget_accrual.value[2]==True and device.accruals.gadget_
↪accrual.value[3]==True and device.accruals.gadget_accrual.value[4]==False}
  s_gadget_e_active:
    - gadget_e_complete{device.accruals.gadget_accrual.value[4]==False}
    - gadget_g2_complete{device.accruals.gadget_accrual.value[3]==False and device.accruals.gadget_
↪accrual.value[4]==True}
    - gadget_t_complete{device.accruals.gadget_accrual.value[3]==True and device.accruals.gadget_
↪accrual.value[4]==True and device.accruals.gadget_accrual.value[5]==False}
  s_gadget_t_active:
    - gadget_t_complete{device.accruals.gadget_accrual.value[5]==False}
```

(continues on next page)

(continued from previous page)

```
    - gadget_e_complete{device.accruals.gadget_accrual.value[4]==False and device.accruals.gadget_
↪accrual.value[5]==True}
```

There's a lot happening here, so let's get the easy stuff out of the way first:

```
##! mode: gadget
  award_gadget:
    - gadgets_earned
    - gadgets_available
```

The "award_gadget" event - triggered by the accrual completion, simply adds one to both `player_vars` we configured in step one.

```
##! mode: gadget
  s_gadget_a_active:
    #if a is hit and unlit
    - gadget_a_complete{device.accruals.gadget_accrual.value[1]==False}
```

This is our first conditional event, which covers the case of "a" having not yet been hit. When the "a" switch is active, trigger the event "gadget_a_complete" if it hasn't been seen by the accrual. Note the `value[1]` which refers to the 2nd index of our accrual.

```
##! mode: gadget
    - gadget_g1_complete{device.accruals.gadget_accrual.value[0]==False and device.accruals.gadget_
↪accrual.value[1]==True}
```

Now, we trigger gadget_g1_complete if it hasn't been seen by the accrual AND "a" is already complete.

```
##! mode: gadget
    - gadget_d_complete{device.accruals.gadget_accrual.value[0]==True and device.accruals.gadget_
↪accrual.value[1]==True and device.accruals.gadget_accrual.value[2]==False}
```

The final case for "a" is if "g1" and "a" are complete, then trigger the event for "d" if it hasn't been triggered yet.

If all three cases "g1", "a" and "d" have all been captured by the accrual, then nothing happens.

We repeat this series of conditional events for all letters. "g1" and "t" have fewer events because they each only have one neighboring target.

Step 5. Add Your Light Shows

Now, we'll add some visual feedback for the player to know when they've been awarded a letter, or completed the "gadget_accrual". This show is "light_gadget_letter.yaml" and it's in the "shows" folder for the mode. It's pretty straightforward, but uses tokens and tags to be efficient.

```
##! show: light_gadget_letter
- time: 0
  lights:
    (gadget_letter_made_led): (gadget_letter_made_color)

- time: +.05
  lights:
```

(continues on next page)

(continued from previous page)

```
    (gadget_letter_made_led): off

- time: +.05
  lights:
    (gadget_letter_made_led): (gadget_letter_made_color)

- time: +.05
  lights:
    (gadget_letter_made_led): off

- time: +.05
  lights:
    (gadget_letter_made_led): (gadget_letter_made_color)

- time: +.05
  lights:
    (gadget_letter_made_led): off

- time: +.05
  lights:
    (gadget_letter_made_led): (gadget_letter_made_color)

- time: +.05
  lights:
    (gadget_letter_made_led): off

- time: +.05
  lights:
    (gadget_letter_made_led): (gadget_letter_made_color)

- time: +.05
  lights:
    (gadget_letter_made_led): off

- time: +.05
  lights:
    (gadget_letter_made_led): (gadget_letter_made_color)

- time: +.05
  lights:
    (gadget_letter_made_led): off

- time: +.05
  lights:
    (gadget_letter_made_led): (gadget_letter_made_color)

- time: +.05
  lights:
    (gadget_letter_made_led): off

- time: +.05
  lights:
    (gadget_letter_made_led): (gadget_letter_made_color)
```

(continues on next page)

(continued from previous page)

```
- time: +.05
  lights:
    (gadget_letter_made_led): off

- time: +.05
  lights:
    (gadget_letter_made_led): (gadget_letter_final_color)

  duration: -1
```

This show isn't terribly complicated, but let's look at some of the features.

```
##! show: light_gadget_letter
- time: 0
  lights:
    (gadget_letter_made_led): (gadget_letter_made_color)

- time: +.05
  lights:
    (gadget_letter_made_led): off
```

When the show starts, it accepts a token from the `show_player` (we'll configure that next), that tells MPF what corresponding light(s) we're going to flash, and what color to flash them.

In a real Batman '66, we would simply flash the light because the inserts are yellow. However, since many custom games are using RGB LED, we'll allow for any color the builder prefers.

```
##! show: light_gadget_letter
- time: +.05
  lights:
    (gadget_letter_made_led): (gadget_letter_final_color)

  duration: -1
```

The last step is special for two reasons. We're passing in a second color that will be 'held' at the end of the show indefinitely as indicated by `duration -1`. We've done this in order to allow for the same show to end in a 'lit' or 'unlit' state, depending on our need in a situation.

In the code you can download from the link at the beginning of this cookbook, there is another show that lights the LED on the lockdown bar, but it's not worth explaining here.

Step 6. Configure the Show Player

Our show player is watching for events and triggering the appropriate shows.

```
show_player:
  gadget_g1_complete:
    light_gadget_letter:
      priority: 10
      key: gadget_g1_hit_show
      show_tokens:
        gadget_letter_made_led: l_gadget_g1
        gadget_letter_made_color: yellow
        gadget_letter_final_color: yellow
```

```
  gadget_g1_complete:
    light_gadget_letter:
```

When the "gadget_g1_complete" event is triggered, start the "light_gadget_letter" show starts.

```
      key: gadget_g1_hit_show
```

We'll add a key to the show so that we can keep re-using the same show for all the letters.

```
        gadget_letter_made_led: l_gadget_g1
        gadget_letter_made_color: yellow
        gadget_letter_final_color: yellow
```

Finally, we pass show tokens to the show to tell it what light and what color we want for the on steps and the final step. This repeats for all of the individual letters.

```
show_player:
  reset_gadget_lights:
    light_gadget_letter:
      priority: 10
      show_tokens:
        gadget_letter_made_led: gadget_letter
        gadget_letter_made_color: yellow
        gadget_letter_final_color: 000000
```

"reset_gadget_lights" is fired by the accrual when it's complete. We make two small, but important changes. First "gadget_letter" is a `tag` from the machine config assigned to all the letters in GADGET. This will cause all of the letters to play the show simultaneously. Second, "gadget_letter_final_color" is now black/off. This effectively resets the lights and prepares the inserts for a new accrual to begin.

At this point, your Gadget mode is ready to go. You can add scoring in a `variable_player` and extend this by writing ways to use gadgets and reduce the "gadgets_available" player_vars. If any of this feels unclear or I've muddied up the explanation, feel free to join the discussion in the forums at https://groups.google.com/forum/#!topic/mpf-users/oVwBRQOgodY .

Recipe: Modifying the game mode - Dual launch devices

While the following example adds a very unusual feature, it makes for a very simple and clean example of how to override default behavior in MPF.

One of the base assumptions that the MPF system makes is that there is only one launch device. While quite reasonable, what if you wanted both a left and right plugger? You can add a ball device for each system, but MPF expects a *default_source_device* to be defined for the main playfield, and it won't take a list. This means at the start of each player round, the game can only kick up a ball in the default device.

Here's what the hardware configuration for two plungers (and troughs) would look like:

```
#config_version=5
switches:
  # Cabinet Buttons
  s_start_button:
    number:
```

(continues on next page)

(continued from previous page)

```
    tags: start
  s_left_launch_button:
    number:
  s_right_launch_button:
    number:
  # Plunger Trough
  s_left_plunger_lane:
    number:
  s_right_plunger_lane:
    number:
  s_left_trough1:
    number:
  s_left_trough2:
    number:
  s_right_trough1:
    number:
  s_right_trough2:
    number:

coils:
  c_left_plunger:
    number:
    default_pulse_ms: 20
  c_left_trough_eject:
    number:
    default_pulse_ms: 20
  c_right_plunger:
    number:
    default_pulse_ms: 20
  c_right_trough_eject:
    number:
    default_pulse_ms: 20

ball_devices:
  bd_left_trough:
    ball_switches: s_left_trough1, s_left_trough2
    eject_coil: c_left_trough_eject
    tags: trough, home, drain
    eject_targets: bd_left_plunger
  bd_left_plunger:
    ball_switches: s_left_plunger_lane
    eject_coil: c_left_plunger
    player_controlled_eject_event: s_left_launch_button_active
    eject_timeouts: 1s
  bd_right_trough:
    ball_switches: s_right_trough1, s_right_trough2
    eject_coil: c_right_trough_eject
    tags: trough, home, drain
    eject_targets: bd_right_plunger
  bd_right_plunger:
    ball_switches: s_right_plunger_lane
    eject_coil: c_right_plunger
    player_controlled_eject_event: s_right_launch_button_active
    eject_timeouts: 1s
```

(continues on next page)

(continued from previous page)

```
playfields:
  playfield:
    default_source_device: bd_left_plunger
    tags: default

virtual_platform_start_active_switches: s_left_trough1, s_left_trough2, s_right_trough1, s_right_trough2
```

It is the game mode that handles the ball start procedure and assumes a single launch device. Now MPF's game mode does a lot more than that, so in most cases you probably don't want to go through re-writing the whole thing just to change one behavior. Instead we will change the parts we need to.

First, see how the default game mode works. Within the MPF source library you'll see a directory called mpf/modes/game. This is just like the modes directory in your own game definitions. Let's look at the config file first (mpf/modes/game/config/game.yaml):

```
##! mode: game
#config_version=5
mode:
  start_events: game_start
  stop_events: game_ended, service_mode_entered
  priority: 20
  code: mpf.modes.game.code.game.Game
  game_mode: false  # this is the game so it is started outside of a game
  stop_on_ball_end: false
```

This is pretty straight-forward. First the standard mode settings, and then it points to the source for a Python module that defines a class called Game. We can look at that code in `mpf/modes/game/code/game.py`. While we won't repost the full source, you can look at it here. We won't get into all that it does, because we don't need to. Looking through the file, we really only need to know where this mode adds a ball to the playfield. That can be found as the last line of the `_start_ball()` method. It makes the following call:

```
self.machine.playfield.add_ball(player_controlled=True)
```

Looking at the `add_ball()` method from the playfield class (mpf/mpf/devices/playfield.py) we can see that it can actually take a source device as an argument:

```
add_ball(self, balls=1, source_device=None, player_controlled=False) -> bool:
   """Add live ball(s) to the playfield.
   Args:
     balls: Integer of the number of balls you'd like to add.
     source_device: Optional ball device object you'd like to add the
         ball(s) from.
     player_controlled: Boolean which specifies whether this event is
         player controlled. (See not below for details)
```

This means that what we really want is the game class except with slightly different `_start_ball()` method. To do that, we will define our own game mode. Just like any other mode we add it to our folder of modes. Your file layout will become as follows:

```
+-- config
    +-- config.yaml
```

(continues on next page)

(continued from previous page)

```
+-- data
+-- logs
+-- modes
    +-- game
        +-- __init__.py
        +-- config
            +-- game.yaml
        +-- code
            +-- __init__.py
            +-- game.py
```

Your `game.yaml` will look like this:

```
#config_version=5
mode:
  start_events: game_start
  stop_events: game_ended, service_mode_entered
  priority: 20
  code: game.MyGameName
  game_mode: False  # this is the game so it is started outside of a game
  stop_on_ball_end: False
```

Now for our own game mode class that inherits everything it needs from the original Game mode class:

```
from mpf.modes.game.code.game import Game

class MyGameName(Game):
  def __init__(self, *arg, **kwargs):
     super().__init__(*arg, **kwargs)
     self.log.debug("MyGameName init")

  async def _start_ball(self, is_extra_ball=False):
     """Perform ball start procedure.

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

     Posts a queue event called *ball_starting*, giving other modules the
     opportunity to do things before the ball actually starts. Once that
     event is clear, this method calls :meth:`ball_started`.
     """
     :
     # Cut and paste original game.py code for _start_ball() here.
     :
     # Replace self.machine.playfield.add_ball(player_controlled=True) with:
     left_switch_pressed_future = self.machine.switch_controller.wait_for_switch(self.machine.switches[
↪"s_left_launch_button"])
     right_switch_pressed_future = self.machine.switch_controller.wait_for_switch(self.machine.switches[
↪"s_right_launch_button"])
     first_switch = await Util.race({left_switch_pressed_future: "left", right_switch_pressed_future:
↪"right"})
     if first_switch == "left":
         self.machine.playfield.add_ball(source_device=self.machine.ball_devices['bd_left_plunger'],␣
↪player_controlled=True)
```

(continues on next page)

(continued from previous page)

```
    else:
        self.machine.playfield.add_ball(source_device=self.machine.ball_devices['bd_right_plunger'],
↪player_controlled=True)
```

Notice that we've only had to define our `_start_ball()` method. It is really just a copy of the original, except that we wait for one of the two launch buttons and then eject a ball on that side.

Finally, the `__init__.py` files are all empty.

Now, when you hit the start button on your game, both sides will load a ball for each plunger. Again, a weird thing to do, but a simple example of customizing the game mode when you run up against a default that doesn't work for your design.

Here is a complete example:

```
#config_version=5
switches:
  # Cabinet Buttons
  s_start_button:
    number:
    tags: start
  s_left_launch_button:
    number:
  s_right_launch_button:
    number:
  # Plunger Trough
  s_left_plunger_lane:
    number:
  s_right_plunger_lane:
    number:
  s_left_trough1:
    number:
  s_left_trough2:
    number:
  s_right_trough1:
    number:
  s_right_trough2:
    number:

coils:
  c_left_plunger:
    number:
    default_pulse_ms: 20
  c_left_trough_eject:
    number:
    default_pulse_ms: 20
  c_right_plunger:
    number:
    default_pulse_ms: 20
  c_right_trough_eject:
    number:
    default_pulse_ms: 20

ball_devices:
  bd_left_trough:
```

(continues on next page)

(continued from previous page)

```
    ball_switches: s_left_trough1, s_left_trough2
    eject_coil: c_left_trough_eject
    tags: trough, home, drain
    eject_targets: bd_left_plunger
  bd_left_plunger:
    ball_switches: s_left_plunger_lane
    eject_coil: c_left_plunger
    eject_timeouts: 1s
  bd_right_trough:
    ball_switches: s_right_trough1, s_right_trough2
    eject_coil: c_right_trough_eject
    tags: trough, home, drain
    eject_targets: bd_right_plunger
  bd_right_plunger:
    ball_switches: s_right_plunger_lane
    eject_coil: c_right_plunger
    eject_timeouts: 1s

playfields:
  playfield:
    default_source_device: bd_left_plunger
    tags: default

virtual_platform_start_active_switches: s_left_trough1, s_left_trough2, s_right_trough1, s_right_trough2

##! mode: game
#config_version=5
mode:
  start_events: game_start
  stop_events: game_ended, service_mode_entered
  priority: 20
  code: modes.game.code.game.MyGameName
  game_mode: false  # this is the game so it is started outside of a game
  stop_on_ball_end: false

from mpf.modes.game.code.game import Game
from mpf.core.utility_functions import Util

class MyGameName(Game):
  def __init__(self, *arg, **kwargs):
     super().__init__(*arg, **kwargs)
     self.log.debug("MyGameName init")

  async def _start_ball(self, is_extra_ball=False):
     """Perform ball start procedure.

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

     Posts a queue event called *ball_starting*, giving other modules the
     opportunity to do things before the ball actually starts. Once that
     event is clear, this method calls :meth:`ball_started`.
     """
     event_args = {
```

(continues on next page)

(continued from previous page)

```
        "player": self.player.number,
        "ball": self.player.ball,
        "balls_remaining": self.balls_per_game - self.player.ball,
        "is_extra_ball": is_extra_ball}

    self.debug_log("***************************************************")
    self.debug_log("***************** BALL STARTING *****************")
    self.debug_log("**                                               **")
    self.debug_log("**    Player: {}    Ball: {}   Score: {}".format(self.player.number,
                                                                 self.player.ball,
                                                                 self.player.score
                                                                 ).ljust(49) + '**')
    self.debug_log("**                                               **")
    self.debug_log("***************************************************")
    self.debug_log("***************************************************")

    await self.machine.events.post_async('ball_will_start', **event_args)
    '''event: ball_will_start
    desc: The ball is about to start. This event is posted just before
    :doc:`ball_starting`.
    args:
    ball: The ball number
    balls_remaining: The number of balls left in the game (not including this one)
    is_extra_ball: True if this ball is an extra ball (default False)
    player: The player number'''

    await self.machine.events.post_queue_async('ball_starting', **event_args)
    '''event: ball_starting
    desc: A ball is starting. This is a queue event, so the ball won't
    actually start until the queue is cleared.
    args:
    ball: The ball number
    balls_remaining: The number of balls left in the game (not including this one)
    is_extra_ball: True if this ball is an extra ball (default False)
    player: The player number'''

    # register handlers to watch for ball drain and live ball removed
    self.add_mode_event_handler('ball_drain', self.ball_drained)

    self.balls_in_play = 1

    self.debug_log("ball_started for Ball %s", self.player.ball)

    await self.machine.events.post_async('ball_started', **event_args)
    '''event: ball_started
    desc: A new ball has started.
    args:
    ball: The ball number
    balls_remaining: The number of balls left in the game (not including this one)
    is_extra_ball: True if this ball is an extra ball (default False)
    player: The player number'''

    if self.num_players == 1:
        await self.machine.events.post_async('single_player_ball_started')
```

(continues on next page)

(continued from previous page)

```
            '''event: single_player_ball_started
            desc: A new ball has started, and this is a single player game.'''
        else:
            await self.machine.events.post_async('multi_player_ball_started')
            '''event: multi_player_ball_started
            desc: A new ball has started, and this is a multiplayer game.'''
            await self.machine.events.post_async(
                'player_{}_ball_started'.format(self.player.number))
            '''event player_(number)_ball_started
            desc: A new ball has started, and this is a multiplayer game.
            The player number is the (number) in the event that's posted.'''

        if not hasattr(self.machine, "playfield") or not self.machine.playfield:
            raise AssertionError("The game did not define default playfield. Did you add tags: default to␣
↪one of your "
                                 "playfield?")

        left_switch_pressed_future = self.machine.switch_controller.wait_for_switch(self.machine.switches[
↪"s_left_launch_button"])
        right_switch_pressed_future = self.machine.switch_controller.wait_for_switch(self.machine.switches[
↪"s_right_launch_button"])
        first_switch = await Util.race({left_switch_pressed_future: "left", right_switch_pressed_future:
↪"right"})
        if first_switch == "left":
            self.machine.playfield.add_ball(source_device=self.machine.ball_devices['bd_left_plunger'],␣
↪player_controlled=True)
        else:
            self.machine.playfield.add_ball(source_device=self.machine.ball_devices['bd_right_plunger'],␣
↪player_controlled=True)
```

Recipe: Sequential Drop Target Banks

This guide shows you how to build an MPF config for a drop target bank with targets that must be hit in a specific order.

The mode starts with one target flashing as the correct target to hit, and all the rest off. Hitting the correct target will keep the hit target down, hold the light on, and flash the next target. Hitting an incorrect target will reset that target's coil and keep the light off.

Step 1. Create a sequential_drops mode and lane shots

We'll create a separate mode called *sequential_drops* to manage the game logic. Separate modes keep the code clean and make it easy to turn the drop sequence on and off as needed (e.g. during a multiball or wizard mode).

The first thing our mode needs is *shots:*. Each drop target will be a shot (in this example, we'll have four). Each shot has a switch, a light, and a shot profile to track its state.

Each shot will also have unique `advance_events` configured, which will advance the shot (from "off" to "lit") when its predecessor is hit, and again advance the shot (from "lit" to "down") when it is hit. The final shot does not need to advance from "lit" to "down" because the sequence resets when it's hit.

Each shot also has `reset_events` configured, so that the entire sequence can be reset after completion.

```
##! mode: sequential_drops

mode:
  start_events: start_mode_sequential_drops
  stop_events: stop_mode_sequential_drops
  priority: 200

shots:
  drop_1:
    advance_events: advance_drop_1, drop_1_lit_hit
    reset_events: reset_drop_sequence
    switch: s_drop_1
    profile: drop_sequence
    show_tokens:
      led: l_drop_1
  drop_2:
    advance_events: drop_1_lit_hit, drop_2_lit_hit
    reset_events: reset_drop_sequence
    switch: s_drop_2
    profile: drop_sequence
    show_tokens:
      led: l_drop_2
  drop_3:
    advance_events: drop_2_lit_hit, drop_3_lit_hit
    reset_events: reset_drop_sequence
    switch: s_drop_3
    profile: drop_sequence
    show_tokens:
      led: l_drop_3
  drop_4:
    advance_events: drop_3_lit_hit
    reset_events: reset_drop_sequence
    switch: s_drop_4
    profile: drop_sequence
    show_tokens:
      led: l_drop_4
```

Step 2. Create a profile for the targets

We can create a *shot_profile* for the targets that starts with the light off, flashes it after one advancement, and keeps the light on solid after a second advancement. By default, a shot will advance its profile when the shot is hit, but we don't want that here so we'll set `advance_on_hit: false`.

This profile uses three built-in shows, `off`, `flash`, and `on`. These shows accept the `show_tokens` from our shots. In this case, it is the LED we wish to control. You can create your own shows to change LED color, play sounds, etc.

```
##! mode: sequential_drops

shot_profiles:
  drop_sequence:
    advance_on_hit: false
```

(continues on next page)

(continued from previous page)

```
    states:
      - name: off
        show: off
      - name: lit
        show: flash
      - name: down
        show: on
```

Step 3. Create a Sequence Logic Block to track the progression

MPF includes a number of convenient ways for tracking progress called Logic Blocks, including the *sequence* that we can use to require a series of events to occur in a specific order.

The below sequence requires all four drop target shots to be hit, but only registers a hit if the shot is in the "lit" state. This allows us to track where we are in the sequence without having to monitor each shot individually.

The sequence also has `restart_events` so we can restart when the mode starts and when the sequence completes. All logicblocks have a default completion event called *logicblock_(name)_complete* so we don't need to explicitly define any completion event.

```
##! mode: sequential_drops

sequences:
  drop_sequence:
    restart_events: reset_drop_sequence
    events:
      - drop_1_lit_hit
      - drop_2_lit_hit
      - drop_3_lit_hit
      - drop_4_lit_hit
```

Step 4. Start, advance, and reset the shots

We will use events to manage the behavior of the shots and the drop targets. The first step is to identify all the rules of how the sequence and shots behave.

- Rule 1: When the mode starts, reset the drop sequence
- Rule 2: When the sequence is completed, reset the drop sequence

On a reset, all of the shots will be in their "off" state. We need the first target to be "lit" in order for the sequence to start.

- Rule 3: When the sequence resets, advance the first target from "off" to "lit"

When a shot is in the "off" state and gets hit, we want to fire the reset coil for the target so that the target stays up.

- Rule 4: When an "off" shot is hit, reset its coil

We can apply all of these rules based on the corresponding events, like follows.

```
##! mode: sequential_drops

event_player:
  # When the mode starts, reset the drop sequence
  mode_sequential_drops_started: reset_drop_sequence

  # When the sequence is completed, reset the drop sequence
  logicblock_drop_sequence_complete: reset_drop_sequence

  # When the sequence resets, advance the first target
  reset_drop_sequence: advance_drop_1

  # When an "off" shot is hit, reset its coil
  drop_1_off_hit: reset_drop_1
  drop_2_off_hit: reset_drop_2
  drop_3_off_hit: reset_drop_3
  drop_4_off_hit: reset_drop_4
```

The above configuration requires that each drop target coil has the corresponding reset events, as configured below.

```

drop_targets:
  drop_1:
    switch: s_drop_1
    reset_coil: c_drop_1
    reset_events: ball_starting, machine_reset_phase_3, reset_drop_1
  drop_2:
    switch: s_drop_2
    reset_coil: c_drop_2
    reset_events: ball_starting, machine_reset_phase_3, reset_drop_2
  drop_3:
    switch: s_drop_3
    reset_coil: c_drop_3
    reset_events: ball_starting, machine_reset_phase_3, reset_drop_3
  drop_4:
    switch: s_drop_4
    reset_coil: c_drop_4
    reset_events: ball_starting, machine_reset_phase_3, reset_drop_4
```

Step 5. Rewards for progression and completion

When a drop target is hit, The sequence logic block keeps track of whether it is the part of the sequence or not. We can easily award points for progression with the *logicblock_(name)_hit* event (when a lit target is hit) and the *logicblock_(name)_complete* event (when the full sequence is completed).

```
##! mode: sequential_drops

variable_player:
  logicblock_drop_sequence_hit:
    score: 1000
```

(continues on next page)

(continued from previous page)

```
  logicblock_drop_sequence_complete:
    score: 50_000
```

Full Example Code

The full code from this example can be found as a fully-working game template in the MPF Examples repository.

https://github.com/missionpinball/mpf-examples/tree/dev/cookbook/sequential_drop_banks

Related Docs

- *shots:*
- *shot_groups:*
- *shot_profiles:*
- *sequences:*
- *variable_player:*

Recipe: Skillshot (with Lane Change)

This guide shows you how to build an MPF config for a skillshot with rotating rollover lanes (a.k.a "lane change").

For a description of rollover lanes, see *Rollover Lanes with Lane Change*

Step 1. Create a skillshot mode and lane shots

Skillshots are a self-contained set of rules, so it's wise to create a separate mode that can be started when a player's ball starts and ended after the skillshot is hit (or missed).

We'll assume that the machine has switches defined in the `switches:` config section for each of the lanes, called `s_lane_left`, `s_lane_middle`, and `s_lane_right`. We'll also use corresponding lights `l_lane_left` etc. to indicate which lane is lit.

```
#config_version=5
modes:
  - skillshot_with_lane_change
switches:
  s_lane_left:
    number: 1
  s_lane_middle:
    number: 2
  s_lane_right:
    number: 3
lights:
  l_lane_left:
```

(continues on next page)

(continued from previous page)

```
    number: 1
  l_lane_middle:
    number: 2
  l_lane_right:
    number: 3
shot_profiles:
  skillshot_profile:
    states:
      - name: off
      - name: on

##! mode: skillshot_with_lane_change
```

The first thing our mode needs is *shots:*. Each lane will count as a shot, and for this example we'll have three lanes "left", "middle", and "right".

```
##! mode: skillshot_with_lane_change

mode:
  start_events: start_mode_skillshot_with_lane_change
  stop_events: stop_mode_skillshot_with_lane_change
  priority: 1000

shots:
  skillshot_left:
    advance_events: advance_skillshot_left
    profile: skillshot_profile
    switch: s_lane_left
    show_tokens:
      led: l_lane_left
  skillshot_middle:
    advance_events: advance_skillshot_middle
    profile: skillshot_profile
    switch: s_lane_middle
    show_tokens:
      led: l_lane_middle
  skillshot_right:
    advance_events: advance_skillshot_right
    profile: skillshot_profile
    switch: s_lane_right
    show_tokens:
      led: l_lane_right
```

Step 2. Creating a profile for the lanes

We can create a *shot_profile* for the lanes that starts with the light off and makes it flash if that lane is lit for the skillshot.

By default, a shot will advance its profile when the shot is hit, but we don't want that here so we'll set `advance_on_hit: false`. Instead, we have explicit `advance_events` set on the shots so we can advance them for the lane change.

```
##! mode: skillshot_with_lane_change
shot_profiles:
  skillshot_profile:
    advance_on_hit: false
    states:
      - name: off
        show: off
      - name: lit
        show: flash
```

Step 3. Creating a shot_group for the lanes

To tell MPF that the lane shots are related to each other, we create a *shot_group* with all the shots in it.

Shot groups are powerful because they control behavior of all the shots together. In this case, we'll use our shot group to rotate the lit shots.

```
##! mode: skillshot_with_lane_change

shot_groups:
  skillshot:
    shots: skillshot_left, skillshot_middle, skillshot_right
    disable_rotation_events: s_plunger_lane_inactive
    rotate_left_events: s_flipper_left_active
    rotate_right_events: s_flipper_right_active
```

The `rotate_left_events` and `rotate_right_events` trigger the lane changes based on the flipper events. The `disable_rotation_events` will prevent the player from changing lanes after they plunge the ball, for a true "skill" shot. (If you want to allow lane changes after plunge, just remove that line.)

Step 4. Light a random shot when the mode starts

The starting state of the shot profile is "off", so we need to pick one shot at random and advance it to its "lit" state. We'll use the *random_event_player:* for this.

```
##! mode: skillshot_with_lane_change
random_event_player:
  mode_skillshot_started:
    events:
      - advance_skillshot_left
      - advance_skillshot_middle
      - advance_skillshot_right
```

Step 5. Rewards for Skillshot

When the player hits the lit skillshot shot, they get an award of points. We can use the *variable_player:* for this.

When a shot in a shot group is hit, the shot group will post an event with the state name of the shot that was hit. By using the shot group events, we can check when *any* shot is hit, rather than having to check each shot in the group individually.

```
##! mode: skillshot_with_lane_change
variable_player:
  skillshot_lit_hit:
    score: 20_000
```

Step 6. Ending the mode on skillshot hit, or any other hit

After any skillshot lane is hit, the skillshot mode should end. We can again use the shot group to detect *any* shot being hit, but we'll use a hit event *without* any state because it doesn't matter whether the shot was lit or not.

We also want to end the skillshot mode if any other switch on the playfield was hit, which we can detect from the *playfield_active* event. However, when the skillshot is hit the *playfield_active* event will post before the *skillshot_lit_hit* event, so if we end the mode immediately then no score will be awarded. Instead, we add a 1 second delay after playfield activation before ending the mode.

```
##! mode: skillshot_with_lane_change
event_player:
  skillshot_hit: stop_mode_skillshot
  playfield_active: stop_mode_skillshot|1s
```

Full Example Code

The full code from this example can be found as a fully-working game template in the MPF Examples repository.

https://github.com/missionpinball/mpf-examples/tree/dev/cookbook/skillshot_with_lane_change

Related Docs

- *random_event_player:*
- *shots:*
- *shot_groups:*
- *shot_profiles:*
- *variable_player:*

Recipe: Skillshot (with Auto-Rotate)

This guide shows you how to build an MPF config for a skillshot with automatically-rotating targets. When the player's turn starts the target shots will rotate one "lit" shot rapidly, and when the ball is plunged the rotation will stop and the lit shot will flash as the skillshot target.

Step 1. Create a skillshot mode and shots

Skillshots are a self-contained set of rules, so it's wise to create a separate mode that can be started when a player's ball starts and ended after the skillshot is hit (or missed).

```
#config_version=5
modes:
  - skillshot_with_auto_rotate
switches:
  s_dropbank_1:
    number: 1
  s_dropbank_2:
    number: 2
  s_dropbank_3:
    number: 3
  s_dropbank_4:
    number: 4
  s_dropbank_5:
    number: 5
lights:
  l_dropbank_1:
    number: 1
  l_dropbank_2:
    number: 2
  l_dropbank_3:
    number: 3
  l_dropbank_4:
    number: 4
  l_dropbank_5:
    number: 5

##! mode: skillshot_with_auto_rotate
# mode will be defined below
```

The first thing our mode needs is *shots:*. Each possible target will be a shot (in this example, we'll have five). Each shot has a switch, a light, and a shot profile to track its state. The sample code below uses dropbank switches for the skillshot, but you are free to use any switches you like.

Each shot will also have unique `advance_events` configured, which will be explained in detail in section 4. What's important to note now is that the first shot includes `advance_events: mode_skillshot_started` so that this shot will automatically light when the mode starts, as the first shot in the rotation.

```
##! mode: skillshot_with_auto_rotate
mode:
  start_events: start_mode_skillshot_with_auto_rotate
  stop_events: stop_mode_skillshot_with_auto_rotate
  priority: 1000

shots:
  skillshot_drop_1:
    switch: s_dropbank_1
    advance_events: mode_skillshot_with_auto_rotate_started, advance_skillshot_1
    profile: skillshot_profile
    show_tokens:
```

(continues on next page)

(continued from previous page)

```
    leds: l_dropbank_1
  skillshot_drop_2:
    switch: s_dropbank_2
    advance_events: advance_skillshot_2
    profile: skillshot_profile
    show_tokens:
      leds: l_dropbank_2
  skillshot_drop_3:
    switch: s_dropbank_3
    advance_events: advance_skillshot_3
    profile: skillshot_profile
    show_tokens:
      leds: l_dropbank_3
  skillshot_drop_4:
    switch: s_dropbank_4
    advance_events: advance_skillshot_4
    profile: skillshot_profile
    show_tokens:
      leds: l_dropbank_4
  skillshot_drop_5:
    switch: s_dropbank_5
    advance_events: advance_skillshot_5
    profile: skillshot_profile
    show_tokens:
      leds: l_dropbank_5
```

Step 2. Create a profile for the targets

We can create a *shot_profile* for the targets that starts with the light off, lights it solid after one advancement, and makes it flash after a second advancement. By default, a shot will advance its profile when the shot is hit, but we don't want that here so we'll set `advance_on_hit: false`.

When the mode starts, all shots will be in the first profile state "off". The first shot will immediately advance to the "on" state (from the `advance_events: mode_skillshot_with_auto_rotate_started` noted above). Every time the shot group rotates, the next shot in sequence will shift to "on". This creates the rotation effect of the lit shot moving across the targets.

When the ball is plunged, whichever shot is in the "on" state will be advanced to the "lit" state and its light will flash. When any shot is hit, we'll check whether it is "lit" or not to know whether the skillshot should be awarded.

```
##! mode: skillshot_with_auto_rotate
shot_profiles:
  skillshot_profile:
    advance_on_hit: false
    states:
      - name: off
        show: off
      - name: on
        show: on
      - name: lit
        show: flash
```

Step 3. Create a shot_group for the lanes, and a rotation timer

To tell MPF that the five shots are related to each other, we create a *shot_group* with all the shots in it.

Shot groups are powerful because they control behavior of all the shots together. In this case, we'll use our shot group control the rotation of the shots, and a *timer* to trigger a rotation every half-second.

```
##! mode: skillshot_with_auto_rotate

shot_groups:
  skillshot:
    shots:
      - skillshot_drop_1
      - skillshot_drop_2
      - skillshot_drop_3
      - skillshot_drop_4
      - skillshot_drop_5
    rotate_events: timer_skillshot_rotate_tick

timers:
  skillshot_rotate:
    tick_interval: 500ms
    start_running: true
    control_events:
      - event: s_plunger_lane_inactive
        action: stop
```

The `rotate_events` will move the state of the shots each time the timer ticks, and the ball leaving the plunger lane will stop the timer and thus stop the rotation.

Step 4. Flash the lit shot when the rotation stops

When the timer stops, one of the shots will be in the "on" state. Whichever shot this is should be advanced to the "lit" state so the light is flashing, and we can use conditional events to listen for the timer stop and advance *only* the lit shot.

Shot profile states are numbered starting with zero, so our "off" state is number 0 and the "on" state is number 1. The following code will only post the advance event for a shot if that shot is in state number 1, a.k.a. "on".

```
##! mode: skillshot_with_auto_rotate
event_player:
  timer_skillshot_rotate_stopped:
    - advance_skillshot_1{device.shots.skillshot_drop_1.state==1}
    - advance_skillshot_2{device.shots.skillshot_drop_2.state==1}
    - advance_skillshot_3{device.shots.skillshot_drop_3.state==1}
    - advance_skillshot_4{device.shots.skillshot_drop_4.state==1}
    - advance_skillshot_5{device.shots.skillshot_drop_5.state==1}
```

Each shot configured in step 1 above has `advance_events` that correspond to its shot number, so the above event player will trigger the correct shot to advance to its "lit" state.

Step 5. Rewards for skillshot

When the player hits the lit shot, they get an award of points. We can use the *variable_player:* for this.

When a shot in a shot group is hit, the shot group will post an event with the state name of the shot that was hit. This way, we can check when *any* shot is hit rather than having to check each shot individually.

```
##! mode: skillshot_with_auto_rotate
variable_player:
  skillshot_lit_hit:
    score: 20_000
```

Step 6. End the mode on skillshot hit, or any other hit

After any skillshot shot is hit, the skillshot mode should end. We can again use the shot group to detect *any* shot being hit, but we'll use a hit event *without* a state name because it doesn't matter whether the shot was lit or not.

We also want to end the skillshot mode if any other switch on the playfield was hit, which we can detect from the *playfield_active* event. However, when the skillshot is hit the *playfield_active* event will post before the *skillshot_lit_hit* event, so if we end the mode immediately then no score will be awarded. Instead, we add a 1 second delay after playfield activation before ending the mode.

```
##! mode: skillshot_with_auto_rotate
event_player:
  # Add these lines after timer_skillshot_rotate_stopped (defined above)
  skillshot_hit: stop_mode_skillshot
  playfield_active: stop_mode_skillshot|1s
```

Full Example Code

The full code from this example can be found as a fully-working game template in the MPF Examples repository.

https://github.com/missionpinball/mpf-examples/tree/dev/cookbook/skillshot_with_auto_rotate

Related Docs

- *shots:*
- *shot_groups:*
- *shot_profiles:*
- *timers:*
- *variable_player:*

CHAPTER 25

Config file reference

This section contains details about every possible entry you can use in your YAML config files. Each entry also has information about whether it's valid in your machine-wide config, a mode-specific config, or both.

Instructions

As you dig into the specific settings for individual config sections, it's important to understand how various settings mentioned in the reference are used:

Config file instructions

This section contains some general formatting guidelines.

Understanding the #config_version setting

Since MPF is mainly "programmed" with YAML-based config files, we need a way for MPF to know that the config file(s) it's loading are compatible with the version of MPF that's running.

This is specified in the very first line of a config file (in both the machine-wide configs and mode config files). You specify the config version with a list that starts with a hash sign, like this:

```
#config_version=5
```

In YAML, lines that start with # are ignored, which means the YAML processor skips this line, but MPF uses it to make sure the config file it's trying to load will work with that version of MPF.

Not every new version of MPF changes the config_version number. If we release a new version of MPF that does not have a new config_version number, then you can use the new version of MPF without needing to make any changes to your config files.

Updating your config files to the latest version

MPF includes a config file migration tool that can automatically migrate your config files to the latest version.

Which versions of MPF require which config_versions?

- MPF 0.50+: `#config_version=5`
- MPF 0.30-0.33: `#config_version=4`
- MPF 0.20-0.21: `#config_version=3`
- MPF 0.19: `#config_version=2`
- MPF 0.17-0.18: `#config_version=1`
- MPF 0.1-0.16: config_version not used (a.k.a. config version 0)

Machine config files

Some config sections can only be used in your machine-wide configs. Others can only exist in *mode_config* or some in either of them. Those are usually hardware related and/or exist outside of the game. You can see if this is the case at the top of the relevant *config* section. For instance *lights* are defined machine-wide because they are used throughout the whole lifecycle of a machine.

Mode config files

Modes usually start with a `mode:` section (see *mode*) which defines their priority and when they start or stop:

```
##! mode: mode1
mode:
  start_events: ball_starting
  stop_events: timer_mode_timer_complete, shot_right_ramp
  priority: 300
```

Not all config sections can be used in your machine-wide config (see *machine_config*). Some devices may only exist in modes (usually if they require an active game with one or more players). You can see if this is the case at the top of the relevant *config* section. For instance *extra_balls* are player-bound and can only be used in modes.

Using dynamic runtime values in config files

MPF config files can contain values in the form of links to dynamic placeholders which are evaluated live when MPF is running rather than being hard-coded into a config file.

Dynamic values can come from several sources, including player variables, machine variables, operator settings, properties of devices, etc. (Read on for a full list.)

For example, you might want to have a shot called “jackpot” that scores a multiplier which is the number of shots made times 100k points.

Without dynamic values, your variable_player (scoring) section would be static, like this:

```
##! mode: mode1
variable_player:
  shot_jackpot_hit:
    score: 100000
```

But let's say you have a player variable called "troll_hits" which holds the number of trolls hit that you want to multiply by 100,000 when the shot is made. You can use the "current_player" dynamic value in your variable_player config like this:

```
##! mode: mode1
variable_player:
  shot_jackpot_hit:
    score: current_player.troll_hits * 100000
```

You can access other values dynamically as well, such as a timer ticking away a hurry-up or a counter to track how many times a multiplier switch has been hit

```
##! mode: mode1
variable_player:
  collect_hurryup:
    score: 1000 * device.timers.hurryup_clock.ticks_remaining * device.counters.hurryup_multiplier.value
```

Another example might be operator settings. Rather than hard coding tilt warnings to 3, you might want to like the operator choose the tilt warnings.

So instead of this:

```
##! mode: tilt
# in your tilt mode
tilt:
  warnings_to_tilt: 3
```

You would have this instead:

```
# in your machine config
settings:
  warnings_to_tilt:
    label: Number of tilt warnings
    values:
      0: "no warnings"
      1: "1"
      2: "2"
      3: "3"
      5: "5"
      10: "10"
    default: 3
    key_type: int
    sort: 600
##! mode: tilt
# in your tilt mode
tilt:
  warnings_to_tilt: settings.warnings_to_tilt
```

Note the example above requires that you have a `settings:` section in your machine config and that you've defined a setting called "tilt_warnings". See *Tilt* for more details.

You can also use dynamic values in *conditional events*.

Types of dynamic values

You can use the following types of placeholders.

Current Player Variables

You can access a player variable `X` of the current player using `current_player.X`. For instance, `current_player.my_player_var` will access `my_player_var` of the current player. This placeholder is only available when a game is active.

Common player variables are:

- `current_player.score` - Score of the current player
- `current_player.ball` - Current ball

Player Variables of Specific Player

You can access a player variable `X` of a specific player `P` using `players[P].X`. `P` starts at 0. So player 1 will be `players[0].P`. For instance, `players[1].my_player_var` will access `my_player_var` for player 2. `players[0].my_player_var` will access player 1. This placeholder is only available when a game is active.

Common player variables are:

- `players[0].score` - Score of player 1
- `players[1].score` - Score of player 2
- `players[2].score` - Score of player 3
- `players[3].score` - Score of player 4

Game Variables

You can access game variable `X` using `game.X`. This placeholder is only available when a game is active.

Common game variables are:

- `game.max_players` - Maximum players currently allowed
- `game.num_players` - Number of players in game
- `game.balls_per_game` - Balls per game
- `game.balls_in_play` - Balls in play
- `game.tilted` - True if the game has been tilted
- `game.slam_tilted` - True if the game has been slam tilted

Additionally, a game has all common mode variables (see below). `game.X` is just a convenient way to access `mode.game.X`.

Machine Variables

You can access machine variable `X` using `machine.X`.

Common machine variables are:

- `machine.player1_score` - Player 1 score from the last game
- `machine.player2_score` - Player 2 score from the last game
- `machine.player3_score` - Player 3 score from the last game
- `machine.player4_score` - Player 4 score from the last game
- `machine.credits_string` - String for credits or freeplay
- `machine.credits_value` - Human readable credits string

Settings

You can access setting `X` using `settings.X`.

Devices

You can access property `X` of device `D` of type `T` using `device.T.D.X`. For instance you can access the value of counter `my_counter` using `device.counters.my_counter.value`.

Common device properties are:

- `device.counters.my_counter.value`
- `device.counters.my_counter.enabled`
- `device.flippers.left_flipper.enabled`
- `device.playfields.playfield.balls`
- `device.ball_devices.my_lock.balls`
- `device.counters.superjets_counter.value`
- `device.accruals.magic_tokens.enabled`
- `device.sequences.world_tour.completed`

MPF uses consistent names across devices, so for example any device that tracks a number will have a `value` property and any device that can be enabled/disabled will have an `enabled` property. The full list of properties available for a specific device are listed in the "Monitorable Properties" section of that device's documentation page.

Modes

You can access property `X` of mode `M` using `mode.M.X`.

Common mode properties are:

- `mode.my_mode.active`

Using if/else logic with dynamic values

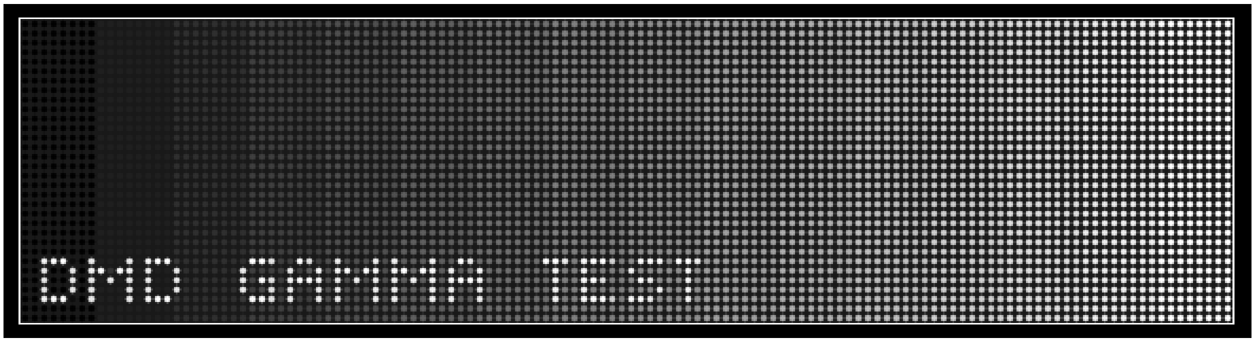
```
##! mode: mode1
counters:
  my_counter:
    count_events: count_up
    count_complete_value: 5 if player.wizard_complete else 3
```

Gamma correction in MPF

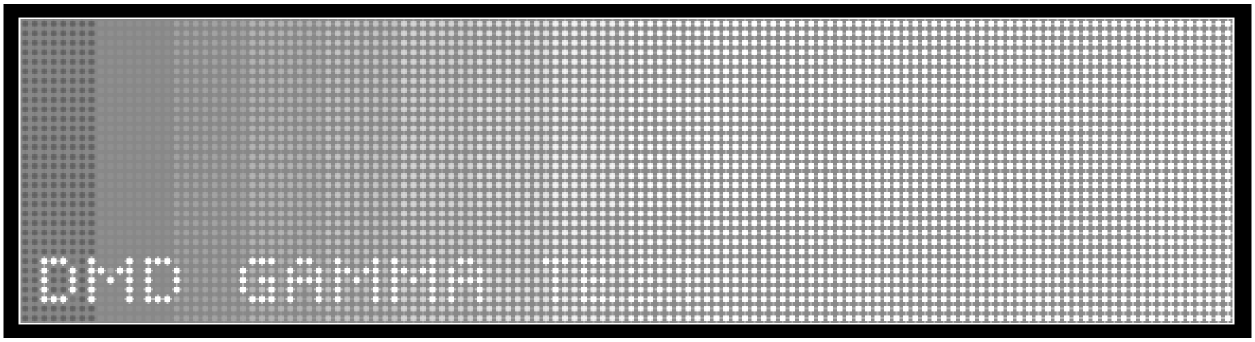
MPF includes functionality to allow you to adjust the gamma of the color information that is sent to physical DMDs (RGB and mono) and to RGB LEDs. (You don't need to set the gamma of an LCD display since that's handled by your OS.)

You can read full details in the Gamma correction article on Wikipedia, but the quick explanation is that the human eye doesn't not perceive a change in brightness at the same ratio that an LED sets its brightness.

When you're setting colors in MPF, you expect that 100% brightness looks fully bright, and that 50% looks like 50%, etc. Here is a screenshot of a slide which has 16 bars which fade from off to fully white, in a more-or-less even fashion:



However if you show this slide on your physical DMD with no gamma correction, it looks something like this:



Even though the individual pixels are showing their "correct" brightness, the human eye can't really tell a different between 50% and 100%, and pretty much everything on the right half of the DMD looks fully white.

So you can adjust this by setting the gamma value. By default, MPF uses a gamma value of 2.5 for RGB LEDs, and 2.2 for RGB DMDs. (It also uses a value of 1.0 for mono DMDs since some of the hardware controllers do their own internal gamma correction, though others don't, so you might have to change them.

We recommend you read the documentation for the *dmds:*, *rgb_dmds:*, and *light_settings:* (for LEDs) to set the proper gamma.

Tuning your DMD gamma

MPF includes a built-in gamma test slide (the one used in the images above) which you can use to dial-in your gamma setting.

The easiest way to show this slide on your physical DMD is to make a temporary addition to your machine config to add a slide player, like this:

```
slide_player:
  mode_attract_started:
    dmd_gamma_test:
      priority: 10000000
```

This will just show the gamma test slide at a crazy high priority so it shows on top of everything else. (Remember if your DMD is not your default display, you'll also have to add `target: dmd` or whatever you use to target slides to your DMD.)

Now you can play with different gamma settings for your DMD in either your `dmds:` or `rgb_dmds:` section. (Note you'll have to restart MPF after each change you make.)

Note that you might also have to adjust `brightness:` along with `gamma:`. For example, some people had to set the brightness of their RGB DMDs to a super low value, like `0.1` or `0.2` before MPF had gamma control, but with proper gamma settings, you can probably take your brightness up to somewhere around `0.5`.

We like to use the gamma test slide and set the brightness first based on the right-most brightest block, and then once that's set, we start messing with the gamma. It will probably be some trial-and-error, but once it's dialed in it's a "set it and forget it" type of thing.

How to enter gain values in config files

The sound-related items in your config files contain various volume settings that may be specified as a gain value. MPF gives you the flexibility to specify gain values as simple numeric values between 0.0 and 1.0 or as a decibel string between -inf and 0.0 db. Individuals with audio or video editing experience may be more comfortable working with decibel values.

Entering a simple numeric gain value

To enter a simple numeric gain value, simply enter a number between 0.0 and 1.0 with no appended label string. Some examples:

```
volume: 0.1334

volume: 1.0

volume: 0.0
```

Entering a gain value in decibels

To enter a gain value in decibels, enter your value between -inf and 0.0 and add a "db" after your value. (This can be uppercase or lowercase, and you can put a space in between your value and the letters if you want.)

Note: -inf indicates the minimum gain value (equivalent to 0.0 in a simple numeric gain value) and should not contain a "db" suffix. For all other decibel values if you do not enter the "db" suffix after your value, then MPF will read in the gain value as a simple numeric gain value between 0.0 and 1.0.

Some examples:

```
volume: -17.5db

volume: 0.0 db

volume: -inf
```

It makes no difference whether you enter your gain values in simple numeric format or decibels, as MPF will convert everything to simple gain values under-the-hood when it reads in your configuration files.

How to enter time strings in config files

You machine configuration files are full of settings which require time values to be entered, such as "10 seconds" or "250 milliseconds."

Rather than arbitrarily decide which values should be entered as seconds versus milliseconds, we've built MPF so that you can enter either one whenever a time entry is needed which MPF will internally convert to the proper value.

These time values are used all over the place. (Ball device count delays, ball save time, ball search settings, reset delays, slide expiration times, etc.)

We'll use an example from a ball device for the `ball_count_delay:` setting. (Again, this is just an example. You use these same options whenever you need to enter a time value):

Entering a time duration in seconds

To enter a time duration in seconds, simply add an "s" or "sec" after your number. (This can be uppercase or lowercase and you can put a space in between your number and the letters if you want.) Some examples:

```
ball_count_delay: 0.5s

ball_count_delay: 0.5 S

ball_count_delay: 0.5sec
```

Entering a time duration in milliseconds

To enter a time duration in seconds, simply add an "ms" or "msec" after your number. (This can be uppercase or lowercase, and you can put a space in between your number and the letters if you want.)

Note that if you do not enter and letters, then MPF will read in the time duration in whatever the default scale is for that particular setting. (The instructions for each setting should say whether the default is seconds or ms.

Some examples:

```
ball_count_delay: 500ms

ball_count_delay: 500 MS

ball_count_delay: 500msec

ball_count_delay: 500
```

It makes no difference whether you enter your time durations as seconds or milliseconds, as MPF will convert everything to milliseconds (since that's the default for *ball_count_delay* when it reads in your configuration files.

Entering a time duration in minutes, hours, or days

You can also enter time strings in MPF for time periods longer than seconds or milliseconds. While this isn't practical for things like ball device delays, it's used in certain modules (like the credits module) for some settings.

Some examples:

```
credit_expiration_time: 2m     # 2 minutes

credit_expiration_time: 2h     # 2 hours

credit_expiration_time: 2d     # 2 days
```

Case insensitivity in config files

Setting names config files are case sensitive (starting with 0.50). In 0.17 to 0.33 settings were case insensitive but it caused many problems and thus has been dropped. Generally, a safe approach is to use only lower case in config files, though, it is fine to use upper case in slides. Using lower case is recommended for naming modes, devices, timers, etc. . . .

Device Control Events

Many devices in MPF have configuration options which lets them be controlled via events. (These are called "device control events".) For example, flippers and autofire coils have *enable_events* and *disable_events*, shots have *enable_events*, *disable_events*, and *reset_events*, shot groups have *enable_events*, *disable_events*, *reset_events*, *rotate_right_events*, and *rotate_left_events*, etc.

You can specify these events in each device's settings on a machine- wide basis in your machine config, and you can also specify these events that are only active when a mode is active in your mode config files. There are several options for how you specify these device control events, depending on what you want to do.

If you have just one event

Even though these configuration entries use the word "events" (plural), you can configure them for just one event. For example, if you have a flipper device that you want to enable when a ball starts, you can add the following line to the configuration for your flipper:

```
enable_events: ball_started
```

If you have multiple events

If you want one of these actions to be performed based on any one of multiple events, you can enter multiple events. For example, maybe you want to disable a flipper when the ball ends, but you also want to make sure it's disabled when a tilt or slam tilt event is posted. In that case you'd enter your configuration like this:

```
disable_events: ball_ending, tilt, slam_tilt
```

Note that in this case, the flipper will disable if *any* of these events is posted. If you want to get fancy and require that multiple events need to be posted before you disable your flipper, then you would use an Accrual or Sequence Logic Block to track those events, and then you'd add a new event to your `events_when_complete:` in that Logic Block and then enter that same event in the `disable_events:` for your flipper.

Note that when you're entering multiple events, you can enter them all on the same line separated by commas, or you can enter each one on its own line started with a dash and a space, like this:

```
disable_events:
    - ball_ending
    - tilt
    - slam_tilt
```

It makes no difference to MPF, rather this is just a personal preference for how you want your config files to look.

If you want to configure "delays" before performing your action

You can also enter delays (in either seconds or milliseconds) which cause the enable, disable, or reset events to wait after one of your events is fired. Here's an example from the "Solids" drop target bank in Big Shot:

```
reset_events:
    ball_starting: 0
    collect_special: .75s
```

In this case when the *ball_starting* event is posted, MPF will reset the drop target group immediately (no delay, due to the "0" value), and when the *collect_special* event is posted, MPF will wait 0.75 seconds before resetting it. (So you see that different events can have different delays.) In case you're wondering why we did this, take a look at the reset_events configuration for the other bank of drop targets (called "Stripes") in Big Shot:

```
reset_events:
    ball_starting: 0.25s
    collect_special: 1s
```

If you look at these two sets of configurations together, you see that when the *ball_starting* event is posted, MPF will reset the Solids drop target bank immediately and then wait a quarter of a second before resetting the Stripes drop target bank. We did this so that the reset emulates the original characteristics of resetting one then the other in succession, rather than resetting them both at the same time.

Also note that we have a similar quarter-second delay between the two drop target banks when we reset them after the special is collected, but in this case we reset them after 0.75 and 1 second. That's because that collecting the special awards a replay which fires the knocker, but if the knocker fires at the same time as the drop targets are reset then the player can't hear the knocker since the drop target reset coils in Big Shot are so massive. So when the special is collected, we fire the knocker immediate, then 0.75 seconds later we reset the Solids drop target bank, then 0.25 seconds after that we reset the Stripes drop target bank.

You can enter these delay times in either seconds or milliseconds, as outlined *here*. All this is done via the config files with no custom Python code needed! :)

Messing with priorities

By default the handler will have the priority of your mode or 1 if its ouside of a mode. In addition, some devices increase the priority of some handlers over others. For instance, disable is handled before enabled (in case you are using both on the same event). Normally, this is just fine and you do not have to worry. However, there are cases where you want to increase the priority of a certain handler. You add `.x` to your event to increase the priority by `x`.

In the following example, we ensure that the device will first enable, then score and finally disable:

```
enable_events: ball_started.3
score_events: ball_started.2
disable_events: ball_started.1
```

Without explicit priorities (or some logic in the device) the order of the three handlers would be random and you might see the following entry in the log:

```
Duplicate handler for class MyDevice on event ball_started with priority 1. Handlers: x
```

You can read more about *event handler priorities*

Overwriting config files

Help us to write it

Specifying Colors in Config Files

Colors in config files can be specified by name (like "red") or by hex value ("ff0000").

You can see a list of valid color names (and their respective colors) here.

In addition to the 140 standard named colors, MPF adds the following color options:

- `on` - turns on an LED with that LED's `default_color:` setting. (Default is "white" if you don't specify a color.)
- `off` - maps (0,0,0) which is more intuitive than "black" when you're working with LEDs.
- **`stop` - removes the current color being displayed allowing a color from a lower priority light_player or** show_player to become visible.

You can also specify color by hex string. If you do this, do *NOT* put a # in it, since YAML files use those for comments which are ignored.

- CORRECT: `color: ff0000`
- WRONG: `color: #ff0000`

Specifying opacity / alpha

For colors which will be processed by the media controller (such as slide background and widget colors), you can optionally add two more characters to a hex color to specify the alpha value.

For example:

- `ff0000ff` (fully opaque)
- `ff000080` (50% opacity)

See the *Widget Opacity & Transparency* documentation for details.

Understanding tags

General Theory

A common definition of a tag is "a label attached to someone or something for the purpose of identification or to give other information". This sums up the whole idea behind tags in MPF. You can add one or more tags on to the various parts of your game. These tag identifiers can then be used in various ways such as firing events or identifying a device in some particular way.

Tags and Events

Some tags will cause events to be generated. An example of this is a switch device. You can tag a switch device with one or more tags.

```
switches:
  mygame_switch_button_start:
    number: 1
    tags: start, skyfall
```

In this case, whenever the start switch is activated, there will be two events fired. You will see something like this in the log:

```
2018-09-26 20:32:14,215 : INFO : EventManager : Event: ======'sw_start'====== Args={}
2018-09-26 20:32:14,215 : INFO : EventManager : Event: ======'sw_start_active'====== Args={}
2018-09-26 20:32:14,215 : INFO : EventManager : Event: ======'sw_skyfall'====== Args={}
2018-09-26 20:32:14,215 : INFO : EventManager : Event: ======'sw_skyfall_active'====== Args={}
```

Both events are prefixed with `sw_` as a default. You can override this with the *mpf:* section.

Note: Please note that those events will only show up if either a handler for them exists (i.e. an event_player) or when you set `debug: True` to your switch. This is purely a performance optimization and also will safe you a lot of log lines.

Power of Tags

While tags and events can be used interchangeably at times, the real power lies in multiple tagging. When you use the same tags on multiple devices it can save you coding time and reduce the size of your configurations.

Example 1 - Pop Bumpers

For this example, a game with 3 popbumpers will all behave in the same way. To start we will give 100 points for every hit of a pop bumper.

Firstly we define the popbumper switches.

```
switches:
  mygame_popbumper_left:
    number: 55
    tags: mygame_popbumper
  mygame_popbumper_top:
    number: 56
    tags: mygame_popbumper
  mygame_popbumper_right:
    number: 57
    tags: mygame_popbumper
```

Now we want to score 100 points every time a pop bumper is hit. We have two ways of accomplishing this same goal. One with pure events and one with tags.

Example with events:

```
##! mode: my_mode
variable_player:
  mygame_popbumper_left_active:
    score: 100
  mygame_popbumper_top_active:
    score: 100
  mygame_popbumper_right_active:
    score: 100
```

Now with tags:

```
##! mode: my_mode
variable_player:
  sw_mygame_popbumper:
    score: 100
```

As you can see, if you have a repeating event you can save yourself some time and coding by using tags. Any switch tagged as *mygame_popbumper* will echo a *sw_mygame_popbumper* event.

Example 2 - Playfield is active

Another example is tagging specific switches on a playfield to validate if a ball is in play or not. These would be any switches a ball could hit within regular game play which are not part of a device. Some devices such as drop targets will trigger their own switch during ball search and we do not want them to end ball search doing that. Therefore, they got built-in support for marking the playfield active and your should not tag those switches (MPF will also complain if you do).

For our purposes we will check if a ball hits the roll over in the orbit after it was plunged. At that point it is obviously on the playfield and ball search should not start.

All we need to do is add a tag:

```
switches:
  mygame_orbit_l:
    number: 55
    tags: playfield_active
  mygame_orbit_r:
    number: 56
    tags: playfield_active
```

Reserved Tags in MPF

MPF contains some reserved tags that are used for certain devices. An example of this is a ball trough.

```
ball_devices:
  mygame_balldevice_trough:
    ball_switches: mygame_switch_trough_1, mygame_switch_trough_2, mygame_switch_trough_3
    eject_coil: mygame_coil_trough_eject
    eject_targets: mygame_balldevice_shooter_lane
    tags: trough, home
```

The two tags on the ball trough device assist MPF in determining various characteristics of this device. Namely that it is considered a 'home' device where balls can come to rest when a game is not in play. And the 'trough' tag to help MPF denote that this is a ball trough and not some other style of captive device like a saucer.

Understanding the debug: setting

Almost every device and platform in MPF contains a `debug` setting. If you set this to `True` MPF will generate more log output for this device. This may greatly increase the log size and decrease performance. Enable it if you got problems with a certain device or platform to debug problems later on.

Config player "express" configs

Todo: Need to add this

How to add lists to config files

Throughout the Mission Pinball Framework config files, there are several places where the configuration items need to be a "list" or a "list of lists." The MPF config files are in a YAML format, so you add list items by following the YAML spec, but it can be a kind of confusing. So this page is our "how to" guide for the various ways you can add list items to MPF config files. First of all, there are several different places we need lists. For example, device tags, logic block events, switches that make up shots, etc. For our explanation, we'll use a generic list item with generic configurations. Some examples:

```
coils:
  flipperLeft:
    number: SD18
    tags: flipper, player   # this is a list
```

```
##! mode: mode1
shots:
  outlane:
    switch: leftOutlane, rightOutlane    #this is a list
```

```
auditor:
  save_events:   # This config wants a list
    - game_started    # This is the first list item
    - ball_ended    # This is the second list item
    - game_ended    # This is the third list item
```

```
accruals:
  my_accrual:
    events:
      - sw_eightball     # this is the first list item
      - drop_targets_Solids_lit_complete, drop_targets_Stripes_lit_complete     # 2nd list item, which itself has two items
```

Valid options for lists

Ok, so let's say you have a config item that needs a list. We'll use a made-up config called "config" with three list items: item1, item2, and item3. You can enter this into your config file in one of several ways. First, you can enter all the items on one line separated by commas:

```
config: item1, item2, item3
```

Second, you can enter each item on its own line, indented, with each line starting with a dash, like this: (Be sure to include the space after the dash before the list item. It's a YAML thing.)

```
config:
    - item1
```

(continues on next page)

(continued from previous page)

```
    - item2
    - item3
```

So you have two options. Which one should you pick? It really doesn't matter. You can use whichever one has the style you prefer and whichever one makes your config files easiest to read. (We tend to just use commas, but if it's a long list then we'll put each item on its own line so the line doesn't wrap.)

Valid options for "lists of lists"

Some config items require "lists of lists" where there is a list with multiple items, and then each of those items is itself another list which may have multiple items. (This is seen a lot in MPF's Logic Blocks where we have multiple steps that can each be made up of one or more events.) The easiest way to enter these into your configuration files is to combine the method using commas and dashes, like this:

```
config:
    - item1, item2
    - item3, item4, item5
    - item6
```

How to create and understand YAML files

Indentation

any number is fine

2, 4, 3, 17, whatever

only key is that things at the same level are indented all the same

any increase in indent indicates that line is a subsection of the line above it

Dashes

Colons

Quotes

Text Templates

Text templates can contain python format strings to show *text placeholder*.

This is an example which will show the player 1 score of the previous game as number:

```
Player 1 score: {machine.player1_score:d}
```

Current score (during a game only):

```
Score {current_player.score:d}
```

Any *variable* needs to be enclosed in `{}`. Either you can use `{variable}` or `{variable:format_string}`. Any python format string will work here.

Common format strings

Assuming `variable` has a value of 1337.

Alignment and Padding

Left aligned and padded to 10 characters:

```
{variable:10}
```

Output:

```
"1337      "
```

Right aligned and padded to 10 characters with zeros:

```
{variable:0>10}
```

Output:

```
"0000001337"
```

Centered and padded to 10 characters with spaces:

```
{variable:^10}
```

Output:

```
"   1337   "
```

Number as float (2 decimals):

```
{variable:5.2f}
```

Output:

```
" 1337.00"
```

Number as integer:

```
{variable:5d}
```

Output:

```
" 1337"
```

Truncating long strings

Centered and padded to 10 characters with spaces:

```
{variable:.3}
```

Output:

```
"133"
```

Index of config sections

Here's a list of every single config section from both MPF and the MPF-MC. Some of these are valid only in machine-wide configs, and others only work in mode config files. (And some are valid in both.) The detail page for each setting indicated which type of config file it's valid in.

accelerometers:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

Hardware platforms which support accelerometers
P3-Roc
MMA8451-based I2C accelerometers

The `accelerometers:` section of your config is where you configure accelerometers, including how many G forces trigger different events.

Like other hardware devices, you create a sub-entry for each accelerometer, then under there you configure additional settings. For example:

```
accelerometers:
  test_accelerometer:
    number: 1
    level_x: 0
    level_y: 0
    level_z: 1
    hit_limits:
      0.5: event_hit1
      1.5: event_hit2
    level_limits:
      2: event_level1
      5: event_level2
```

Required settings

The following sections are required in the `accelerometers:` section of your config:

number:

Single value, type: `string`. Defaults to empty.

Number of this device in your hardware platform. The actual meaning of this number depends on your hardware platform.

Optional settings

The following sections are optional in the `accelerometers:` section of your config. (If you don't include them, the default will be used).

alpha:

Single value, type: `number` (will be converted to floating point). Default: `0.8`

The smoothing factor for single exponential smoothing (aka sliding window).

hit_limits:

One or more sub-entries. Each in the format of `number` (will be converted to floating point) : `string`

Events which are posted at a certain G-force/acceleration. You can specify multiple limits. You might use those to trigger *tilt* warnings.

level_limits:

One or more sub-entries. Each in the format of `number` (will be converted to floating point) : `string`

How much degree may the level be off? You can define multiple limits and which event should be posted when it exceeded.

level_x:

Single value, type: `integer`. Default: `0`

`level_x`, `level_y` and``level_z`` define the default axis which is considered as levelled. Defaults to `(0, 0, 1)` which means that the board is laying straight on the ground. If you mount it in the cab you want about 3 degree. Under the playfield you want 6-7 degree.

level_y:

Single value, type: `integer`. Default: `0`

`level_x`, `level_y` and``level_z`` define the default axis which is considered as levelled. Defaults to `(0, 0, 1)` which means that the board is laying straight on the ground. If you mount it in the cab you want about 3 degree. Under the playfield you want 6-7 degree.

level_z:

Single value, type: `integer`. Default: `1`

`level_x`, `level_y` and``level_z`` define the default axis which is considered as levelled. Defaults to `(0, 0, 1)` which means that the board is laying straight on the ground. If you mount it in the cab you want about 3 degree. Under the playfield you want 6-7 degree.

platform:

Single value, type: `string`. Defaults to empty.

Name of the platform this accelerometer is connected to. The default value of `None` means the default hardware platform will be used. You only need to change this if you have multiple different hardware platforms in use and this coil is not connected to the default platform.

See the *Mixing-and-Matching hardware platforms* guide for details.

platform_settings:

One or more sub-entries. Each in the format of `string` : `string`

The platform-specific hardware settings of this accelerometer.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Todo: *Help us to write it*

Related How To guides

- *Accelerometers*

accruals:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The structure of accrual logic blocks are like this:

```
accruals:
   the_name_of_this_logic_block:
      <settings>
   some_other_logic_block:
      <settings>
   a_third_logic_block:
      <settings>
```

Note that the actual name of the logic block doesn't really matter. Mainly they're used in the logs.

Required settings

The following sections are required in the `accruals:` section of your config:

events:

List of one (or more) events. The device will add handlers for those events. Defaults to empty.

The events section of an accrual logic block is where you define the events this logic block will watch for in order to make progress towards completion.

The real power of logic blocks is that you can enter more than one event for each step, and *only one* of the of the events of that step has to happen for that step to be complete.

Another way to look at it is that there's an *AND* between all the steps. For the Accrual to complete, you need Step 1 *AND* Step 2 *AND* Step 3. But since you can enter more than one event for each step, you could think of those like *OR*s. So you have Step 1 (event1 *OR* event2) *AND* Step 2 (event3) *AND* Step 3 (event4 *OR* event5), like this:

```
accruals:
  my_accrual:
    events:
      - event1, event2
      - event3
      - event4, event5
```

It might seem kind of confusing at first, but you can build this up bit-by-bit and figure them out as you go along.

You can enter anything you want for your events, whether it's one of MPF's built-in events or a made-up event that another logic block posts when it completes. (This is how you chain multiple logic blocks together to form complex logic.)

For example:

```
accruals:
  logic_block_1:
    events:
      - event1
      - event2
      - event3
      - event4
      - event5
    events_when_complete: logic_block_1_done
  logic_block_2:
    events:
      - event1, event2, event3
      - event4
      - event5
    events_when_complete: logic_block_2_done
```

In the example above, there are two logic blocks. The first one just has five steps that need to complete (in any order since we're dealing with accrual logic blocks), and each step only has one event that will mark is as complete. So basically any of those five events 1-5 can be posted in any order, and then *logic_block_1_done* will be posted.

In the second example, if event 1, 2, or 3 is posted, that will count for step 1, and then both events 4 and 5 need to be posted for steps 2 and 3. (Again, in any order.)

So in the second one, you could get event4, event2, then event5 posted, for example, and that will lead to *logic_block_2_done* being posted.

Note that you can have two logic blocks with the same events at the same time, and MPF will track the state of each logic block separately. So in the above config with those two logic blocks, if the events were posted in the order event2, event3, event4, then event5, that would complete logic block 2. Then later if event1 was posted, that would complete logic block 1.

Optional settings

The following sections are optional in the `accruals:` section of your config. (If you don't include them, the default will be used).

advance_random_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

The advance_random_events section of an accrual logic block is where you define the event or events that this logic block will watch for in order to randomly complete one of the steps in the logic block. As stated above, while there can be multiple steps that could complete this step of the logic block, this will act as one of the events as well that will complete the given step.

This will not update any lights that are associated with the events that are required to complete this step. For example, if you have a shot that could complete this step, and the step is completed by this event, the light will still remain on even though it will not progress this logic block any further. To update the lights you will want to add a hit_event to the underlying shot. This should be the event from the log logicblock_YOUR_ACCRUAL_hit {step == X} where YOUR_ACCRUAL is the name of your accrual, and X is the value of the step to which this shot is tied, which begins with 0.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Currently unused.

Optional settings

The following sections are optional in the `logic_blocks_common:` section of your config. (If you don't include them, the default will be used).

disable_events:

List of one (or more) device control events (*Instructions for entering device control events*).

Event(s) that will disable this logic block.

A logic block must be enabled to track hits, progress, and to post events.

disable_on_complete:

Single value, type: `boolean` (`true`/`false`). Default: `true`

True/False (or Yes/No) which controls whether this logic block disables itself once it completes. This does not reset the current value.

enable_events:

List of one (or more) device control events (*Instructions for entering device control events*).

Event(s) that will enable this logic block.

A logic block must be enabled to track hits, progress, and to post events.

If you don't have any enable_events listed, then the logic block will automatically be enabled when the player's ball starts.

events_when_complete:

List of one (or more) events.

Events that will be posted when this device is completed.

events_when_hit:

List of one (or more) events.

Events that will be posted when this device is hit or advanced.

persist_state:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Boolean setting (yes/no or true/false) which controls whether this logic block remembers where it was from ball-to-ball. If `False`, then this logic block will reset itself whenever a new ball starts. If `True`, then this logic block will be saved to the player variable *<logic_block_name>_state*.

Note that logic block state is reset on mode end when this is `False` and, as normal modes stop at the end of a ball, the state is always maintained on a per-player basis, regardless of what this setting is configured for.

reset_events:

List of one (or more) device control events (*Instructions for entering device control events*).

Event(s) that will reset this logic block back to its original value. This has no effect on the enabled/disabled state of the block.

Note that there are also `reset_on_complete:` and `persist_state:` settings which also affect how and when the logic block is reset.

You can reset a logic block regardless of whether it's enabled.

reset_on_complete:

Single value, type: `boolean` (`true`/`false`). Default: `true`

True/False (or Yes/No) which controls whether this logic block resets itself once it completes. This just resets the current value or progress. It does not change the enabled or disabled state.

Note, `disable_on_complete` default is `true`, which may seem like reset isn't working. For something like a counter that automatically starts again change `disable_on_complete` to `false`.

restart_events:

List of one (or more) device control events (*Instructions for entering device control events*).

List of one (or more) events which, when posted, will restart this logic block. A restart is a reset, then an enable, combined into a single action.

start_enabled:

Single value, type: `boolean` (`true`/`false`).

If `true` this device will start enabled. If `false` this device will start disabled. If you omit this the device will start enabled unless you specify `enable_events` in which case the device will start disabled.

Related How To guides

- *Accrual Logic Blocks*
- *Integrating Logic_Blocks and Shows*

achievements:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**YES**

The `achievements:` section of your config is where you configure *player-based "achievement" tracking*.

Like most things in MPF configs, the highest-level entries in the `achievements:` section of your config are the names of the individual achievements, and then indented under each of those are the settings for that individual achievement.

Here's an example achievements section from Brooks & Dunn:

```
##! mode: mode1
achievements:
  world_tour:
    show_tokens:
      leds: l_world_tour
    show_when_selected: flash
    show_when_started: flash
    show_when_completed: on
    events_when_started: start_world_tour_mode
    restart_after_stop_possible: true
    events_when_completed: rotate_mission_rotator, light_mission_select
    complete_events: world_tour_success
    enable_events: world_tour_fail, ball_will_end
  money_bags:
    show_tokens:
      leds: l_money_bags
    show_when_selected: flash
    show_when_started: flash
    show_when_completed: on
    events_when_started: start_money_bags_mode
    restart_after_stop_possible: true
    events_when_completed: rotate_mission_rotator, light_mission_select
    complete_events: money_bags_success
    enable_events: money_bags_fail, ball_will_end
  music_awards:
    show_tokens:
      leds: l_music_awards
    show_when_selected: flash
    show_when_started: flash
    show_when_completed: on
    events_when_started: start_music_awards_mode
    restart_after_stop_possible: true
    complete_events: music_awards_success
    events_when_completed: rotate_mission_rotator, light_mission_select
    enable_events: music_awards_fail, ball_will_end
  jukebox:
    show_tokens:
      leds: l_jukebox_insert
    show_when_selected: flash
    show_when_started: flash
    show_when_completed: on
    events_when_started: start_jukebox_mode
    restart_after_stop_possible: true
    events_when_completed: rotate_mission_rotator, light_mission_select
```

(continues on next page)

(continued from previous page)

```
    complete_events: jukebox_success
    enable_events: jukebox_fail, ball_will_end
  play_poker:
    show_tokens:
      leds: l_play_poker
    show_when_selected: flash
    show_when_started: flash
    show_when_completed: on
    events_when_started: start_play_poker_mode
    restart_after_stop_possible: true
    events_when_completed: rotate_mission_rotator, light_mission_select
    complete_events: play_poker_success
    enable_events: play_poker_fail, ball_will_end
```

More examples:

- *Recipe: The Addams Family Mansion Awards*
- *achievement (example config files)*

Shows

The `show_when_xxx` settings control which show is played when this achievement switches to a new state.

Note that whatever show was playing from the previous state will be stopped.

Also, any tokens configured in the `show_tokens:` section will be passed to the show here.

Events posted by achievements

You can configure achievements to post certain events when they change state.

Note that all achievements will by default post events in the form *achievement_(name)_state_(state)* when they change state. The events listed below as `events_when_xxx`, if defined, will replace the default event.

Control Events

The following `xxx_events` settings specify which MPF events cause this achievement to move to a new state.

Optional settings

The following sections are optional in the `achievements:` section of your config. (If you don't include them, the default will be used).

complete_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, cause this achievement to switch to its “completed” state. This must be in the “started” state in order to be moved to the “completed” state when these events post. These events will also cause the achievement to play the show defined in the `show_when_completed:` setting and to emit (post) events in the `events_when_completed:` setting.

disable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, cause this achievement to switch to its “disabled” state. These events will also cause the achievement to play the show defined in the `show_when_disabled:` setting and to emit (post) events in the `events_when_disabled:` setting.

enable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, cause this achievement to switch to its “enabled” state. These events will also cause the achievement to play the show defined in the `show_when_enabled:` setting and to emit (post) events in the `events_when_enabled:` setting.

enable_on_next_ball_when_enabled:

Single value, type: `boolean` (`true`/`false`). Default: `true`

If True/Yes, this achievement will stay “enabled” when the next ball starts if it was enabled when the last ball ended. If False/No, this achivement will be changed to “disabled” when the next ball starts.

This is similar to the `restart_on_next_ball_when_started:` event from above, except it applies to the “enabled” state instead of the “started” state.

This setting will also play the `show_when_enabled:` show and post the `events_when_enabled:` events when re-enabling, but will not play or post anything when disabling.

events_when_completed:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A single event, or a list of events, that will be posted when this achievement is complete.

events_when_disabled:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A single event, or a list of events, that will be posted when this achievement is disabled.

events_when_enabled:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A single event, or a list of events, that will be posted when this achievement is enabled.

events_when_selected:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A single event, or a list of events, that will be posted when this achievement is selected.

events_when_started:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A single event, or a list of events, that will be posted when this achievement is started.

events_when_stopped:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A single event, or a list of events, that will be posted when this achievement is stopped.

reset_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, cause this achievement to reset back to its default state (which will either be "disabled" or, if you have `start_enabled: true`, "enabled")

restart_after_stop_possible:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Is it possible to restart this achievement after it's been stopped?

restart_on_next_ball_when_started:

Single value, type: `boolean` (`true`/`false`). Default: `false`

If True/Yes, then this achievement will stay in the "started" state when the player's next ball starts if it was in the "started" state when the previous ball ended. This is useful if you want to restart a mode that was running when the ball ended.

Note that this restart will also play the `show_when_started:` show, and it will also post the `events_when_started:` events.

If False/No, this achievement's state will change from "started" to "stopped" when the next ball starts. This will *not* play the `show_when_stopped:` show and it will *not* post the `events_when_stopped:` events.

select_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, cause this achievement's selected property to 'true'. These events will also cause the achievement to play the show defined in the `show_when_selected:` setting and to emit (post) events in the `events_when_selected:` setting.

Note that "selected" property, in MPF, is used to describe an achievement that is currently selected ("highlighted" or "lit") and available to be started. This would typically be tied to a show (via the `show_when_selected:` setting) that causes a light or LED to flash.

Also, note that more than one achivement may be selected at a time; see `unselect_events:` to change the selected property to false.

show_tokens:

One or more sub-entries. Each in the format of `string` : `string`

This is an indented list of key/value pairs for the *show tokens* that will be sent to the shows that are played when this achievement changes state. (See the settings called "show_when_XXX" further down in this documentation.)

show_when_completed:

Single value, type: string name of a *shows* device. Defaults to empty.

Name of the show that will be started when this achievement has been completed.

show_when_disabled:

Single value, type: string name of a *shows* device. Defaults to empty.

Name of the show that will be started when this achievement has been disabled.

show_when_enabled:

Single value, type: string name of a *shows* device. Defaults to empty.

Name of the show that will be started when this achievement has been enabled.

show_when_selected:

Single value, type: string name of a *shows* device. Defaults to empty.

Name of the show that will be started when this achievement has been selected.

show_when_started:

Single value, type: string name of a *shows* device. Defaults to empty.

Name of the show that will be started when this achievement has been started.

show_when_stopped:

Single value, type: string name of a *shows* device. Defaults to empty.

Name of the show that will be started when this achievement has been stopped.

speed:

Single value, type: `number` (will be converted to floating point). Default: `1`

Playback speed of all shows that are referenced by this achievement.

start_enabled:

Single value, type: `boolean` (`true`/`false`). Defaults to empty.

Whether this achievment is enabled or disabled when it is first loaded.

start_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Events in this list, when posted, cause this achievement to switch to its "started" state. These events will also cause the achievement to play the show defined in the `show_when_started:` setting and to emit (post) events in the `events_when_started:` setting.

stop_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Events in this list, when posted, cause this achievement to switch to its "stopped" state. These events will also cause the achievement to play the show defined in the `show_when_stopped:` setting and to emit (post) events in the `events_when_stopped:` setting.

sync_ms:

Single value, type: `integer`. Defaults to empty.

A sync_ms value used for any shows which are started by this achievement. See the *full sync_ms documentation for details*.

unselect_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, cause this achievement's 'selected' property to become false.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Enables debug logging.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Not used.

Related How To guides

- *Achievement Groups*
- *Achievements*

achievement_groups:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**YES**

The `achievements_groups:` section of your config is where you configure grouping of multiple *player-based "achievement" tracking*.

Like most things in MPF configs, the highest-level entries in the `achievement_groups:` section of your config are the names of the individual achievement group, and then indented under each of those are the settings for that group.

Here's an example achievement_groups section from Brooks & Dunn. (This is related to the example in the achievements config documentation.)

```
##! mode: mode1
achievement_groups:
  my_group:
    achievements: world_tour, money_bags, music_awards, jukebox, play_poker
    enable_events: enable_mission_selection
    start_selected_events: shot_lower_vuk_from_playfield_hit
    select_random_achievement_events: rotate_mission_rotator
    events_when_enabled: mission_rotator_ready
    rotate_right_events: sw_toggle
    show_tokens:
      leds: l_begin_round
    show_when_enabled: flash
```

More examples:

- *Recipe: The Addams Family Mansion Awards*
- *achievement (example config files)*

Required settings

The following sections are required in the `achievement_groups:` section of your config:

achievements:

List of one (or more) values, each is a type: string name of a *achievements* device. Defaults to empty.

This is a list of the achievements (from the `achievements:` section of your mode config) that make up this group. The order here defines the order individual achievements are rotated in via the `rotate_right_events:` and/or `rotate_left_events:` settings.

Optional settings

The following sections are optional in the `achievement_groups:` section of your config. (If you don't include them, the default will be used).

allow_selection_change_while_disabled:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Controls whether the currently selected achievement can be changed when the achievement group is disabled. If False/No, then the rotate and select random events will have no effect when the group is disabled.

auto_select:

Single value, type: `boolean` (`true`/`false`). Default: `false`

If True, this achievement group will automatically ensure that one of its member achievements is always selected. The selected achievement will be chosen at random from all the achievements in the "enabled" states (and the "stopped" states if `restart_after_stop_possible:` is set to True).

disable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, disable this achievement group. These events will also cause the achievements to play the show defined in their `show_when_disabled:` setting and to emit (post) events in their `events_when_disabled:` settings.

disable_random:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Todo: *Help us to write it*

---

disable_while_achievement_started:

Single value, type: `boolean` (`true`/`false`). Default: `true`

If True, this achievement will automatically disable itself when any of its member achievements are in the "started" states. This is the default behavior because an achievement group is typically used to select an achievement to run, and while an achievement is running, you usually want to disable the selection process for the next achievement.

enable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, will enable this achievement group. This will play the `show_when_enabled:` and will post events in the `events_when_enabled:` settings.

This will also check to see if all the member achievements are complete, it will check to see if there are no more enabled achievements, and it will update the selected achievement.

Starting the selected achievement only works if the group is enabled. In other words, if something has to be "lit" before an achievement can start, then that is done via the group's "enable" functionality.

enable_while_no_achievement_started:

Single value, type: `boolean` (`true`/`false`). Default: `true`

If True, this achievement will automatically enable itself when none of its member achievements are in the "started" states. This is the default behavior because an achievement group is typically used to select an achievement to run, so when none are running, you want to enable the group so that the next achievement can be selected.

events_when_all_completed:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A single event, or a list of events, that will be posted when all the achievements in this group are in the "completed" state. This is useful for posting events to start a wizard mode, for example.

events_when_enabled:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A single event, or a list of events, that will be posted when this achievement group is enabled.

events_when_no_more_enabled:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A single event, or a list of events, that will be posted when one of the events in the `select_random_achievement:` is posted but there are no more available achievements to be selected.

rotate_left_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Same as `rotate_right_events:`, but it rotates the selected achievement in the opposite direction.

rotate_right_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Causes the states of the available achievements in this group to be rotated to the right.

Note that the `allow_selection_change_while_disabled:` controls whether these events will work when the achievement group is disabled.

This is used to "switch" the current selected achievement. For example, many games have main achievements you need to complete to get to wizard mode. Completed achievements have a light that's solid on, available (enabled) achievements have a light that's off (since they're not yet complete but available to be played), and the current selected achievement has a light that's flashing (indicating that it's the next one to be played).

Then when you hit a slingshot or pop bumper, the currently selected (flashing) achievement changes, but you only want to rotate with other achievements that are enabled (available but not yet complete).

So if this is the current state:

- Mission 1: completed
- Mission 2: selected
- Mission 3: enabled
- Mission 4: enabled
- Mission 5: enabled

And then one of the `rotate_right_events:` is posted (like from a pop bumper hit), the new list would look like this:

- Mission 1: completed
- Mission 2: enabled
- Mission 3: selected
- Mission 4: enabled
- Mission 5: enabled

Notice that the "selected" state moved from Mission 2 to Mission 3, and the completed state of Mission 1 did not change.

Even though these are called "rotate" events, what really happens is that when this rotation occurs, the previously selected achievement changes from "selected" to "enabled", and the newly selected achievement changes from "enabled" to "selected". Both achievements will stop their current shows and play the shows associated with their new states, and both will post the events associted with their new states.

Note that if you want to select a random achievement instead of the next one on the list, you can use a `select_random_achievement_events:` event instead.

select_random_achievement_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, will randomly pick one of the available achievements and change it to its "selected" state. This is useful when a game is starting and you want one of the available achievements to start in a selected state. (e.g. pick a random mission to be highlighted.)

Note that the `allow_selection_change_while_disabled:` controls whether these events will work when the achievement group is disabled.

The "available" achievements which could be chosen here include achievements that are one of the following:

- enabled
- selected
- stopped (if the achievement's `restart_after_stop_possible:` is true/yes

An example of this would be in Attack From Mars, where the next country is randomly chosen (selected) after you default the saucer for the previous country.

If there are no more available events to be selected, then the events in `events_when_no_more_enabled:` are posted.

Note that if you want to always select a certain achievement (instead of randomly picking one), then you can just set that particular achievement's `select_events:` entry rather than using this random selecting setting.

show_tokens:

One or more sub-entries. Each in the format of `string` : `string`

This is an indented list of key/value pairs for the *show tokens* that will be sent to the shows that are played when this achievement changes state.

Note that you can configure `show_tokens:` at the group level (here) or the individual achievement level. That's done for convenience, and in practical use, you'd just configure the show tokens in one place.

show_when_enabled:

Single value, type: string name of a *shows* device. Defaults to empty.

Name of the show that will be started when this achievement group has been enabled. Also, any tokens configured in the `show_tokens:` section will be passed to the show here.

start_selected_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Events in this list, when posted, cause any achievements in this group that are in the "selected" state to switch to their "started" state. (Typically there would only be a single achievement in the group that's "selected" at any time, but you could have more than one.)

These events only work if the achievement group is enabled.

When the individual achievements change from "selected" to "started", they will play their `show_when_started:` shows and post their `events_when_started:` events.

sync_ms:

Single value, type: `integer`. Defaults to empty.

Todo: *Help us to write it*

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Not used

Related How To guides

- *Achievement Groups*
- *Achievements*

animations:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `animations:` section of your config is where you list the reusable "named" animations.

Note that while you can add animations in both the machine-wide and a mode-specific config, the list of animations is global, meaning that any animation is available in any mode, and you can't have two different animations with the same name.

For example:

```
animations:
  fade_in:
    property: opacity
    value: 1
    duration: 1s
  fade_out:
    property: opacity
    value: 0
    duration: 1s
```

The above example defines animations named `fade_in` and `fade_out` that you can use, by name, in any widget or widget_player config where you would ordinarily define your own animations.

Related How To guides

- *How to animate display widgets*

assets:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `assets:` section of a config file lets you configure the default settings for different types of assets based on what folder those assets are in. Any settings you specify here are just the defaults, though, and you can still override the defaults for an individual asset by adding an entry for it to your machine or mode config file.

Let's take a look at an example:

```
assets:
  images:
    default:
      load: preload
    preload:
      load: preload
```

(continues on next page)

(continued from previous page)

```
  on_demand:
    load: on_demand
  potato:
    some_key: some_value
    something_else: whatever
```

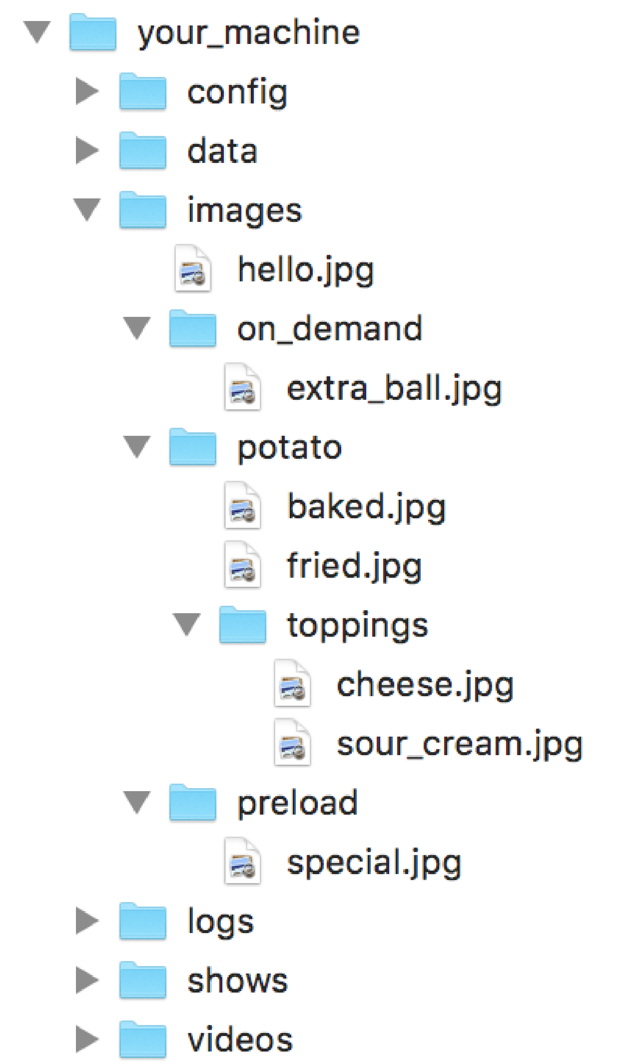
The above config contains the asset settings for *image* assets. Notice there are 4 entries under `images:`: `default`, `preload`, `on_demand`, and `potato`. Those names represent sub-folders that could contain image assets.

Then under each of those, there are one or more key/value pairs. These key/value pairs are applied to assets located in the sub-folders above.

Note: Although you can create sub-folders nested as many levels deep as you wish, only the top-level sub-folder can be listed in the assets section. Any assets in sub-folders below the top level will inherit the settings from their top-level sub-folder parent.

The `default` entry is special, as it applies to the root folder as well as any assets that are in folders that are not specified here.

Consider the following files & folders in a machine folder with the `assets:` section from above:



In this case, `/your_machine/images/hello.jpg` would have the `default:` settings applied, `/your_machine/images/preload/special.jpg` would have the `load: preload` key/value pair applied to it, `/your_machine/images/potato/toppings/cheese.jpg` would have the `some_key: some_value` and `something_else: whatever` key/value pairs applied to it, etc.

The `assets:` section of the config file doesn't really care what the key/value pairs are. They're just the defaults for the assets in those folders, and if they're not valid settings then MPF will give you an error. (Note that different types of assets have different settings options and different keys & values that are correct.)

Currently MPF supports four kinds of assets. Click on each to go to that asset type's description in the config file reference which will explain what settings and be used and what the options are.

Asset types include:

- *shows:* (use file_shows entry)
- *images:*
- *sounds:*
- *videos:*

auditor:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The *auditor:* section of the machine configuration file lets you control what events the MPF's auditor module includes in its audits.

Here's an example which is the settings we including in the default *mpfconfig.yaml* file. (So these are the settings that are included by default with every game you run.) Also, by default, the auditor saves its audits to `/audits/audits.yaml` in the folder for each machine. (Check out the documentation on the Auditor to see a sample audit log file.)

```
auditor:
  save_events: ball_ended game_ended
  audit: shots switches events player
  events: ball_search_begin machine_init_phase_1 game_started game_ended machine_reset
  player: score
  num_player_top_records: 10
```

Optional settings

The following sections are optional in the `auditor:` section of your config. (If you don't include them, the default will be used).

audit:

List of one (or more) values, each is a type: `string`. Defaults to empty.

This is a list of the various types of things you want to include in your audit file. There are currently four options:

- shots - tracks the number of times each shot has been made
- switches - tracks the number of times each switch has been hit.
- events - whether the auditor should audit certain events. (Add the events you want to track to the *events* section.)
- player - includes player variables (score, maybe shots or goals they've achieved, etc.) See the *player* section below for details.

events:

List of one (or more) events. The device will add handlers for those events. Defaults to empty.

A list of which events you want to audit. These are the names of any events you want.

num_player_top_records:

Single value, type: `integer`. Default: `1`

For player-specific variables, you have the option of track the "top" number of each. So in the example above, since the only player item is *score*, the auditor will track the top 10 highest scores, plus the total count and the overall average.

player:

List of one (or more) values, each is a type: `string`. Defaults to empty.

A list of player variables you want to audit. The auditor will save a certain number (configurable via the `num_player_top_records:` setting), as well as the total number of entries and the current average.

reset_audit_events:

List of one (or more) events. The device will add handlers for those events. Default: `auditor_reset,factory_reset`

Events to reset audits in the machine. This is used by the service mode.

save_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `ball_ended`

Default: `ball_ended`

Events in this list, when posted, trigger the auditor to save its audits to disk.

Related How To guides

- *Auditor*

autofire_coils:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The *autofire_coils:* section of your config file contains all the settings for the coils which you would like to fire automatically based on a switch activation in a pinball machine.

Here's an example:

```
switches:
  s_left_sling:
    number: 1
  s_right_sling:
    number: 2
coils:
  c_left_sling:
    number: 1
    default_pulse_ms: 10ms
  c_right_sling:
    number: 2
    default_pulse_ms: 10ms
autofire_coils:
  left_sling:
    coil: c_left_sling
    switch: s_left_sling
  right_sling:
    coil: c_right_sling
    switch: s_right_sling
```

Note that autofire coils in MPF are 1-to-1 in terms of coils-to- switches, so a single entry is for one switch to control one coil. On some platforms, you can have two switches control a single coil (or two coils controlled by a single switch), but to do that you would create two separate *autofire_coils:* entries with one coil and one switch each. (And again, that's platform-specific. Check your hardware platform documentation for details.)

If you're wiring your slingshots and you want two switches to control a single coil, on nearly 100% of pinball machines in the world, those two switches are wired together and use a single input, so the hardware sees them as a single switch. (Just be sure to wire them in parallel, not series, so that either switch closing causes the hardware to see the switch activation.) The top-level setting is the name you can refer to this autofire coil as, such as `left_sling:` or `right_sling:` in the example above.

Then each entry has the following required and optional settings:

Required settings

The following sections are required in the `autofire_coils:` section of your config:

coil:

Single value, type: string name of a *coils* device. Defaults to empty.

The name of the coil you want to fire. (Actually, perhaps we should phrase it as the name of the coil you want to change the state on, because you can also use these autofire coil rules to cause coils to stop firing based on a switch change.)

switch:

Single value, type: string name of a *switches* device. Defaults to empty.

The name of the switch which will trigger the autofire coil. More precisely, this switch is used together with the `coil` in the hardware rules which will instruct your pinball hardware to pulse the coil.

Optional settings

The following sections are optional in the `autofire_coils:` section of your config. (If you don't include them, the default will be used).

ball_search_order:

Single value, type: `integer`. Default: `100`

A relative value which controls the order individual devices are pulsed when ball search is running. Lower numbers are checked first. Set to `0` if you do not want this device to be included in the ball search. See the *Ball Search* documentation for details.

coil_overwrite:

Single value, type: *coil_overwrites*. Defaults to empty.

You can overwrite `recycle`, `pulse_ms`, `pulse_power` or `hold_power` of the coil for this device.

This is an example:

```
switches:
  s_left_sling:
    number: 1
coils:
  c_left_sling:
    number: 1
    default_pulse_ms: 10ms
autofire_coils:
  stronger_left_sling:
    coil: c_left_sling
    switch: s_left_sling
    coil_overwrite:
      pulse_ms: 20ms
```

In this example we increase `pulse_ms` of the slingshot. If you define multiple versions of a autofire_coil (here slingshot) make sure that you only enable one of them at a time.

coil_pulse_delay:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `0`

This setting will delay the pulse of your `coil` by a certain milliseconds after your `switch` has activated. Please note that this has to be supported in your hardware platform and not all platforms do that.

disable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `ball_will_end, service_mode_entered`

Disables this autofire coil by clearing the hardware rule from the pinball controller hardware.

enable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `ball_started`

Enables this autofire coil by writing the hardware rule to the pinball controller hardware.

playfield:

Single value, type: string name of a *playfields* device. Default: `playfield`

The name of the playfield that this autofire device is on. The default setting is "playfield", so you only have to change this value if you have more than one playfield and you're managing them separately.

reverse_switch:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Boolean which controls whether this autofire device fires when the switch is active or inactive. The default behavior is that the coil is fired when the switch goes to an active state. If you want to reverse that, so the coil fires when the switch goes to inactive, then set this to False. (This is what you would use if you have an opto.) Default is *False*.

switch_overwrite:

One or more sub-entries. Each in the format of `string` : `string`

You can overwrite the `debounce` setting of your switch in this device.

timeout_disable_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `0`

To prevent machine gunning of your autofire coils (i.e. pops or slings) you can define a windows `timeout_watch_time`. If more than `timeout_max_hits` hits to your switch (and thus responses by your coil) are seen by MPF it will disable the hardware rule for `timeout_disable_time` and reinstall it afterwards.

timeout_max_hits:

Single value, type: `integer`. Default: `0`

To prevent machine gunning of your autofire coils (i.e. pops or slings) you can define a windows `timeout_watch_time`. If more than `timeout_max_hits` hits to your switch (and thus responses by your coil) are seen by MPF it will disable the hardware rule for `timeout_disable_time` and reinstall it afterwards.

timeout_watch_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `0`

To prevent machine gunning of your autofire coils (i.e. pops or slings) you can define a windows `timeout_watch_time`. If more than `timeout_max_hits` hits to your switch (and thus responses by your coil) are seen by MPF it will disable the hardware rule for `timeout_disable_time` and reinstall it afterwards.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

See the *documentation on the debug setting* for details.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

The plain-English name for this device that will show up in operator menus and trouble reports.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Special / reserved tags for autofire coils: *None*

See the *documentation on tags* for details.

Related How To guides

- *Autofire Coils*
- *Pop Bumpers*
- *kickbacks:*

ball_devices:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `ball_devices:` section of your config is where you configure your *ball devices*.

You can find examples here:

- *Troughs*
- *Plungers*
- *Scoops/Vertical UP Kickers (VUKs)/Saucer Holes*

Optional settings

The following sections are optional in the `ball_devices:` section of your config. (If you don't include them, the default will be used).

auto_fire_on_unexpected_ball:

Single value, type: `boolean` (`true`/`false`). Default: `true`

If a ball randomly shows up in this device, should it be automatically ejected?

ball_capacity:

Unknown type. See description below.

Optional value for how many balls this device can hold. You only need to specify this if your device holds more balls that it has *ball_switches* for. (In other words, probably 99% of the ball devices in the world don't need this because they have one switch for each ball.) Some devices, like the Dead World lock in *Judge Dredd* or the gumball machine in *Twilight Zone* don't have a 1-to-1 mapping for ball switches to balls held, so you would use this setting to tell MPF how many balls that device can hold. Default will be set to the number of *ball_switches* there are.

ball_missing_target:

Single value, type: string name of a *playfields* device. Default: `playfield`

When a ball is goes missing from a device, this is the name of the ball device that will get the ball added to it. (After all, the ball didn't just vaporize. It went somewhere.) The default is *playfield*. (In other words, if a ball disappears from a device, MPF assumes it's on the playfield unless you specify a different device here.) Most devices have ball switches which means that a ball which disappears from a device that only has an exit to another device will be picked up by that device. But if you have a device that leads into another device that doesn't know how many balls it has, or if you have multiple playfields, you can set that target here. Default is *playfield*.

ball_missing_timeouts:

List of one (or more) values, each is a type: `time string (ms)` (*Instructions for entering time strings*). Defaults to empty.

A list of timeouts that correspond to how much time after a ball goes missing passes before MPF assumes that ball went into this device's target device. This is a list, so you can enter multiple values to match the multiple entries in your *eject_targets:* list. If you don't enter a value here, or if the number of values you enter here are less than the number of eject targets this device has, MPF uses *20 seconds* as the default.

ball_search_order:

Single value, type: `integer`. Default: `200`

A relative value which controls the order individual devices are pulsed when ball search is running. Lower numbers are checked first. Set to `0` if you do not want this device to be included in the ball search. See the *Ball Search* documentation for details.

ball_switches:

Unknown type. See description below.

A list of switch names that are active when a ball is in the device. It's assumed there is a one-to-one *ball switch* to *ball* ratio, so if you have three switches then MPF assumes that device can hold three balls. (Note that if your device can hold more balls than it has switches for, like the gumball machine in *Twilight Zone* , then you can use the *ball_capacity:* setting to specify how many balls it can hold.) MPF uses these switches to count how many balls a device has at any time by counting how many of them are active. Note that "active switch" means "there is a ball here." So if you have a trough with opto switches which "invert" their state, then you will have to configure those switches with the "NC" (normally closed) type in the `switches:` section of your config file. Default is *None* . (Meaning this device tracks the number of balls it has virtually based on *entrance_switch* activations.)

captures_from:

Single value, type: string name of a *playfields* device. Default: `playfield`

This is the name of the ball device that this device captures balls from. In other words, if a ball randomly appears in this device, it assumes it came from this *captures_from* device. Default is *playfield*.

confirm_eject_event:

Single event. The device will add an handler for this event. Defaults to empty.

This is the name of the event that will be used to confirm a successful ball eject if you have `confirm_eject_type: event`.

confirm_eject_switch:

Single value, type: string name of a *switches* device. Defaults to empty.

This is the name of the switch activation that will be used to confirm a successful ball eject if you have `confirm_eject_type: switch`.

confirm_eject_type:

Single value, type: one of the following options: target, switch, event, fake. Default: `target`

Whenever the a ball device attempts to eject a ball, it needs to verify that the ball was actually ejected properly. There are several ways that eject verification can take place, and this option allows you to specify which verification method you want. Note that many of these options require further configuration settings. Options for confirming the eject include:

- `target` (default) - This device will confirm the eject via a ball successfully entering the "target" device it was ejecting the ball to. (The target device is one of the entries from your *eject_targets:* list and can either be a *ball device* or the *playfield*. Note that if the target device is a playfield and the playfield already has an active ball, then the eject confirmation will be changed to *count* since it wouldn't know if a playfield switch being hit was based on the newly-ejected ball or one of the existing playfield balls.
- `event` - The ball device will look for a specific event, and when it sees that event, it knows the eject was successful. This can be any event you want, specified via the *confirm_eject_event:* setting.
- `switch` - If your ball device has a switch which is activated when the ball exits, you can use this *switch*type of confirmation. Then when the ball device sees this switch become active (even if it's momentary), it knows the eject was successful. An example of this might be if there's a switch on the ball gate at the top of a plunger lane. Note that you only want to use this type of eject confirmation if the eject confirmation switch cannot be activated by balls on the playfield. Otherwise if you're trying to eject a ball when you already have one in play, you wouldn't know if the newly-ejected ball hit that switch or if an existing live ball hit it. This can be any switch you want, specified via the *confirm_eject_switch:* setting.
- `fake` - This is a setting that's used by other devices (such as the ball lock) when they do not want to use eject confirmation because they have another way of confirming the eject. It's not an option that you would use when setting up devices, but it's included here in case you happen to see a reference to it in the code or the log files.

counter:

Unknown type. See description below.

Todo: *Help us to write it*

---

eject_all_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Causes this device to eject all its balls.

eject_coil:

Unknown type. See description below.

The coil that is fired to eject a ball from this device.This *eject_coil* is optional, since some devices (like a manual plunger or the playfield) don't have eject coils. Default is *None*.

eject_coil_enable_time:

Unknown type. See description below.

When using an `eject_coil` and specifying `eject_coil_enable_time` MPF will enable to `eject_coil` for `eject_coil_enable_time` instead of pulsing that coil.

eject_coil_jam_pulse:

Unknown type. See description below.

This is the pulse time, in ms, that the eject coil will use if the jam switch is active and the first eject attempt failed to eject the ball. (In other words, if the jam switch is active, the ball device will try to eject the ball with the regular pulse time. If that fails, then subsequent ejects will use this pulse time instead. Default is *None* which means the ball device will not change the pulse time after 2 attempts.

eject_coil_max_wait_ms:

Unknown type. See description below.

MPF might delay the eject by `eject_coil_max_wait_ms` to ensure consistent pulses. See *psus:* for details.

eject_coil_reorder_pulse:

Unknown type. See description below.

Pulse duration to use to reorder balls. If the ball device assumes that the balls are not settled properly it will pulse the `eject_coil` for `eject_coil_reorder_pulse` and recount the balls. This might happen if multiple balls disappear or the `jam_switch` is active.

eject_coil_retry_pulse:

Unknown type. See description below.

The new pulse time, in ms, that the eject coil will use if the eject has failed too many times. This pulse time is used up until the device stops trying. Default is *None* which means the ball device will not change the pulse time after failed attempts.

Note that the number of times the ball device will attempt the eject before increasing the pulse time is controlled in the `retries_before_increasing_pulse:` setting.

eject_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Causes this device to eject one ball.

eject_targets:

List of one (or more) values, each is a type: string name of a *ball_devices* device. Default: `playfield`

A list of one or more ball devices and/or the word "playfield" which is used to specify all the ball devices this device can directly eject a ball to. This is a very important concept and can be somewhat confusing, so bear with us as we try to explain it.

Every time a ball device ejects a ball, MPF needs to "confirm" that the ball was successfully ejected. There are several different methods which can be used to confirm the eject, and you configure which method you want to use for each ball device via the *confirm_eject_type:* setting.

In many cases, it's possible that a single ball device can actually eject a ball into one of several different targets. For example, in *Star Trek: The Next Generation*, the main plunger catapult fires the ball into the top of the playfield where there is a controlled drop target blocking the entrance to a subway. If that drop target is up, then the ball bounces off it and then is live on the playfield. If that drop target is down, a ball ejected from the catapult flies past it and into the subway. Once in the subway, there is a series of diverters which can activate or deactivate to route the ball to either the *left VUK*, the *leftcannon*, or the *right cannon*. In that machine, the *left VUK*, *left cannon*, and *right cannon* are all ball devices. So the *eject_targets:* setting looks like this:

```
eject_targets: playfield, bd_leftVUK, bd_leftCannonVUK, bd_rightCannonVUK
```

In other words, the *eject_targets:* list is a list of *all possible ball devices* that this device can eject a ball to.

Notice that the word *playfield* is also in that list, because if that drop target is up, then the ball ejected from the catapult ends up on the playfield, so *playfield* is a valid target too. (In MPF, the playfield is also a ball device.)

At this point you might be wondering what the point of this is? The reason you specify all these target devices is because MPF's ball controller and ball device code work hand-in-hand with MPF's diverter code to automatically "route" balls to ball devices that want them. So in *Star Trek*, you can use a command to say "the left VUK should have one ball," and MPF will see the source device for that ball (the *catapult*, in this case, since it includes *bd_leftVUK*in its list of eject targets) and it will cause the catapult to eject a ball. (What's happening behind the scenes is that the catapult posts an event which says "I'm ejecting a ball with a target destination of the *bd_leftVUK*"), and all the diverters (including that top drop target) will see that and automatically position themselves accordingly so the ball gets to where it needs to go.

Note that you only want to include devices in this list that are directly accessible as targets for balls ejecting from this device. In other words your machine will probably have lots of ball locks and other devices that the player can hit via flippers and balls from the playfield. Those devices should not be on this list, because technically balls enter them from the playfield, not from the catapult.

The order of your *eject_targets:* list doesn't really matter except for the first entry. If a ball device is ever asked to eject a ball but a target is not specified, then the first entry on this list will be used as the target. (In practice this shouldn't really ever happen.)

eject_timeouts:

List of one (or more) values, each is a type: `time string (ms)` (*Instructions for entering time strings*). Defaults to empty.

This is an optional list of one or more MPF time strings that specify how long the device should wait for an ejected ball to be confirmed before it assumes the eject failed. The order you enter them here matches up with the order of your *eject_targets*. For example, consider the following two lines from a ball device configuration:

```
eject_targets: playfield, bd_leftVUK, bd_leftCannonVUK, bd_rightCannonVUK
eject_timeouts: 500ms, 2s, 4s, 4s
```

When this device is ejecting a ball to the *playfield*, the timeout will be *500ms*. When it's ejecting to the *bd_leftVUK*, the timeout is *2 seconds*, etc. If you don't specify a list of eject timeouts, or if the length of the list is less than the number of eject targets, then the default value of *10 seconds* is used.

See *Fine-tuning ball device timing* for details about thouse timeouts.

ejector:

Unknown type. See description below.

You ejector implemententation and settings. By default MPF will select an implementation based on the settings and configure it accordingly.

Default ejectors (you can use those via the ball device config):

- mpf.devices.ball_device.pulse_coil_ejector.PulseCoilEjector
- mpf.devices.ball_device.enable_coil_ejector.EnableCoilEjector
- mpf.devices.ball_device.hold_coil_ejector.HoldCoilEjector

Additional ejectors:

- mpf.devices.ball_device.event_ejector.EventEjector

```
ball_devices:
  device_with_eject_event:
    ejector:
      class: mpf.devices.ball_device.event_ejector.EventEjector
      events_when_eject_try: my_ball_device_eject
    ball_switches: s_ball_switch1, s_ball_switch2
```

entrance_count_delay:

Unknown type. See description below.

This is the time delay (in MPF time string format) that this ball device will wait before counting the balls after any of the *ball_switches* changes state. This delay exists because there's often a "settling time" when a ball first enters a device where the balls are bouncing around and the switches change state really fast. Default is *500ms*.

entrance_event_timeout:

Unknown type. See description below.

How long does the ball need after an `entrance_event` to settle in the ball device? This is used for some heuristics to determine if this is a new ball or if the ball returned from a failed eject.

entrance_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

device control events format.

Default: `None` (Note that if you add an entry here, it will replace the default. So if you also want the default value(s) to apply, add them too.)

These events tell this ball device that a ball has entered (been added to) the device.

entrance_switch:

Unknown type. See description below.

The name of a switch that is activated when a ball enters the device. Most devices don't have this, since they have the ball switches that are updated and will count the balls. But some devices, like those that do not have switches for each ball, have a switch at the entrance that is triggered when a ball enters. This switch has no effect if your ball device has *ball_switches*. Default is *None*.

entrance_switch_full_timeout:

Unknown type. See description below.

When using an `entrance_switch` and setting this to anything except 0, the device will be considered to be full after `entrance_switch_full_timeout` ms. This is used in some troughs where the last ball sits on the entrance switch (see *How to configure an older style trough with two coils and only one ball switch*).

entrance_switch_ignore_window_ms:

Unknown type. See description below.

How long should another entrance switch be ignored after a previous activation?

exit_count_delay:

Unknown type. See description below.

This is the time delay that the device will wait before counting the balls after any after it attempts to eject a ball if the device is configured to verify the eject via a count of the switches.

hold_coil:

Unknown type. See description below.

The name of a coil that is held in the enabled position to hold a ball. This is used in place of an *eject_coil*, and it's for devices that have to hold (like a post) to keep a ball in the device. Disabling the hold coil releases a ball. Default is *None*. An example for such a hold coil is the lock that comes up below Magneto in X-Men. A further lock of this kind is in Avatar below Jake Sully in the transporter link.

hold_coil_release_time:

Unknown type. See description below.

This is the time (in MPF time string format) that devices with *hold_coils* will hold their coil open to release a ball. Default is *1 second*.

hold_events:

Unknown type. See description below.

These events cause this device to enable its hold coil.

hold_switches:

Unknown type. See description below.

A switch (or list of switches) that indicates a ball is in position to be captured by a *hold_coil*. Default is *None*.

idle_missing_ball_timeout:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `5s`

How long should the device wait before declaring a ball missing if it disappeared outside of an eject? Usually balls do not disappear when the device is not ejecting.

jam_switch:

Unknown type. See description below.

Some pinball trough devices have a switch in the "exit lane" part of the trough that can detect if a ball fell back into the trough from the plunger lane. (The extra switch is needed because when the trough ejects the ball, the remaining balls in the trough will all roll down, so if the ejected ball falls back in, it ends up sitting "on top" of the existing balls, so a normal trough ball switch won't see it.)

This switch is known by different names by different manufacturers, having variously been called *trough jam*, *ball up* switch, or *ball stacked* switch. If your ball device has a switch that can detect jams, enter that switch name here. The ball device code in the MPF has a jam switch handler which watches what happens to that switch. For example, if there's an eject in progress and the jam switch becomes active, it assumes the ball fell back in and will try the eject again.

max_eject_attempts:

Single value, type: `integer`. Default: `0`

Defines how many times this ball device will attempt to eject a ball before deciding that the eject permanently failed. A value of zero means there's no limit. (e.g. the device will just keep trying to eject the ball forever.)

mechanical_eject:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Boolean setting which is used to specify whether this ball device has a mechanical eject option. In MPF, a *mechanical eject* is what happens when a player is able to eject a ball from the ball device mechanically, without MPF knowing about it. (A traditional spring- powered plunger is the most common use.) This setting is used because when a mechanical eject happens, from MPF's standpoint it's like the ball just disappeared, so this setting is used to let MPF know that that might happen. Set this to *True* if a mechanical eject is an option for this ball device. Note that it's entirely possible to have devices that support both mechanical ejects as well as coil-fired ejects (with an *eject_coil*), such as a plunger lane with a spring plunger and a coil-fired collar which can be used in auto or manual mode. Default is *False*. However, if this device does not have an *eject_coil* or *hold_coil* defined, then the mechanical_eject setting will automatically be set to *True*.

player_controlled_eject_event:

Single event. The device will add an handler for this event. Defaults to empty.

When using player controlled eject wait for this event to autofire the ball. (*Instructions for entering device control events*)

request_ball_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

These events cause this device to request a ball to be sent to it.

retries_before_increasing_pulse:

Unknown type. See description below.

The number of times this ball device will attempt to eject the ball before increasing the eject coil pulse time as specified in the `eject_coil_retry_pulse:` above.

Note that this number is the attempts that it will increase the pulse, so the default setting of 4 means that it will try the original pulse value 3 times and then increase it on the 4th.

target_on_unexpected_ball:

Single value, type: string name of a *ball_devices* device. Defaults to empty.

Target playfield to use when capturing an unexpected ball.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

See the *documentation on the debug setting* for details.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

The plain-English name for this device that will show up in operator menus and trouble reports.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

See the *documentation on tags* for details.

Special-purpose tags for ball devices include:

- `home` - Specifies that any balls here are "home" and that the game can start. When MPF boots up, any balls that are in devices not tagged with "home" are automatically ejected.
- `drain` - Specifies that a ball entering this device means the ball has "drained" from the playfield. (i.e. it's used to indicate a player lost the ball, versus some other random playfield lock.)

- `trough` - Specifies that this device holds the ball(s) that are not in play. In most cases, your "drain" and "trough" tags will be the same device, though older games (Williams System 11 and early WPC) actually have two devices under the apron, with a "drain" device receiving balls from the playfield which it then immediately kicks over to a "trough" device which holds the balls that are not in play. + `no-eject-on-ballsearch` - Specifies that this device should never attempt to eject a ball as a result of ball search, even when idle and containing no balls.

The use of `ball_add_live` is discontinued. Use `default_source_device` in your *playfield* instead.

Related How To guides

- *Ball Devices*

ball_holds:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `ball_holds:` section of your config is used to list and configure *ball holds*.

Note that ball holds are used to temporarily hold a ball while the game is doing something else. (Starting a video mode, playing an intro show, etc.) If you want to hold and lock a ball towards multiball, use the `multiball_locks:` section instead.

Ball holds do not affect the "balls in play" count, and they are not used to hold balls from ball-to-ball or between players.

Here's an example

```
ball_devices:
  bd_bunker:
    eject_coil: c_eject
    ball_switches: s_ball1
ball_holds:
  bunker:
    balls_to_hold: 1
    hold_devices: bd_bunker
```

Each sub-entry under the `ball_holds:` section is the name of the logical ball hold ("bunker") in the example above. Then each named ball hold has the following settings:

Required settings

The following sections are required in the `ball_holds:` section of your config:

hold_devices:

List of one (or more) values, each is a type: string name of a *ball_devices* device. Defaults to empty.

A list of one (or more) ball devices that will collect balls which will count towards this hold.

Optional settings

The following sections are optional in the `ball_holds:` section of your config. (If you don't include them, the default will be used).

balls_to_hold:

Single value, type: `integer`. Defaults to empty.

The number of balls this ball hold should hold. If you don't include it, then the ball hold capacity will be automatically calculated based on the combined capacity of all the ball devices that make up this ball hold.

If one of the associated hold devices receives a ball and this ball hold is full, then the ball device will just release the ball again.

disable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Event(s) which disable this ball hold.

enable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Event(s) which enable this ball hold.

priority:

Single value, type: `integer`. Default: `0`

Relative priority when claiming balls entering a device. This can be used to give one *ball_hold* or *multiball_lock* preference when claiming balls.

release_all_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Event(s) which cause this ball hold to release all balls.

release_one_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Event(s) which cause this ball hold to release a single ball.

release_one_if_full_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Event(s) which cause this ball hold to release a single ball only if the ball hold contains the number of balls that matches its `balls_to_hold:` setting.

reset_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `machine_reset_phase_3, ball_starting, ball_will_end, service_mode_entered`

Event(s) which cause this ball hold to reset its held ball count.

source_playfield:

Single value, type: string name of a *ball_devices* device. Default: `playfield`

The name of the playfield that feeds balls to this hold. If you only have one playfield (which is most games), you can leave this setting out. Default is the playfield called *playfield*.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

See the *documentation on the debug setting* for details.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

A descriptive label.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Special / reserved tags for ball holds: *None*

See the *documentation on tags* for details.

Related How To guides

- *Ball Holds*
- *Mystery Awards*
- *Ball Locks*
- *Scoops / Vertical up Kickers (VUKs) / Saucer holes*

ball_locks:

Important: The "ball_locks config section have been removed in MPF 0.54. It's been replaced with the *ball_holds:* and *multiball_locks:* sections.

ball_saves:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `ball_saves:` section of your config is where you create *ball save devices*. Here's an example:

```
ball_saves:
  default:
    active_time: 10s
    hurry_up_time: 2s
    grace_period: 2s
    enable_events: mode_base_started
    timer_start_events: balldevice_plunger_lane_ball_eject_success
    auto_launch: true
    balls_to_save: 1
    debug: true
```

Optional settings

The following sections are optional in the `ball_saves:` section of your config. (If you don't include them, the default will be used).

active_time:

Single value, type: `time string (secs) or template` (*Instructions for entering time strings* and *Instructions for entering templates*). Default: `0`

How long the ball save is active (in MPF time string format) once it starts counting down. This includes the *hurry_up_time,* but does not include the *grace_period* time. Leave this setting out (or set it to 0) for unlimited time. Default is *0*.

auto_launch:

Single value, type: `boolean` (`true`/`false`). Default: `true`

True/False which controls whether the ball save should auto launch the saved ball or wait for the player to launch it.

ball_locks:

List of one (or more) values, each is a type: string name of a *ball_devices* device. Defaults to empty.

Use those devices first when ejecting balls to the playfield on ball save. If there are not enough balls in the lock more balls will be requested to the source_playfield.

balls_to_save:

Single value, type: `integer`. Default: `1`

How many balls this ball saver should save before disabling itself. Set it to -1 for unlimited.

delayed_eject_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Delay the eject until a event from `delayed_eject_events` is posted. For instance, this can be used in combination with `only_last_ball` at the end of a wizard mode to drain all balls and continue the game later.

disable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `ball_will_end, service_mode_entered`

Event(s) which disable this ball save, meaning a drained ball will no longer be saved.

early_ball_save_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Event(s) which will trigger a ball save to take place before the current ball has drained. A typical example of this might be switch activation events from outlane switches which can be used to trigger a ball save as soon as the ball hits the outlane.

eject_delay:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `0`

Delay the eject of the new ball for `eject_delay` ms. This might be useful if you want to play a show or some sounds first for dramatic reasons.

enable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Event(s) which enable this ball save. This also starts the ball save timer running unless there are optional timer_start_events present, see below.

grace_period:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `0`

The "secret" time (in MPF time string format) the ball save is still active. This is added onto the *active_time*. Default is *0*.

hurry_up_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `0`

The time before the ball save ends (in MPF time string format) that will cause the *ball_save_<name>_hurry_up* event to be posted. Use this to change the script for the light or trigger other effect. Default is *0*.

only_last_ball:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Only save the last ball. In case two balls are in play and only one drains it will not be saved.

source_playfield:

Single value, type: string name of a *ball_devices* device. Default: `playfield`

Playfield to eject the saved balls to.

timer_start_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, start this ball saver's countdown timer provided the ball save has been enabled, above. This allows the timer to be started separate from the save being enabled. For example, a light might turn on when a the ball save is enabled at the beginning of a players turn. To avoid the timer running out (if the player takes a break before plunging a ball) the timer can be configured not to start until an event in this list is posted.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see more debug output.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

The plain-English name for this device that will show up in operator menus and trouble reports.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Special / reserved tags for ball saves: *None*

See the *documentation on tags* for details.

Related How To guides

- *Ball Saves*
- *Ball Start and End Behaviour*
- *Kickbacks*

bcp:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `bcp:` section of your config file controls how the MPF core engine communicates with the standalone media controller.

There's a default `bcp:` section in the default `mpfconfig.yaml` system-wide defaults section that should be fine to get started, and then you can override it if needed for a specific situation:

```
bcp:
  connections:
    local_display:
      host: localhost
      port: 5050
      type: mpf.core.bcp.bcp_socket_client.BCPClientSocket
      required: true
      exit_on_close: true
  servers:
    url_style:
      ip: 127.0.0.1
      port: 5051
      type: mpf.core.bcp.bcp_socket_client.BCPClientSocket
  debug: false
```

Optional settings

The following sections are optional in the `bcp:` section of your config. (If you don't include them, the default will be used).

connections:

List of one (or more) values, each is a type: *bcp_connection*. Defaults to empty.

The *connections:* section is where you can specify the connections the MPF core engine will make to standalone media controllers. MPF supports connecting to multiple media controllers simultaneously which is why you can add multiple entries here.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see more debug messages in the log.

servers:

List of one (or more) values, each is a type: *bcp_server*. Defaults to empty.

The *servers:* section is where you can specify bcp server instances which can be connected from other processes. For instance, this is used for the *service cli*. MPF supports connecting to multiple servers simultaneously which is why you can add multiple entries here.

Related How To guides

- *The MPF Unity BCP Server*
- *Creating your own Media Controller*

bcp_connection:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

The `connections:` setting in your `bcp:` section of your config is where you configure BCP connections MPF should establish on startup.

Required settings

The following sections are required in the `bcp_connection:` section of your config:

type:

Single value, type: `string`.

The class to implement the transport. Use `mpf.core.bcp.bcp_socket_client.BCPClientSocket` to use the standard MPF BCP protocol.

More implementations are possible here. For instance, a highly efficient implemententation for production or an encrypted socket for communication over the Internet.

Optional settings

The following sections are optional in the `bcp_connection:` section of your config. (If you don't include them, the default will be used).

exit_on_close:

Single value, type: `boolean` (Yes/No or True/False). Default: `True`

Whatever MPF should exit if this connection disconnects. This is usually true for the media manager because we want MPF to exit once it is closed.

host:

Single value, type: `string`.

The host to connect to for this connection.

port:

Single value, type: `integer`. Default: `5050`

The port to connect to for this connection.

required:

Single value, type: `boolean` (Yes/No or True/False). Default: `True`

Whatever this connection is required for MPF to run. Set this to false if you want MPF not to wait for this connection on start.

Related How To guides

- *The MPF Unity BCP Server*
- *Creating your own Media Controller*

bcp_server:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

The `servers:` setting in your `bcp:` section of your config is where you configure listeners for incoming BCP connections.

Required settings

The following sections are required in the `bcp_server:` section of your config:

type:

Single value, type: `string`.

The class to implement the transport. Use `mpf.core.bcp.bcp_socket_client.BCPClientSocket` to use the standard MPF BCP protocol.

Optional settings

The following sections are optional in the `bcp_server:` section of your config. (If you don't include them, the default will be used).

ip:

Single value, type: `string`.

The IP to bind the server on. Starting in MPF 0.33, you can use `ip: None` and MPF will listen for incoming connections on all network interfaces.

port:

Single value, type: `integer`. Default: `5050`

The port to listen for incoming connections.

Related How To guides

- *The MPF Unity BCP Server*
- *Creating your own Media Controller*

bitmap_fonts:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `bitmap_fonts:` section of your config is where you configure non-default parameter values for any bitmap font assets you want to use in your game. Note: You do *not* have to have an entry for every single bitmap font you want to use, rather, you only need to add individual assets to your config file that have settings which differ from the default values. (This section is part of the MPF media controller and only available if you're using MPF-MC for your media controller.)

A bitmap font is one that stores each glyph (character) as an array of pixels (that is, a bitmap). It is less commonly known as a raster font. Bitmap fonts are simply collections of raster images of glyphs. For each variant of the font, there is a complete set of glyph images, with each set containing an image for each character. For example, if a font has three sizes, and any combination of bold and italic, then there must be 12 complete sets of images.

MPF-MC currently supports Portable Network Graphics (.png), Graphic Interchange Format (.gif), and bitmap (.bmp) image files for bitmap fonts. In order for MPF to use the bitmap font, a font descriptor must be present. This contains the information necessary to locate each glyph (character) in the bitmap image and other associated information. The font descriptor information may be loaded from a file or provided in the asset settings (font descriptor file format). MPF supports both .xml, .fnt, and .txt files for font descriptor files (binary files are not currently supported).

There is a great online tool for generating bitmap fonts (and the associated font descriptor file) from True Type Fonts: http://kvazars.com/littera/

Here's an example:

```
bitmap_fonts:
  F1fuv:
    file: F1fuv.png
    descriptor: [' !"#$%&,()*+`-./', '0123456789:;<=>?', '@ABCDEFGHIJKLMNO', 'PQRSTUVWXYZ[\]^_', ''
↪'abcdefghijklmno', 'pqrstuvwxyz{|}~ ']
  example_font:
    file: example_font.png
    descriptor: example_font_descriptor.xml
```

Optional settings

The following sections are optional in the `bitmap_fonts:` section of your config. (If you don't include them, the default will be used).

descriptor:

Unknown type. See description below.

Here is an example of a descriptor list for a bitmap image that contains three rows of 15 characters and the specific characters mapped to each position in each row:

```
descriptor: [ 'abcdefghijklmno', 'pqrstuvwxyz 012', '3456789,.:=<>-+' ]
```

Remember the descriptor list only works for monospaced characters (characters that are all the same width and height).

file:

Single value, type: `string`. Defaults to empty.

The file to load when using this bitmap font.

load:

Single value, type: `string`. Defaults to empty.

When should the asset loader load this file? One out of `mode_start`, `on_demand` or `preload`.

Related How To guides

- *Bitmap Fonts*
- *Bitmap Fonts (asset type)*

blinkenlight_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

Note: This section can also be used in a show file in the `blinkenlights:` section of a step.

The `blinkenlight_player:` section of your config is where you add or remove colors to or from a blinkenlight based on events. It's also used in shows (via the `blinkenlights:` section) to add or remove colors in that show step.

Example from a config file:

```
blinkenlight_player:
  some_event:
    my_blinkenlight1:
      action: add
      color: red
      key: mykey1
  some_other_event:
    my_blinkenlight1:
      action: remove
      key: mykey1
```

In the example above, when the event called `some_event` is posted, the color red will be added to my_blinkenlight1's list of colors (this will cause the light to immediately start flashing if it wasn't already). The new color will have the key `mykey1`. The key is used like a name of the color, so that it can be removed later using that key. When the event `some_other_event` is posted, the red color (key `mykey1`) will be removed from the blinkenlight.

Example blinkenlight player from a show:

```
##! show: test
- time: 0
  blinkenlights:
    my_blinkenlight1:
      action: add
      color: blue
      key: blue_color
    my_blinkenlight2: purple
```

The first example shows the full config, while the second shows the "express" config. (What's an "express config?" Details *here*.

The blinkenlight player's express config is the "add" action.

See *Blinkenlight player* for details.

Optional settings

The following sections are optional in the `blinkenlight_player:` section of your config. (If you don't include them, the default will be used).

action:

Single value, type: one of the following options: add, remove, remove_mode, remove_all. Default: `add`

What action the blinkenlight should perform. The `remove_all` action will remove all the colors from the blinkenlight, effectively turning it off. The `remove_mode` action will remove all the colors that were added by the mode that the `remove_mode` action is coming from (remember that a blinkenlight can have colors added from lots of different modes – that's its whole purpose!).

color:

Single value, type: color_or_token. Default: `white`

The only action that requires a color setting is the `add` action. It sets the color to add to this blinkenlight. Color values may be a hex string (e.g. `22FFCC`), a list of RGB values (e.g. `[50, 128, 206]`), a color name (e.g. `turquoise`), or a brightness value (i.e. `AA` or `120`). MPF knows 140+ standard web color names, and you can define your own custom colors in the *named_colors:* section of your config. If you use brightness on an RGB light MPF will use the brightness for every channel. For instance brigness `AA` will result in color `AAAAAA`.

key:

Single value, type: `string`. Defaults to empty.

You can think of this value as a name for the color you're adding or removing from the blinkenlight. If you add a color, then the key allows you to remove the color later using the key to specify which color to remove. If you don't specify a key, then the color is considered "keyless" (see *Blinkenlight player* for more information about keyless colors).

Related How To guides

- *Blinkenlight player*

blinkenlights:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `blinkenlights:` section of your config file is used to define lights on your pinball machine that can be shared by multiple different modes at the same time. Blinkenlights work best with RGB LEDs, because each mode that uses a blinkenlight can specify a color that the blinkenlight should flash for that mode. The blinkenlight will then flash each of its colors in a cycle according to a given schedule.

Here's an example section:

```
blinkenlights:
  blinkenlight_1:
    color_duration: 1s
    off_when_multiple: false
    light: l_left_ramp_arrow
    priority: 1000
  blinkenlight_2:
    cycle_duration: 1s
    off_when_multiple: false
    light: l_right_ramp_arrow
    priority: 1000
```

With the above configuration, two blinkenlights are configured: blinkenlight_1 and blinkenlight_2. You can then use a *blinkenlight_player* to add or remove colors to these blinkenlights from within your modes.

The options are as follows:

Optional settings

The following sections are optional in the `blinkenlights:` section of your config. (If you don't include them, the default will be used).

color_duration:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Defaults to empty.

Either `color_duration` or `cycle_duration` (see below) must be specified for each blinkenlight, but not both.

This specifies the amount of time that each of the blinkenlight's colors will be turned on. For example, if this value is `1s`, then each of the blinkenlight's colors will be on for 1 second.

The more colors that are added to the blinkenlight when `color_duration` is specified, the longer it will take to cycle through all the colors. If you want the blinkenlight's cycle to always last the same amount of time regardless of how many colors the blinkenlight has, then use `cycle_duration` instead (see below).

cycle_duration:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Defaults to empty.

Either `cycle_duration` or `color_duration` (see above) must be specified for each blinkenlight, but not both.

This specifies the length of one cycle of the blinkenlight. The blinkenlight will show all of its colors in one cycle. This includes the "off" color at the end of the cycle, if applicable (see `off_when_multiple` below). For example, if this value is `1s`, and the blinkenlight has 2 colors in its list, and `off_when_multiple` is `false`, then each color will be displayed for 0.5 seconds. If `off_when_multiple` is `true`, then the "off" color will count as a color in the blinkenlight's cycle, and so the each color will only be displayed for 1/3 of a second.

The more colors that are added to the blinkenlight when `cycle_duration` is specified, the shorter each color will be displayed. If you want each color to be displayed for a certain length of time regardless of the number of colors, then use `color_duration` instead (see above).

light:

Single value, type: string name of a *lights* device. Defaults to empty.

This is the name of the light which this blinkenlight controls.

off_when_multiple:

Single value, type: `boolean` (`true`/`false`). Default: `false`

This specifies whether or not to include an "off" color at the end of each cycle when the blinkenlight has more than one color in its list.

For example, if the blinkenlight has 2 colors (red and green) and `off_when_multiple` is `False` (the default value), then the cycles will be red, green, red, green. However, if `off_when_multiple` is `True`, then the cycles will be red, green, off, red, green, off. The "off" color in this case is treated as another color for the purposes of the `color_duration` and `cycle_duration` settings above.

A blinkenlight that only has 1 color in its list will be off at the end of its cycle, regardless of whether `off_when_multiple` is `True` or `False`. For example, the cycles of a blinkenlight that has 1 color (red) will be red, off, red, off.

priority:

Single value, type: `integer`. Default: `0`

The priority of the blinkenlight. If there is a show that uses this blinkenlight's light, and the show and the blinkenlight are happening at the same time, then the light will be controlled by whichever one has the highest priority.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Todo: *Help us to write it*

Related How To guides

Todo: *Help us to write it*

bonus (mode_settings:)

Config File section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**YES**

This section explains how to use the `mode_settings:` section for your machine's End of Ball Bonus mode. You should probably read the full *End of Ball Bonus* documentation first, and then just use this for a reference for the settings later.

Note that the "mode_settings:" section is pretty much a generic placeholder that any mode can use for its own custom settings. So the settings described here are specifically the settings that are used by MPF's built-in bonus mode, and so these settings are only valid in the bonus mode's mode configuration file.

Here's an example from *Brooks 'n Dunn*:

```
##! mode: mode1
mode_settings:
  display_delay_ms: 4000
  hurry_up_delay_ms: 500
  hurry_up_event: flipper_cancel
  bonus_entries:
    - event: quarter_bonus
      score: current_player.quarters * current_player.album_value
    - event: wizard_bonus
      score: 25000
      player_score_entry: num_albums
```

Settings

display_delay_ms:

Time value, default `2s`

The time between each "display event" generated by the bonus mode when its running. (In other words, this is essentially how long each bonus slide is show.) This can be overridden on a slide-by-slide basis.

hurry_up_delay_ms:

Time value, default `500ms`

The time between each "display event" after the bonus "hurry up" mode has been triggered. So if the `display_delay_ms:` is 2 seconds, and then the player hits both flippers at the same time to "hurry up" the bonus display, that hurry up time will be used here.

Note that if you don't want to show the slides faster, rather you just want to jump directly to the last slide, then you can enter a value of `0` here.

hurry_up_event:

Name of an event. Default is `flipper_cancel`.

The event that will cause the bonus mode to change its delay between slides from the `display_delay_ms:` time to the `hurry_up_delay_ms:` time. When this event is posted, the next slide is shown immediately, and the timing is set to the new hurry up value.

end_bonus_event:

Name of an event. Default is `None`.

If you enter an event name here, the bonus mode will pause before posting its *bonus_done* event and wait for this event to be posted. If this event is `None`, then the bonus mode will automatically end. You can enter an event name here if you have something custom you want to do in the bonus mode.

keep_multiplier:

Boolean True/False or Yes/No. Default is `False`.

Controls whether the bonus_multiplier player variable should be reset (to 1) when the bonus mode is over. Default is `False` which will not keep the bonus. (e.g. default is to reset it). Conditional values are supported.

Also note that you can use *dynamic values* here if you want to do math or use settings to make this configurable.

bonus_entries:

A list of sub-entries, with one entry for each "thing" you want to track in the bonus.

This is the real meat of the bonus section. Many modern pinball machines have lots of different things that go into the bonus calculation. So rather than just saying, "Your bonus is 5400 points", it's more like "5 aliens x 25k points each, plus 15 modes x 1m each, plus 4 combos x 100k each, all times the bonus multiplier."

Since this section is entered like a list, you need a dash and a space at the beginning of each new entry so MPF knows where one entry ends and one begins.

Here's how an example might look based on the aliens, modes, and combos example just mentioned:

bonus_entries:

- event: alien_bonus score: 25000 player_score_entry: aliens
- event: mode_bonus score: 1000000 player_score_entry: num_modes
- event: combo_bonus score: 100000 player_score_entry: combos

Let's look at each option you can use in each bonus entry:

event: (required)

The name of the event that is posted by the bonus mode. You should use a `slide_player:` in your bonus mode with slide entries based on these names, so when the bonus mode posts that event, you can show a slide with the relevant information for that bonus entry.

score: (required)

How many points this bonus entry is worth. Note that this will be multiplied by the `player_score_entry:` (if it's present). Also note that you can use *dynamic values* here if you want to do advanced math.

player_score_entry:

An optional name of a player variable that will be multiplied by the `score:` entry. This is useful for the "easy" entries where it's just "some player variable multiplied by some score". (For example, "number of aliens times 25,000".) In the example above, the first entry called "alien_bonus" will multiply the "aliens" player variable times 25000.

Note that the bonus mode doesn't care what player variable you use, and it would be up to you to make sure that the player variable you choose is updated throughout your game (either through a `variable_player:` section or a logic block or something like that).

Also if you choose not to include this entry, that's fine. In that case the `score:` entry will be used by itself. Notice in the example at the top of this page from *Brooks 'n Dunn* that it's not used when we need the advanced math of multiplying two player variables together.

reset_player_score_entry:

Boolean (True/False or Yes/No). Default is `False`.

If this is true/yes, then the bonus mode will reset the `player_score_entry:` back to 0 once the bonus mode is over. This is just a convenience thing for simpler bonus calculations that need to be reset per ball. You don't have to use it can could also reset the player variable some other way.

skip_if_negative:

Boolean (True/False or Yes/No). Default is `False`.

If this is True/Yes and if the score calculation for this bonus entry is less than 0, the event for this bonus entry is not posted *and* the value is not subtracted from the player's score.

skip_if_zero:

Boolean (True/False or Yes/No). Default is `True`.

If this is True/Yes, then if the score calculation for this bonus entry turns out to be 0, then the event for this bonus entry is not posted. This is nice if you don't want a bonus screen to show up for something the player has not done, like "0 ramps = 0 points" or whatever. (Or maybe you want to make this "true" to show the player how bad they are?) :)

coil_overwrites:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

Some devices offer one or multiple `coil_overwrites:` settings where you can overwrite coil settings.

Most commonly this is used in *flippers:* and *autofire_coils:*.

Optional settings

The following sections are optional in the `coil_overwrites:` section of your config. (If you don't include them, the default will be used).

hold_power:

Single value, type: float(0,1).

Overwrite the `hold_power` of the coil for this device. See `default_hold_power` in *coils:* for details.

pulse_ms:

Single value, type: `time string (ms)` (*Instructions for entering time strings*).

Overwrite the `pulse_ms` of the coil for this device. See `default_pulse_ms` in *coils:* for details.

pulse_power:

Single value, type: float(0,1).

Overwrite the `pulse_power` of the coil for this device. See `default_pulse_power` in *coils:* for details.

recycle:

Single value, type: `boolean` (Yes/No or True/False).

Overwrite the `recycle` setting of the coil for this device. See `recycle` in *coils:* for details.

Related How To guides

- *flippers:*
- *kickbacks:*
- *autofire_coils:*

coil_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

Note: This section can also be used in a show file in the `coils:` section of a step.

The `coil_player:` section of your config is where you configure coil/solenoid/driver actions (pulse, enable, disable, etc.) based on events. It's also used in shows (via the `coils:` section) to perform coil actions in that show step.

Example from a config file:

```
coil_player:
  some_event: coil_1
  some_other_event:
    coil_2:
      action: enable
      hold_power: .5
```

In the example above, when the event called `some_event` is posted, coil_1 will pulse. When the event `some_other_event` is posted, coil_2 will enable (be held on) at power level 0.5 (means 50% of maximum power).

Note that the `some_event: coil_1` is entered in a different way than the `some_other_event:`. The first one has a simple key/value pair, whereas the second has a complete nested sub-configuration.

The first example shows the "express" config, while the second shows the full config. (What's an "express config?" Details *here*.

The coil player's express config is the "pulse" action.

Example coil player from a show:

```
##! show: test
- time: 0
  coils:
    coil1: pulse
```

See *Coil player* for details.

Optional settings

The following sections are optional in the `coil_player:` section of your config. (If you don't include them, the default will be used).

action:

Single value, type: one of the following options: pulse, on, off, enable, disable. Default: `pulse`

What action the coil should perform. Note that "on" and "enable" are the same, and that "disable" and "off" are the same.

hold_power:

Single value, type: `number` (will be converted to floating point). Defaults to empty.

This setting lets you control how much power is sent to the coil when it's "held" in the on position. This is an float value from 0-1 (i.e. 0% power to 100% power) which controls the relative power. If not set it will use `default_hold_power` of the *coils:*.

max_wait_ms:

Single value, type: `integer`. Defaults to empty.

The maximum time in ms which MPF might use to delay this pulse for power management reasons. See *Power Management in Software* for details.

pulse_ms:

Single value, type: `integer`. Defaults to empty.

The number of milliseconds you'd like this coil to pulse for. This setting overrides the coil's *default_pulse_ms* setting. Note that this setting only affects pulse actions. Make sure you are not exceeding the coil's *max_pulse_ms* setting. If not set it will use `default_pulse_ms` of the *coils:*.

pulse_power:

Single value, type: `number` (will be converted to floating point). Defaults to empty.

The power factor which controls how much power is applied during the initial pulse phase of the coil's activation. (Note that not all hardware platforms support variable pulse power.)

If not set it will use `default_pulse_power` of the *coils:*.

Related How To guides

- *Coil player*
- *Coils (Solenoids)*
- *Shakers*

coils:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `coils:` section of your config is used to map coil (solenoid) names to driver board outputs. You can also set the default pulse times, set tags, and specify power levels for coils that get held on. This section *can* be used in your machine-wide config files. This section *cannot* be used in mode-specific config files. Here's an example section:

```
coils:
  flipper_right_main:
    number: A0-B0-0
    default_pulse_ms: 30
    max_pulse_ms: 100
    default_pulse_power: 0.7
    max_pulse_power: 1.0
  flipper_right_hold:
    number: A0-B0-1
    default_hold_power: 0.25
    max_hold_power: 0.5
  knocker:
    number: A0-B1-0
    default_pulse_ms: 20
    max_pulse_ms: 100
  pop_bumper_left:
    number: A0-B1-1
    default_pulse_ms: 18
    max_pulse_ms: 100
```

(continues on next page)

(continued from previous page)

```
ball_gate:
  number: A0-B1-2
  default_hold_power: 0.375
  max_hold_power: 0.5
```

Warning: Please ensure that you have established *common ground between logic and coil power* before turning on high voltage on your coils (especially on homebrew machines). Ignoring this might lock on your coils, overheat them, burn down your house or kill you. We are serious, floating grounds are dangerous. If you are not an electrical engineer read the *guide about voltages and power*.

In a nutshell: You need to connect your logic ground (5V/12V) and your high voltage ground (48V or 80V). A *power entry or power filter board* is a convenient solution to solve this (and more) issues.

Always turn all PSUs off when connecting power or you might fry all boards at once. This is generally a good idea but even more important when connecting more than one power supply to a board.

IF YOU DID NOT UNDERSTAND WHAT THIS WARNING MEANS STOP NOW AND TRY TO UNDERSTAND IT. OTHERWISE YOUR HARDWARE WILL LIKELY BURST INTO FLAMES AND YOU NEED TO WAIT A FEW DAYS FOR A REPLACEMENT OR EVEN WORSE IT MIGHT KILL YOU. IGNORING THIS IS THE MOST COMMON CAUSE FOR BROKEN DRIVER BOARDS.

The options are as follows:

Required settings

The following sections are required in the `coils:` section of your config:

number:

Single value, type: `string`. Defaults to empty.

This is the number of the coil which specifies which driver output the coil is physically connected to. The exact format used here will depend on which control system you're using and how the coil is connected.

See the *How to configure "number:" settings* guide for details.

Optional settings

The following sections are optional in the `coils:` section of your config. (If you don't include them, the default will be used).

allow_enable:

Single value, type: `boolean` (`true`/`false`). Default: `false`

MPF will not enable any coil at 100% power unless you also add an `allow_enable: true` entry to that coils' settings. We include this as a safety precaution since many coils will burn up if you enable them on solid, so the fact that you have to explicitly allow this for a coil prevents you from screwing something up and accidentally enabling a coil that isn't supposed to be enabled. If you have a `default_hold_power:` setting less than 8 (full power), then you don't need this `allow_enable:` entry since you are implying you want to hold the coil by adding the *default_hold_power* setting. The default default_hold_power is 100%, so if you just want to be able to enable a coil at 100% then just add `allow_enable: true` and you don't have to add a *default_hold_power* entry. If you try to enable a coil that does not have *default_hold_power* configured or *allow_enabled* set to true, then the coil will not actually be enabled and you'll get an error in your log file.

default_hold_power:

Single value, type: float(0,1). Defaults to empty.

This setting lets you control how much power is sent to the coil when it's "held" in the on position. This is an float value from 0-1 (i.e. 0% power to 100% power) which controls the relative power.

Different hardware platforms implement the hold power in different ways, so this 0-1 *default_hold_power* setting provides a generic interface that works with all hardware platforms. (You can also add platform- specific settings here for more fine-grained control of how the hold power is applied. See the How To guide for your specific hardware platform for details.) This `default_hold_power:` section is optional, and you only need it for coils you intend to hold on. In other words, if a coil is just pulsed (which is most of them), then you don't need to worry about this section.

This provides the default value for any enable calls on the coil. Devices might call enable with a different power setting.

default_pulse_ms:

Single value, type: `time string (ms) or template` (*Instructions for entering time strings* and *Instructions for entering templates*). Defaults to empty.

The default amount of time, in milliseconds, that this coil will pulse for. This can be overridden in other ways, but this is the default that will be used most of the time. Default is *10ms*, which is extremely weak, but set low for safety purposes.

default_pulse_power:

Single value, type: float(0,1). Defaults to empty.

The power factor which controls how much power is applied during the initial pulse phase of the coil's activation. (Note that not all hardware platforms support variable pulse power.) See the section on *default_hold_power:* above for details. It will also used in rules.

default_recycle:

Single value, type: `boolean` (`true`/`false`). Defaults to empty.

Controls whether this coil should add a small delay before it's allowed to be fired again. (This is used on things like pop bumpers and slingshots to prevent "machine gunning.")

This is a boolean setting because it's implemented differently depending on the hardware platform used. See the documentation for your specific hardware platform if you'd like more control than what's available with the straight on/off settings.

default_timed_enable_ms:

Single value, type: `time string (ms) or template` (*Instructions for entering time strings* and *Instructions for entering templates*). Defaults to empty.

Todo: *Help us to write it*

disable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Disables this coil (meaning that if it's active, it's shut off).

enable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Enables (holds on) this coil. This requires that *allow_enable* is true or that a *default_hold_power* or *max_hold_power* setting is configured.

max_hold_duration:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Defaults to empty.

Todo: *Help us to write it*

max_hold_power:

Single value, type: float(0,1). Defaults to empty.

This controlls the maximum allowed hold power for this this coil. While *default_hold_power* sets the default for all enable calls on the coil this defined the upper limit. If this is not set MPF will use *default_hold_power*. Usually you can omit this setting.

max_pulse_ms:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Defaults to empty.

Maximum allowed pulse time for this coil. If set, MPF will raise an error if any code tries to pulse the coil for more than `max_pulse_ms`.

max_pulse_power:

Single value, type: float(0,1). Default: `1.0`

Set the maxium pulse power. If pulse is called on the coil without any parameters *default_pulse_power* is used.

platform:

Single value, type: `string`. Defaults to empty.

Name of the platform this coil is connected to. The default value of `None` means the default hardware platform will be used. You only need to change this if you have multiple different hardware platforms in use and this coil is not connected to the default platform.

See the *Mixing-and-Matching hardware platforms* guide for details.

platform_settings:

Single value, type: dict. Defaults to empty.

Dict of platform specific settings. Consult your platform documentation for those settings.

psu:

Single value, type: string name of a *psus* device. Default: `default`

Specify to which *power supply unit* this coil is connected. This is used for power management. In some cases, MPF can deliberately delay coil pulses to prevent too many coils from firing and drawing to much current from your PSU.

pulse_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

device control events format.

Default: `None` (Note that if you add an entry here, it will replace the default. So if you also want the default value(s) to apply, add them too.)

Event(s) that pulse this coil (at its default_pulse_ms and power settings).

pulse_with_timed_enable:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Todo: *Help us to write it*

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

See the *documentation on the debug setting* for details.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

A descriptive name for this device which will show up in the service menu and reports.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Special / reserved tags for coils: *None*

See the *documentation on tags* for details.

Related How To guides

- *Coils (Solenoids)*

color_correction_profile:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

The `color_correction_profiles:` section of your *light_settings:* is where you list your color correction profiles for your lights.

Optional settings

The following sections are optional in the `color_correction_profile:` section of your config. (If you don't include them, the default will be used).

gamma:

Single value, type: `number` (will be converted to floating point). Default: `2.5`

Specifies the gamma correction value for the lights. The default is 2.5.

linear_cutoff:

Single value, type: `number` (will be converted to floating point). Default: `0.0`

This is best explained by quoting the FadeCandy documentation: By default, brightness curves are entirely nonlinear. By setting linearCutoff to a nonzero value, though, a linear area may be defined at the bottom of the brightness curve. The linear section, near zero, avoids creating very low output values that will cause distracting flicker when dithered. This isn't a problem when the lights are viewed indirectly such that the flicker is below the threshold of perception, but in cases where the flicker is a problem this linear section can eliminate it entirely at the cost of some dynamic range. To enable the linear section, set linearCutoff to some nonzero value. A good starting point is 1/256.0, corresponding to the lowest 8-bit PWM level.

linear_slope:

Single value, type: `number` (will be converted to floating point). Default: `1.0`

Specifies the slope (output / input) of the linear section of the brightness curve for the lights. The default is 1.0.

whitepoint:

List of one (or more) values, each is a type: `number` (will be converted to floating point). Default: `1.0, 1.0, 1.0`

Specifies the white point (or white balance) of your lights. Enter it as a list of three floating point values that correspond to the red, blue, and green light segments. These values are treated as multipliers to all incoming color commands. The default of *1.0, 1.0, 1.0* means that no white point adjustment is used. *1.0, 1.0, 0.8* would set the blue segment to be at 80% brightness while red and green are 100%, etc.

You can use this to affect the overall brightness of lights (e.g. `0.8, 0.8, 0.8` would be 80% brightness as every color would be multiplied by 0.8). You can also use this to affect the "tint" (lowering the blue, for example).

Related How To guides

- *lights:*
- *light_settings:*

combo_switches:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `combo_switches:` section of your config is where you configure *combo switches* which are used for things like "flipper cancel" or super skill shots where the player holds in one flipper button while hitting the launch button.

Here's an example machine config file using them:

```
#config_version=5

modes:
  - mode1

switches:
  switch1:
    number:
  switch2:
    number:
  switch3:
    number:
  switch4:
    number:
  switch5:
    number:
    tags: tag1
  switch6:
    number:
    tags: tag1
  switch7:
    number:
    tags: tag2
  switch8:
    number:
    tags: tag2
  switch9:
    number:
    tags: left_flipper
  switch10:
    number:
    tags: right_flipper

combo_switches:
  tag_combo:
    tag_1: tag1
    tag_2: tag2
  switch_combo:
    switches_1: switch1
    switches_2: switch2
  multiple_switch_combo:
```

(continues on next page)

(continued from previous page)

```
    switches_1: switch1, switch2
    switches_2: switch3, switch4
  custom_offset:
    switches_1: switch1
    switches_2: switch2
    max_offset_time: 1s
  custom_hold:
    switches_1: switch1
    switches_2: switch2
    hold_time: 1s
  custom_release:
    switches_1: switch1
    switches_2: switch2
    release_time: 1s
  custom_times_multiple_switches:
    tag_1: tag1
    tag_2: tag2
    max_offset_time: 1s
    hold_time: 1s
    release_time: 1s
    debug: true
  custom_events:
    switches_1: switch1
    switches_2: switch2
    events_when_both: active_event, active_event2
    events_when_inactive: inactive_event
    events_when_one: one_event
```

To use combo switches, add a `combo_switches:` section to either a mode or machine config. Then create subsections for each combo you want to use. (A switch can be part of more than one combo.)

The name of each combo doesn't really matter, though it's used to construct the events that are posted by this combo unless you override them.

Note about switch and tag "groups"

MPF's combo switches are meant to be used in pairs of two. (We figure that players only have two hands, so it doesn't really make sense to do combos that require three buttons to be pushed at once. Though if you want that then you can write some custom code for it.)

Usually combos would just be two switches. `left_flipper + right_flipper` or `left_flipper + launch_button`. However to give the most flexibility, you can enter your switches using either tags or switch names. It doesn't matter which you use (and you can mix-and-match if you want), the main thing is for the combo to work, you need to have at least one switch in the "1" side and one switch on the "2" side.

Note that if you have more than one switch in either group (either by specifying multiple switches for the switch config, or by using a tag that's applied to multiple switches, or both), then the combo will become active when *any* switch from either group is active. (This can be useful if you have two-stage flipper buttons where a half-push of the button controls the bottom flipper and a full push controls the top flipper. In that case you technically have two switches per flipper button and you can add both to each group in your combo.)

Optional settings

The following sections are optional in the `combo_switches:` section of your config. (If you don't include them, the default will be used).

events_when_both:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

This is an event (or a list of events) that will be posted when both switches are held in. If you have a `max_offset_time:` configured, then both switches will need to have been pressed within that time. If you have a `hold_time:` configured, then both switches will need to be active for at least that long before this event (or these events) are posted.

If the player pushes both switches, then releases one, then pushes in the switch that was released again, this event will be re-posted.

If you don't set this value, then a default event with the name of your combo plus `_both` will be used.

events_when_inactive:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

This is an event (or list of events) that will be posted when the player releases both of the buttons, essentially "releasing" the combo.

If you don't set this value, then a default event with the name of your combo plus `_inactive` will be used.

events_when_one:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

This is an event (or list of events) that will be posted when the player releases one switch after both switches have been pressed together. (In other words, this event will only be posted after the `events_when_both` event is posted.)

If you don't set this value, then a default event with the name of your combo plus `_one` will be used.

events_when_switches_1:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

This is event (or list of events) will be posted when only switches from `switches_1` were active for `max_offset_time`.

events_when_switches_2:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

This is event (or list of events) will be posted when only switches from `switches_2` were active for `max_offset_time`.

hold_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `0`

How long each button has to be pressed in order for it to count as a combo. The default is `0` which means that as soon as both switches are active, the combo is active.

If you set `hold_time: 1s`, that means that the player will have to press and hold both buttons for 1 second before the combo's "both" event is posted.

max_offset_time:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `-1`

Specifies a time window that a switch from group 1 and group 2 have to be hit within in order to register as a combo.

The default value of `-1` means there is no time limit, meaning that the player can hit and hold one button, and then five minutes later hit the next button, and the combo will count.

If you set `max_offset_time: 1s`, that means that the player will have to hit (and hold) both switches within 1 second of each other.

release_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `0`

How long a button has to be released before the combo will switch from "both" state to the "one" state. The default is `0` which means this is instant.

Note that once both buttons are released, the combo is cleared. This setting only affects the scenario when one button is held in while the other is released.

switches_1:

List of one (or more) values, each is a type: string name of a *switches* device. Defaults to empty.

A switch name (or a list of switches) that will be used for the group 1 of the combo. You can use this setting or the `tag_1:` setting above.

switches_2:

List of one (or more) values, each is a type: string name of a *switches* device. Defaults to empty.

A switch name (or a list of switches) that will be used for the group 1 of the combo. You can use this setting or the `tag_2:` setting above.

tag_1:

List of one (or more) values, each is a type: `string`. Defaults to empty.

A tag (or list of tags) of switches (in the `switches:` section of your machine config that will be used for switches for group 1 of the combo. You can either use a tag, or use the `switches_1:` setting (or both, really).

tag_2:

List of one (or more) values, each is a type: `string`. Defaults to empty.

A tag (or list of tags) of switches (in the `switches:` section of your machine config that will be used for switches for group 2 of the combo. You can either use a tag, or use the `switches_2:` setting (or both).

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Not used.

Related How To guides

- *Combo Switches ("flipper cancel", etc.)*

config:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `config:` section of your configuration files allows you to specify *additional* configuration files that will be read in after the current file is loaded. Here's an example:

```
config:
    - machine.yaml
    - devices.yaml
    - game.yaml
    - textstrings.yaml
    - keymap.yaml
```

Note that each file is on its own line, which starts with a minus, then a space, then the file. (The space is important.) Also note that you can (optionally) specify a path, like this:

```
- config\machine.yaml
- config/my_game/machine.yaml
```

MPF will attempt to convert relative and absolute paths based on your OS, and it can deal with slashes in either direction.

MPF will then open those files one-by-one and merge their settings into the master configuration dictionary. The settings are merged together in the order the files are listed, so if multiple files specify the same configuration option then whichever one comes later in the list will overwrite any options that have already been specified.

You can also have `config:` sections in other config files, meaning that one config file can call another which will call another, etc.

Whenever MPF encounters a new config file, it will add it to the end of the list. And since files are processed in order, if there are any conflicting settings then the last file on the list will "win." Also note that the framework will attempt to load the file from the current working directory (containing the config file that `config:` entry is from. If that fails then it will try the last known good directory that worked for a config file.

Related How To guides

- *Layering Modes Example*

counters:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The counters: section of your config is where you configure counter logic blocks. See also *counters*. The structure of counter logic blocks is like this:

```
##! mode: mode1
counters:
  the_name_of_this_counter:
    count_events: my_count_event
    count_complete_value: 10
  some_other_counter:
    count_events: s_my_switch_active
    starting_count: 50
    count_interval: 10
    count_complete_value: 100
```

Note that the actual name of the counter doesn't really matter. Mainly it's used in the logs and for event names.

Counters no longer save their state in player variables. If you are using something like (YOUR_COUNTER_count) in a slide or widget you can use a *variable_player* to restore the old behaviour:

```
##! mode: my_mode
variable_player:
  logicblock_YOUR_COUNTER_updated:
    YOUR_COUNTER_count:
      int: value
      action: set
```

Required settings

The following sections are required in the counters: section of your config:

count_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

This is an event (or a *list of events*) that, when posted, will increment or decrement the count for this Counter.

Note that if you include multiple events in this list, *any* one of the events being posted will cause the hit count to increase. If you want to track different kinds of events separately, use an *Accrual* or *Sequence* Logic Block instead.

This setting is required.

Optional settings

The following sections are optional in the counters: section of your config. (If you don't include them, the default will be used).

control_events:

List of one (or more) values, each is a type: *counter_control_events*. Defaults to empty.

Control events to change the value of this counter. MPF currently supports adding/substracting from the count or jumping to a certain value.

For instance in the following example `add_five_event` will add 5 to the counter:

```
counters:
  counter_with_control_events:
    count_events: count_up
    control_events:
      - event: add_five_event
        action: add
        value: 5
```

count_complete_value:

Single value, type: `integer` or `template` (*Instructions for entering templates*). Defaults to empty.

When the Counter exceeds (or gets below if you're counting down) this value, it will post its "complete" event and be considered complete.

count_interval:

Single value, type: `integer`. Default: `1`

Specifies the numeric count change is for each hit. In other words, this is how much is added or removed from the count with each hit. Default is 1, but you can make it whatever you want if you want your count to increase by more or less than one whenever a count event occurs. You could use this, for example, in a mode to create a counter that tracks the value of a shot. Maybe it starts at 2,000,000, but each shot a playfield standup increases the value by 250,000.

Default is `1`.

direction:

Single value, type: one of the following options: up, down. Default: `up`

This is either `up` or `down` and specifies whether this counter counts up or counts down.

Default is `up`.

multiple_hit_window:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `0`

This is an *MPF time value string* that will be used to group together multiple *count_events* as if they were one single event. So if you have `multiple_hit_window: 500ms` and you get three hit events 100ms apart, they will all count as one hit.

Note that subsequent hits that come in during the time window do not extend the time. So with the 500ms hit_window from above, the first hit counts and sets the timer, another hit 300ms later won't count, but a third hit 300ms after the second (and 600ms after the initial hit) will count (and it will set its own 500ms timer to ignore future hits).

Default is `0` (which means all hits are counted).

starting_count:

Single value, type: `integer` or `template` (*Instructions for entering templates*). Default: `0`

This is the starting value of the Counter and the value it goes back to when it's reset. Default is zero. If you're configuring a counter with `direction: down`, you'll want to also set this to something more than zero.

Default is `0`.

Note that you can use a *dynamic value* for this setting.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Currently unused.

Optional settings

The following sections are optional in the `logic_blocks_common:` section of your config. (If you don't include them, the default will be used).

disable_events:

List of one (or more) device control events (*Instructions for entering device control events*).

Event(s) that will disable this logic block.

A logic block must be enabled to track hits, progress, and to post events.

disable_on_complete:

Single value, type: `boolean` (`true`/`false`). Default: `true`

True/False (or Yes/No) which controls whether this logic block disables itself once it completes. This does not reset the current value.

enable_events:

List of one (or more) device control events (*Instructions for entering device control events*).

Event(s) that will enable this logic block.

A logic block must be enabled to track hits, progress, and to post events.

If you don't have any enable_events listed, then the logic block will automatically be enabled when the player's ball starts.

events_when_complete:

List of one (or more) events.

Events that will be posted when this device is completed.

events_when_hit:

List of one (or more) events.

Events that will be posted when this device is hit or advanced.

persist_state:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Boolean setting (yes/no or true/false) which controls whether this logic block remembers where it was from ball-to-ball. If `False`, then this logic block will reset itself whenever a new ball starts. If `True`, then this logic block will be saved to the player variable *<logic_block_name>_state*.

Note that logic block state is reset on mode end when this is `False` and, as normal modes stop at the end of a ball, the state is always maintained on a per-player basis, regardless of what this setting is configured for.

reset_events:

List of one (or more) device control events (*Instructions for entering device control events*).

Event(s) that will reset this logic block back to its original value. This has no effect on the enabled/disabled state of the block.

Note that there are also `reset_on_complete:` and `persist_state:` settings which also affect how and when the logic block is reset.

You can reset a logic block regardless of whether it's enabled.

reset_on_complete:

Single value, type: `boolean` (`true`/`false`). Default: `true`

True/False (or Yes/No) which controls whether this logic block resets itself once it completes. This just resets the current value or progress. It does not change the enabled or disabled state.

Note, `disable_on_complete` default is `true`, which may seem like reset isn't working. For something like a counter that automatically starts again change `disable_on_complete` to `false`.

restart_events:

List of one (or more) device control events (*Instructions for entering device control events*).

List of one (or more) events which, when posted, will restart this logic block. A restart is a reset, then an enable, combined into a single action.

start_enabled:

Single value, type: `boolean` (`true`/`false`).

If `true` this device will start enabled. If `false` this device will start disabled. If you omit this the device will start enabled unless you specify `enable_events` in which case the device will start disabled.

Related How To guides

- *Counter Logic Blocks*
- *Integrating Logic_Blocks and Shows*

counter_control_events:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

Counter can contain `control_events:` which can add or substract to the count of your counter. Alternatively, you can set the counter to a certain value using an event.

Required settings

The following sections are required in the `counter_control_events:` section of your config:

action:

Single value, type: one of the following options: add, subtract, jump.

`add` will add `value` to the current count of your counter. `subtract` will subtract `value` from the current count of your counter. `jump` will set your counter to `value`.

event:

Single value, type: `string`.

The event to trigger the `action`.

Optional settings

The following sections are optional in the `counter_control_events:` section of your config. (If you don't include them, the default will be used).

value:

Single value, type: `integer` or `template` (*Instructions for entering templates*).

The value to use in `action`.

Related How To guides

- *counters:*

credits:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `credits:` section of your config contains settings for the credits mode.

There's a full How To guide which walks you through setting up the credits mode, so be sure to read that for the details. This page just contains the settings which control how the credits mode behaves. Here's an example config:

```
credits:
  max_credits: 12
  free_play: false
  price_tier_template: "{{credits}} CREDITS ${{price}}"
  service_credits_switch: s_esc
  switches:
    - switch: s_left_coin
      type: dollars
      value: .25
    - switch: s_right_coin
      type: dollars
      value: 1
  pricing_tiers:
    - price: .50
      credits: 1
    - price: 2
      credits: 5
  events:
    - event: special
      type: special
      credits: 1
    - event: replay
      type: replay
      credits: 1
    - event: high_score_credit
      type: high_score
      credits: 1
    - event: match
      type: match
      credits: 1
  fractional_credit_expiration_time: 15m
  credit_expiration_time: 2h
  persist_credits_while_off_time: 1h
  free_play_string: FREE PLAY
  credits_string: CREDITS
```

Optional settings

The following sections are optional in the `credits:` section of your config. (If you don't include them, the default will be used).

credit_expiration_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings)* . Default: `0`

The amount of time before any credits on the machine are removed (resetting the number of credits back to 0). This timer only runs while the machine is in attract mode, and its reset each time a new credit (or partial credit) is added to the machine. If a game is played, the timer starts fresh when the game is over and the machine goes back to attract mode. This value is entered as a standard MPF time string and can be minutes, hours, or even days long. Default is *2 hours*.

credits_string:

Single value, type: `string`. Default: `CREDITS`

This is the text that will make up the credits_string before the number of credits. For example, if there are 2 1/2 credits on the machine, the credits_string will be *CREDITS 2 1/2*. Default is *CREDITS*.

fractional_credit_expiration_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings)* . Default: `0`

The amount of time before fractions of credits are removed from the machine. This doesn't affect whole credits, so if the machine is sitting there with 2 1/4 credits on it, after this time expires MPF will clear the 1/4 credit leaving 2 whole credits. This timer only runs while the machine is in attract mode, and its reset each time a new credit (or partial credit) is added to the machine. If a game is played, the timer starts fresh when the game is over and the machine goes back to attract mode. This value is entered as a standard MPF time string and can be minutes, hours, or even days long. Default is *15 minutes*.

free_play:

Single value, type: `boolean` (Yes/No or True/False). Default: `yes`

Controls whether the machine is in free play mode. Note that if you want your machine to always be in free play mode, then you can also choose to not use the credits mode altogether.

free_play_string:

Single value, type: `string`. Default: `FREE PLAY`

The text string that will be used in the credits_string machine variable when the machine is in free play. Default is *FREE PLAY*.

max_credits:

Single value, type: `integer`. Default: `0`

The maximum number of credits you want to allow on the machine. Note that pinball machines can't prevent players from adding money to machines, so be careful with this.

Also note that you can use *dynamic values* here if you want to do math or use settings to make this configurable.

persist_credits_while_off_time:

Single value, type: `time string (secs)` (*Instructions for entering time strings)* . Default: `1h`

The amount of time that credits will remain on the machine even when MPF is not running. Set to 0 if you do not want to MPF to retain credits when its powered off. The way this works behind the scenes is that whenever a new credit (or a fraction of a credit) is added to the machine, MPF writes that to disk as a persistent machine variable with an expiration time and date based on the current time plus the delay time you add here. When MPF boots up, it loads the credits from the machine variables file and checks their expiration time, and if it's in the past then it doesn't add them back. This value is entered as a standard MPF time string and can be minutes, hours, or even days long. Default is *1 hour*.

service_credits_switch:

List of one (or more) values, each is a type: string name of a `switches:` device. Default: `None`

This is the name of a switch that's used to add so-called "service credits" to the machine. This switch has a 1-to-1 ratio, meaning that one credit is added to the machine each time this switch is pressed.

switches:

The `switches:` section contains the following nested sub-settings.

A list of switches that, when triggered, add credits (or fractions of a credit) to the machine. Notice that the sub-entries under switches are actually a list with the settings for *switch*, *type*, and *value*, repeated multiple times.

Optional settings

The following sections are optional in the `switches:` section of your config. (If you don't include them, the default will be used).

switch:

Single value, type: string name of a `switches:` device. Default: `None`

The name of the switch (from your machine-wide *switches:* section) for the credit switch.

type:

Single value, type: `string`. Default: `money`

What type of currency is being deposited when that switch is hit. This doesn't affect the actual behavior of MPF, rather it's just used in as the column name and for totaling the earnings reports (so you can track "money" separate from "tokens"). You can enter whatever you want here: *money*, *dollars*, *dinars*, etc.

value:

Single value, type: `number` (will be converted to floating point). Default: `0.25`

How much value is added whenever this switch is hit. Notice that there are no currency symbols here or anything. A value of .25 could be 0.25 dollars or 0.25 Euros or 0.25 Francs—it really doesn't matter. The key is that it's 0.25 of whatever monetary system you have.

Also note that you can use *dynamic values* here if you want to do math or use settings to make this configurable.

price_tier_template

Default "{{credits}} CREDITS ${{price}}"

Placeholder to generate the credits string.

pricing_tiers:

The `pricing_tiers:` section contains the following nested sub-settings.

This is where you actually set your pricing by mapping how many of your monetary units you want to equate to a certain number of credits. The default config is fairly common, with 0.50 currency resulting in 1 credit, with a price break at 2 that gives the player 5 credits instead of 4. (So basically they get one free credit if they put in enough money for 4 credits.) The most important thing to know here is that MPF always requires that 1 credit is used to start a game, and 1 credit is required to add an additional player to a game. So if you want to change the price of your game, you don't change the number of credits per game, rather, you change the number of credits a certain amount of money is worth. The pricing tier discount processing is reset when Ball 2 starts. So if it costs $0.50 for one credit or $2 for 5 credits, if the player puts $0.50 in the machine and plays a game, if they wait until that game is over and deposit another $1.50, they'll only get 3 more credits. You can have as many *pricing_tiers* as you want. The first one dictates how much a regular game costs and is required. If you don't want any price breaks, then just add the first one.

Here's an example:

```
credits:
   # ...
  pricing_tiers:
    - price: .50
      credits: 1
    - price: 2
      credits: 5
```

price:

Price for number of `credits`.

Also note that you can use *dynamic values* here if you want to do math or use settings to make this configurable.

Optional settings

The following sections are optional in the `pricing_tiers:` section of your config. (If you don't include them, the default will be used).

credits:

Single value, type: `integer`. Default: `1`

The total number of credits that will be added based on this price tier

price:

Single value, type: `number` (will be converted to floating point). Default: `.50`

The numeric currency value for this pricing tier.

events:

A list of one or more events with settings which add credits based on MPF events. Like the pricing_tiers section, start each entry here with a minus sign and a space.

```
credits:
   # ...
  events:
    - event: special
      type: special
      credits: 1
    - event: replay
      type: replay
      credits: 1
    - event: high_score_credit
      type: high_score
      credits: 1
    - event: match
      type: match
      credits: 1
```

event:

The event that will trigger a credit action.

type:

String which can be whatever you want, used for audits. This lets you track different types of credits, for example, money in versus replays versus specials versus high score awards, etc.

award:

Numeric value of the number of credits you'd like to award.

custom_code:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `custom_code:` section of your config is a list where you register your custom code classes. You can find an example here: *custom_code (example config files)*.

Related How To guides

- MPF developer documentation.

display_light_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

Note: This section can also be used in a show file in the `display_lights:` section of a step.

The `display_light_player:` section of your config is where you use your lights as a display. See *Using LEDs as display (display_light_player)* for details.

Optional settings

The following sections are optional in the `display_light_player:` section of your config. (If you don't include them, the default will be used).

action:

Single value, type: one of the following options: play, stop. Default: `play`

Play or stop the display.

bcp_connection:

Single value, type: `string`. Default: `local_display`

Which BCP client provides the content for your display. You can usually leave this at the default.

lights:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Which LEDs should receive the updates. You usually use a tag here or '*' for all of them.

max_x:

Single value, type: `integer`. Defaults to empty.

Unused.

max_y:

Single value, type: `integer`. Defaults to empty.

Unused.

min_x:

Single value, type: `integer`. Default: `0`

Unused.

min_y:

Single value, type: `integer`. Default: `0`

Unused.

Related How To guides

- *Using LEDs as display (display_light_player)*

displays:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `displays:` section of your config is where you configure the logical displays in your machine. A display is used to show slides, and can be an on-screen window or a DMD.

You can have more than one display. For example, if you want to have a DMD and also display an on-screen window, you'll actually have two `displays:`, the DMD is one and the on-screen window is the other.

Here's an example `displays:` section from *Demo Man* with two displays:

```
displays:
  window:
    height: 200
    width: 600
  dmd:
    width: 128
    height: 32
    default: true
    round_anchor_x: left
```

In the example above, one of the displays is called *window* and the other is called *dmd*. Note that the names here are completely arbitrary. Just naming a display "window" does not make it show up in the window, and naming a display "dmd" doesn't make it show up in the DMD. (When you configure your window in the `window:` section of your config, you specify the name of the display you want to be the *source* for the window content. Same for the DMD.)

The names of the displays are used as "targets" for your slides. So when you show a slide, you specify which display you want it to show on. If you don't specify a target, it will choose the default. If you only have one display, you never have to worry about this because that display will always be the default. If you have more than one, you can add the `default: true` to a display here to tell MPF which display is your default which is used when you play slides without specifying a target.

Note: Starting in MPF v0.33, If you do not put a `displays:` section in your machine config, MPF will automatically create a single display called "default" with a size of 800x600. (This matches the default window size.)

Each display in your `displays:` section can have the following settings:

Optional settings

The following sections are optional in the `displays:` section of your config. (If you don't include them, the default will be used).

default:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Specifies that this display is the default, meaning it's the display that's used if you show a slide without specifying a target for that slide. If you only have one display, it will be the default automatically.

enabled:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Whether this display is enabled. If false, all slide and widget player calls targeting this display will be ignored.

height:

Single value, type: `integer`. Default: `600`

The height if the display, in pixels. Note that if you're showing this display on the screen, you can scale the screen window which will scale the display. So the height here can be thought of as the "native" height of the display.

round_anchor_x:

Single value, type: `string`. Default: `center`

Indicates that this display should not render widgets on fractional horizontal pixels, e.g. anchoring an 11px-wide widget at -5.5 pixels. When specified with `left` or `right`, this display will round the pixel position to the nearest whole pixel in that direction.

This setting can also be configured on an individual widget to override the display's configuration.

round_anchor_y:

Single value, type: `string`. Default: `middle`

Indicates that this display should not render widgets on fractional vertical pixels, e.g. anchoring an 11px-high widget at -5.5 pixels. When specified with `bottom` or `top`, this display will round the pixel position to the nearest whole pixel in that direction.

This setting can also be configured on an individual widget to override the display's configuration.

width:

Single value, type: `integer`. Default: `800`

The width of the display, in pixels.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Unused.

Related How To guides

- *Displays, DMDs, & Graphics*

digital_outputs:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `digital_outputs:` section of your config is where you configure digital outputs. Those can be either mapped to a light or a driver and support only enabling and diabling. In contrast to a light, `digital_outputs` do not support any fading or pwm/brightness. Opposed to drivers, `digital_outputs` do not support pulsing, pattern or hardware rules. Use them to control digital logic. MPF uses them to control *motors* with additional control logic.

Some platforms such as Stern Spike, Gottlieb System 1 or Gottlieb System 80 use lights outputs to control logic. In other platforms you usually use drivers.

Required settings

The following sections are required in the `digital_outputs:` section of your config:

number:

Single value, type: `string`. Defaults to empty.

The number of your light or driver. The exact meaning of this number depends on your platform but is exactly the same as if this was a light or driver (depending on the `type` setting).

type:

Single value, type: one of the following options: light, driver. Defaults to empty.

Whether this output is mapped as light or driver.

Optional settings

The following sections are optional in the `digital_outputs:` section of your config. (If you don't include them, the default will be used).

disable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Those events will disable this output when posted.

enable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Those events will enable this output when posted.

light_subtype:

Single value, type: `string`. Defaults to empty.

If this is mapped as light (`type: light`) you can set the `subtype` here (see *lights* for details about `subtype`). The exact meaning depends on your platform.

platform:

Single value, type: `string`. Defaults to empty.

In case you want to overwrite the default platform (as defined in `hardware:`), you can choose a platform for this output.

platform_settings:

Single value, type: dict. Defaults to empty.

Dict of platform specific settings. Consult your platform documentation for those settings.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Not used.

Related How To guides

- *Motors*
- *How to use Step Stick Steppers in MPF*
- *Using the Stern Spike Trough*
- *Configuring and Enabling Flippers/Pop Bumpers/Slingshots in LISY*
- *Arduino Pinball Controller*
- *How to configure coils/drivers/magnets (P-ROC/P3-ROC)*

diverters:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

You create and configure your diverters in the *diverters:* section of your machine configuration file. Here's an example from *Star Trek: The Next Generation*:

```
diverters:
  top_diverter:
    activation_coil: c_top_divertor      # WMS uses the -tor spelling
    type: hold
    activation_time: 3s
    activation_switches: s_enter_left_ramp
    enable_events: ball_started
    disable_events: ball_ended, borg_lock_Lit
    targets_when_active: playfield
    targets_when_inactive: bd_borg_ship
  subway_top_diverter:
    activation_coil: c_under_divertor_top
    type: hold
    activation_time: 3s
    activation_switches: s_under_top_hole, s_under_left_hole, s_under_borg_hole
    targets_when_active: bd_left_cannon_vuk
    targets_when_inactive: bd_left_vuk
    feeder_devices: bd_catapult
  subway_bottom_diverter:
    activation_coil: c_under_divertor_bottom
    type: hold
    activation_time: 3s
    activation_switches: s_under_top_hole, s_under_ueft_hole, s_under_borg_hole
    targets_when_active: bd_left_cannon_vuk
    targets_when_inactive: bd_left_vuk
    feeder_devices: bd_catapult
  drop_target:
    activation_coil: c_top_drop_down
    deactivation_coil: c_top_drop_up
    type: pulse
    targets_when_active: bd_left_cannon_vuk, bd_right_cannon_vuk, bd_left_vuk
    targets_when_inactive: playfield
    feeder_devices: bd_catapult
```

Optional settings

The following sections are optional in the `diverters:` section of your config. (If you don't include them, the default will be used).

activate_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Events in this list, when posted, cause this diverter to activate.

activation_coil:

Single value, type: string name of a *coils* device. Defaults to empty.

The name of the coil that is used to activate your diverter.

activation_switches:

List of one (or more) values, each is a type: string name of a *switches* device. Defaults to empty.

A list of one or more switches that trigger the diverter to activate. This switch only activates the diverter if the diverter has been enabled (either manually or via one of the *enable_events*. If you have an activation switch, MPF writes a hardware autofire coil rule to the pinball controller which fires the diverter automatically when the *activation_switch* is hit. This is done so the diverter will have instantaneous response time, needed to get the diverter to fire in time to catch a fast-moving ball.

activation_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `0`

This is how long the diverter stays active once it's been activated. A value of zero (or omitting this setting) means this diverter does not timeout, and it will stay active until it's disabled or you manually deactivate it.

allow_multiple_concurrent_ejects_to_same_side:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Todo: *Help us to write it*

ball_search_hold_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `1s`

How long this diverter will be activated for when it is activated during ball search.

ball_search_order:

Single value, type: `integer`. Default: `100`

A relative value which controls the order individual devices are pulsed when ball search is running. Lower numbers are checked first. Set to `0` if you do not want this device to be included in the ball search. See the *Ball Search* documentation for details.

cool_down_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `0`

How long does the diverter need to cool down until the next eject can happen into the diverter?

deactivate_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, cause this diverter to deactivate.

deactivation_coil:

Single value, type: string name of a *coils* device. Defaults to empty.

The name of the coil that's used to deactivate your diverter. You only need to specify this coil if it's a different coil from from *activation_coil*. (In other words this is only used with diverters that have two coils.)

An example of this is when a drop target is used to block the entrance of a ball device. (For example, the drop target under the saucer in *Attack from Mars*, the drop target to the left of the upper lanes in *Star Trek: The Next Generation*, or the middle letter "D" drop target in *Judge Dredd*.) Each of these has one coil to "knock down" the drop target and a second coil to "reset" the drop target.

By the way, if you have two coils to control a diverter, it doesn't really matter which one is the *activation_coil* and which is the *deactivation_coil*. Just know that after the *activation_coil* is fired, MPF will consider that diverter to be in the active state, and once the *deactivation_coil* is fired, MPF will consider that diverter to be in the inactive state, and set up your targets accordingly.

deactivation_switches:

List of one (or more) values, each is a type: string name of a *switches* device. Defaults to empty.

A list of one or more switches that will deactivate a diverter. (For example, this might be a switch that's "after" the diverter in a subway, so once this switch is activated then MPF knows the ball made it through the diverter and it can deactivate it.)

disable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, disable this diverter. Typically it's *ball_ending* (which is posted when a ball is in the process of ending), meaning this diverter will not be enabled when the next ball is started. You might also set a disable event to occur based on the event posted from a mode ending.

disable_switches:

List of one (or more) values, each is a type: string name of a *switches* device. Defaults to empty.

A list of one more more switches that will automatically disable this diverter. It's optional, since the diverter will also be disabled based on one of your *disable_events* being posted.

enable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, enable this diverter. (Remember that enabling a diverter is not the same as activating it.)

feeder_devices:

List of one (or more) values, each is a type: string name of a *ball_devices* device. Default: `playfield`

This is a list of one or more ball devices that can eject balls which have the option of being sent to this diverter. This is an important part of the diverter's ability to automatically route balls to the devices they go to.

When you configure a *feeder_device:* setting for a diverter, it causes the diverter to watch for balls ejecting from that device. Every ball that's ejected in MPF has a "target" (either a ball device or the playfield), so when a diverter's feeder device ejects a ball, the diverter will see what the eject target is, and if that target is included in the diverter's list of *targets_when_active* or *targets_when_inactive*, then the diverter will activate or deactivate itself to make sure the balls gets to where it needs to go.

playfield:

Single value, type: string name of a *playfields* device. Default: `playfield`

The name of the playfield that this diverter is on. The default setting is "playfield", so you only have to change this value if you have more than one playfield and you're managing them separately.

reset_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `machine_reset_phase_3`

Reset will disable the diverter.

targets_when_active:

List of one (or more) values, each is a type: string name of a *ball_devices* device. Default: `playfield`

This is a list of *all* ball devices that can be reached by a ball passing through this diverter when it's active. Valid options include the names of ball devices and the word "playfield."

This setting exists because diverters in MPF can be configured so that they automatically activate or deactivate when one of their target devices wants a ball. For example, if you have a diverter on a ramp that will route a ball to a lock when its active, you can add the name of that ball device here. Then if that device ever needs a ball, the diverter will automatically activate to send a ball there. This greatly simplifies programming, because all you have to do is essentially say, "I want this device to have a ball," and MPF will make sure the diverter sets itself appropriately to get a ball to that device.

Let's look at the diverter configuration from *Star Trek: The Next Generation* included at the top of this section for an example. In the settings for the *dropTarget* diverter, notice that there are three items in the *targets_when_active:* list: *bd_leftCannonVUK*, *bd_rightCannonVUK*, and *bd_leftVUK*. This means that when this diverter is active, balls passing through it are able to reach any one of those three ball devices. Note that this particular diverter doesn't exactly know how the ball gets to any of those devices—that's actually handled via additional downstream diverters (*subwayTopDiverter* and *subwayBottomDiverter*). All the *dropTarget* diverter needs to know is, "If a ball needs to go to one of these three diverters, then I better be active."

targets_when_inactive:

List of one (or more) values, each is a type: string name of a *ball_devices* device. Default: `playfield`

This is exactly like the *target_when_active:*above, except it represents the target devices that a ball can reach when this diverter is disabled. Looking at the same *dropTarget* diverter example from above, we see that when the *dropTarget* is inactive, the ball is routed to the playfield.

type:

Single value, type: one of the following options: hold, pulse. Default: `hold`

Specifies how the *activation_coil* should be activated. You have two options here:

- `pulse` - MPF will pulse the coil to activate the diverter.
- `hold` - MPF should hold the diverter coil in a constant state of "on" when the diverter is active. Note that if the coil is configured with a *default_hold_power*, then it will use that pwm pattern to hold the coil on. If no *default_hold_power* is configured, then MPF will use a continuous enable to hold the coil. (In this case you would need to add *allow_enable: true* or *max_hold_power* to that coil's configuration in the *coils:* section of your machine configuration file.)

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to `True` to see more debug output.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Tags are currently unused.

Related How To guides

- *Diverters*

dmds:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `dmds:` section of your config is where you configure the settings for physical DMDs (dot matrix displays). You only need this section if you have a physical monochrome DMD connected to a 14-pin header on a hardware controller. If you have an RGB DMD, configure that in the *rgb_dmds:* section.

If you want to show a virtual DMD in an on-screen window, you configure that as a display widget with a dot filter. That does not involve this `dmds:` section.

Note that there are no *height* and *width* settings here. The pixel size of your DMD is determined by the size of the `source:` display which drives the content for this DMD.

```
displays:
  dmd:
    width: 128
    height: 32
dmds:
  my_dmd:  # name of this DMD which can be whatever you want
    brightness: .5
    fps: 25
    gamma: 2.5
```

Note that this section is called `dmds:` (plural). Just like "switches" and "coils" and most everything else in MPF, this is a section that contains all your DMDs. Now since this is a DMD, you probably only have one, (though MPF can support as many as you want), but it's important to note that you add a `dmds:` section to your config, then under that you add an entry for a specific DMD (which can be whatever you want), and then you enter one or more of the following settings:

(If you don't include any of the settings below, the default will be used).

Optional settings

The following sections are optional in the `dmds:` section of your config. (If you don't include them, the default will be used).

brightness:

Single value, type: `number` (will be converted to floating point). Default: `1.0`

A brightness multiplier for the DMD. Default is 1.0 which is full brightness, but if you want to dim the DMD, you can set this to some value lower than 1.0. (e.g. a value of 0.9 will be 90% brightness, etc.)

fps:

Single value, type: `integer`. Default: `30`

How many frames per second this DMD will be updated. A value of 30 should be fine and smooth. Some people claim that higher values look better, but as far as we can tell, that just makes your CPU work harder. But feel free to experiment.

gamma:

Single value, type: `number` (will be converted to floating point). Default: `1.0`

Sets the gamma of the DMD. See *Gamma correction in MPF* for details.

Note that the default setting of `1.0` means that no gamma correction is used. Some physical DMDs do their own internal gamma correction, so this setting is fine. Others require pre-corrected gamma, so you can set that value here.

You might try a value of 2.2 first and adjust up or down until it looks right.

Important: Gamma setting is important!

We can't stress enough that setting the gamma for your DMD is important for making it look right. So click the link above and make the adjustment. It's a one-time thing.

luminosity:

List of one (or more) values, each is a type: `number` (will be converted to floating point). Default: `.299, .587, .114`

A list of three values (from 0.0 to 1.0) that represent the percentage of red, green, and blue that will be used to produce the monochrome colors from the source display. All three of these values should add up to 1.0.

only_send_changes:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Specifies whether every frame is sent to the DMD, or only changed frames.

platform:

Single value, type: `string`. Defaults to empty.

Name of the platform this DMD is connected to. The default value of `None` means the default hardware platform will be used. You only need to change this if you have multiple different hardware platforms in use and this coil is not connected to the default platform.

See the *Mixing-and-Matching hardware platforms* guide for details.

shades:

Single value, type: `integer` (must be a power of 2). Default: `16`

How many shades the physical DMD can show. Modern pinball controllers support 16 shades.

source_display:

Single value, type: `string`. Default: `dmd`

The name of the display (from the `displays:` section of your machine config) that is the source for this physical DMD. Whatever's on the source display will be displayed on the DMD. If you don't specify a source, MPF will automatically use a source display called "dmd".

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Not used.

Related How To guides

- *Using a traditional (single color) physical DMD*

drop_target_banks:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

Once you've configured your individual drop targets, you group them together into banks via the `drop_target_banks:` section of your config file. Here's an example from *Judge Dredd*:

```
drop_target_banks:
  judge:
    drop_targets: j, u, d, g, e
    reset_coils: reset_drop_targets
    reset_on_complete: 1s
```

Notice there are no settings to control lights associated with drop targets, but many machines (like *Judge Dredd* used in the example) have lights for each drop target. To control those lights, you'd create shots based on the lights and switches for each drop target, and then you control them just like any other shot with the shot settings, shot_group settings, and shot profiles. In this case you'd end up specifying your switch for this drop target as well as for a shot for it. It's ok to have the same switch in both places.

Create a subsection under *drop_target_banks:* for each bank of drop targets you have. The name of each section is the name you'll refer to the drop target as in your game code. ("judge", in this example.)

Required settings

The following sections are required in the `drop_target_banks:` section of your config:

drop_targets:

List of one (or more) values, each is a type: string name of a *drop_targets* device. Defaults to empty.

A list of the names of the individual drop targets (from the names you chose in the *drop_targets:* section of your config file) that are included in this bank. Note that single drop target devices can be members of multiple banks at the same time. For example, you might have two banks of three drop targets, from which you could actually actually three drop target banks. One for the first three, one for the second three, and one for all six. Then you could track separate up and down events for a subset of three or for all six getting knocked down.

Optional settings

The following sections are optional in the `drop_target_banks:` section of your config. (If you don't include them, the default will be used).

ignore_switch_ms:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `500ms`

How long this device should ignore switch changes while ball search is running. (Otherwise the ball search pulsing coils will set switches that could add to the score, start modes, etc.

max_reset_attempts:

Single value, type: `integer`. Defaults to empty.

Todo: *Help us to write it*

reset_coil:

Single value, type: string name of a *coils* device. Defaults to empty.

The name of the coil that is fired to reset this bank of drop targets.

reset_coil_max_wait_ms:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `100ms`

Max time allowed to delay the pulse of the reset coil. This is used to prevent excess power usage. See *psus:* for details.

reset_coils:

List of one (or more) values, each is a type: string name of a *coils* device. Defaults to empty.

If your drop target bank has two reset coils (as was common in older machines which huge banks of drop targets), you can add a *reset_coils* section (plural) and then specific a list of multiple coils. In this case, MPF will pulse all the coils at the same time to reset the bank of drop targets.

reset_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `machine_reset_phase_3, ball_starting`

Resets this drop target bank by pulsing this bank's `reset_coil` or `reset_coils`.

reset_on_complete:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Defaults to empty.

By default, when a drop target bank completes, it does not automatically reset. If you want it to reset, then use this setting along with a time delay for when you want it to reset after it completes.

For example:

```
reset_on_complete: 500ms
```

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

See the *documentation on the debug setting* for details.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

A descriptive name for this device which will show up in the service menu and reports.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Special / reserved tags for drop target banks: *None*

See the *documentation on tags* for details.

Related How To guides

- *Drop Target Bank*

drop_targets:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

You configure individual *drop targets* in your machine in the `drop_targets:` section of your machine config file. (This section is only used for individual targets. Once you configure them here, then you group them into banks in the `drop_target_banks:` section.) Here's an example from *Judge Dredd*, with five drop targets we've given names *J*, *U*, *D*, *G*, and *E*.

```
drop_targets:
  j:
    switch: drop_target_j
    reset_coil: reset_drop_targets
  u:
    switch: drop_target_u
    reset_coil: reset_drop_targets
  d:
    switch: drop_target_d
    reset_coil: reset_drop_targets
    knockdown_coil: trip_drop_target_d
  g:
    switch: drop_target_g
    reset_coil: reset_drop_targets
  e:
    switch: drop_target_e
    reset_coil: reset_drop_targets
```

Important: Not all "drop targets" in your machine will be configured as "drop targets." Some machines have drop target mechanisms that actually act as diverters. For example, in *Attack From Mars*, the drop target under the saucer is actually a diverter. When it's up, the ball stays on the playfield. When it's down, the ball enters the lock. *Star Trek: The Next Generation* has this with the drop target up above the lanes, and *The Wizard of Oz* has this for the drop target in front of the Winkie Guard. If a drop target in your machine is guarding a path to somewhere the ball can go, it might be a diverter. Of course sometime a drop target can be both, like the "D" target in *Judge Dredd*. Feel free to post to the forum with questions.

Notice there are no settings to control lights associated with drop targets, but many machines (like *Judge Dredd* used in the example) have lights for each drop target. To control those lights, you'd create shots based on the lights and switches for each drop target, and then you control them just like any other shot with the *shot* settings, *shot_group* settings, and *shot profiles*. In this case you'd end up specifying your switch for this drop target as well as for a shot for it. It's okay to have the same switch in both places.

Create one entry in your `drop_targets:` section for each drop target in your machine. Don't worry about grouping drop targets into banks here. (That's done in the `drop_target_banks:` section.) The drop target name can be whatever you want, and it will be the name for this drop target which is used throughout your machine.

Required settings

The following sections are required in the `drop_targets:` section of your config:

switch:

Single value, type: string name of a *switches* device. Defaults to empty.

The name of the switch that's activated when this drop target is down. (Note that active switch = target down, so if your drop target uses opto switches which are reversed, then you need to configure this switch with *type: NC* in the *switches:* section of your config file.) MPF will automatically update the state of the drop target whenever the switch changes state.

Optional settings

The following sections are optional in the `drop_targets:` section of your config. (If you don't include them, the default will be used).

ball_search_order:

Single value, type: `integer`. Default: `100`

A relative value which controls the order individual devices are pulsed when ball search is running. Lower numbers are checked first. Set to `0` if you do not want this device to be included in the ball search. See the *Ball Search* documentation for details.

disable_keep_up_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, will send a "disable" command to the drop target's reset coil, disabling the "keep up".

enable_keep_up_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, will send enable the drop target's reset coil which means that balls that hit it do not cause the drop target to fall since the reset coil is being held on. Note that this will require either `allow_enable: true` in the coil's configuration or a `default_hold_power:/max_hold_power` setting. See the (*Adjust coil hold power*) documentation for details.

Also note that many drop target coils are not designed to be held on at full power, so you'll most likely want to use a hold power of less than 8. Start low and only use the minimum power you need to keep the drop target up.

ignore_switch_ms:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `500ms`

How long this device should ignore switch changes while ball search is running. (Otherwise the ball search pulsing coils will set switches that could add to the score, start modes, etc. Default is `500ms`.

knockdown_coil:

Single value, type: string name of a *coils* device. Defaults to empty.

This is an optional coil that's used to knock down a drop target. Most drop targets do not have these. (In the *Judge Dredd* example above, you'll notice that only the *D* target has a knockdown coil.

knockdown_coil_max_wait_ms:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `100ms`

Max time allowed to delay the pulse of the knockdown coil. This is used to prevent excess power usage. See *psus:* for details.

knockdown_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, pulse this drop target's knockdown coil. (If this drop target doesn't have a knockdown coil, then these events will have no effect.)

max_reset_attempts:

Single value, type: `integer`. Defaults to empty.

Todo: *Help us to write it*

---

playfield:

Single value, type: string name of a *playfields* device. Default: `playfield`

The name of the playfield that this autofire device is on. The default setting is "playfield", so you only have to change this value if you have more than one playfield and you're managing them separately.

reset_coil:

Single value, type: string name of a *coils* device. Defaults to empty.

The name of the coil that is pulsed to reset this drop target. The pulse time will be whatever you configure as the default pulse time for this coil in the *coils:* section of your machine configuration file. Important: Only enter a *reset_coil* name here if this coil is only resets this drop target. For banks of drop targets where a single coil resets the entire bank of targets, enter the *reset_coil* in the *drop_target_banks:* configuration, not here. Why? Because if you have three drop targets in a bank, you only want to pulse the coil once to reset all the drop targets. If you enter the coil three times (one for each drop target), then it will pulse three times when the bank is reset.

reset_coil_max_wait_ms:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `100ms`

Max time allowed to delay the pulse of the reset coil. This is used to prevent excess power usage. See *psus:* for details.

reset_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `ball_starting, machine_reset_phase_3`

Default: `ball_starting, machine_reset_phase_3`

Resets this drop target. If this drop target is not part of a drop target bank, then resetting this target will pulse its reset coil. If this drop target is part of a drop target bank, then resetting this drop target will have no effect. (Instead you would reset the bank.) Default is *ball_starting, machine_reset_phase_3*.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

See the *documentation on the debug setting* for details.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

A descriptive name for this device which will show up in the service menu and reports.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Special / reserved tags for drop targets: *None*

See the *documentation on tags* for details.

Related How To guides

- *Drop Targets*

dual_wound_coils:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `dual_wound_coils:` section of your config is where you configure dual-would coils that are added to your "coils" device list which can be used anywhere in MPF.

Here's an example:

```
coils:
  c_hold:
    number:
    allow_enable: true
  c_power:
    number:
    default_pulse_ms: 20
switches:
  s_eos:
    number:
dual_wound_coils:
  c_dual_wound:
    hold_coil: c_hold
    main_coil: c_power
    eos_switch: s_eos
```

In the configuration above, a new coil called `c_dual_wound` is created that, when enabled, would energize both the `c_hold` and `c_power` coils. Then when the `s_eos` switch is activated, the `c_power` coil would be de-energized, leaving just the `c_hold` coil active until the `c_dual_wound` coil is deactivated.

Note: Note: Dual-wound flipper coils are configured in the `flippers:` section of the config, so you don't have to define them here. Other dual-wound coils (like for diverters, etc.) should be defined here since other MPF devices do not have explicit support for dual-wound coils.

Required settings

The following sections are required in the `dual_wound_coils:` section of your config:

hold_coil:

Single value, type: string name of a *coils* device. Defaults to empty.

The name of the hold coil winding. This coil must be a valid coil defined in your `coils:` section.

main_coil:

Single value, type: string name of a *coils* device. Defaults to empty.

The name of the main (power) coil winding. This coil must be a valid coil defined in your `coils:` section.

When this dual-wound coils is enabled, this coil will be pulsed for the number of milliseconds specified in the original coil's `default_pulse_ms:` setting.

Optional settings

The following sections are optional in the `dual_wound_coils:` section of your config. (If you don't include them, the default will be used).

eos_switch:

Single value, type: string name of a *switches* device. Defaults to empty.

The name of a switch which, when activated, will disable the power to the main coil winding.

Todo: Verify whether this has been implemented?

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

See the *documentation on the debug setting* for details.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

A descriptive name for this device which will show up in the service menu and reports.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Special / reserved tags for dual-wound coils: *None*

See the *documentation on tags* for details.

Related How To guides

- *Dual-wound Coils*

event_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

Note: This section can also be used in a show file in the `events:` section of a step.

You can use the `event_player:` section of your config files to cause additional events to be automatically posted when a specific event is posted. The event_player can be thought of as a really simple way to implement game logic. (e.g. "When this happens, do this.")

If you add this section to your machine-wide config file, the entries here will always be active. If you enter it into a mode-specific config file, entries will only be active while that mode is active.

This is an example:

```
event_player:
  ball_starting:
    - show_ball_start_animation
    - play_start_sound
    - start_first_mode
  ball_ending:
    - show_ball_ending_animation
    - play_drain_sound
```

See *Event player* for details.

Related How To guides

- *Event player*

extra_balls:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**YES**

The `extra_balls:` section of your config is where you configure which events trigger and reset extra ball awards.

Note that this extra ball abstract device only takes care of awarding the extra ball and can lit a light when enabled/available. The logic to qualify for an extra ball has to be implemented in your mode. See *Extra Balls* for more details.

Here's an example:

```
##! mode: mode1
extra_balls:
  my_mode_eb:
    award_events: alien_smashed
```

In the above example, the extra ball called `my_mode_eb` will be given to the player when the event `alien_smashed` is posted. After that, future `alien_smashed` events will not lead to additional extra balls. (The `my_mode_eb` extra ball is "used up", in a sense).

This is all tracked per-player in a player variable dictionary called "extra_balls_awarded"

Optional settings

The following sections are optional in the `extra_balls:` section of your config. (If you don't include them, the default will be used).

award_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, award this extra ball to the current player.

enabled:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Whether the device starts enabled or disabled.

group:

Single value, type: string name of a *extra_ball_groups* device. Defaults to empty.

The extra ball group which this ball belongs to which can further limit the maximum number of balls and enable/disable the device.

light_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Event to light the extra ball (if enabled).

max_per_game:

Single value, type: `integer`. Default: `1`

Maximum number of extra balls to award per player for this particular extra ball device. This might be further limited by the extra_ball_group max_per_game limit. In that case if either of the two limits is exceeded no more balls will be awarded.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

See the *documentation on the debug setting* for details.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

A descriptive name for this device which will show up in the service menu and reports.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Special / reserved tags for extra balls: *None*

See the *documentation on tags* for details.

Related How To guides

- *Extra Balls*

extra_ball_groups:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `extra_ball_groups:` section of your config is where you. . .

Todo: *Help us to write it*

Optional settings

The following sections are optional in the `extra_ball_groups:` section of your config. (If you don't include them, the default will be used).

award_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Immediately awards an extra ball.

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

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

award_lit_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events to award a lit extra ball. If the player does not have any lit extra balls, this method does nothing.

enabled:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Whether this ball group is enabled.

light_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

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

lit_memory:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Todo: *Help us to write it*

max_lit:

Single value, type: `integer`. Defaults to empty.

Max concurrent lit extra balls.

max_per_ball:

Single value, type: `integer`. Defaults to empty.

Maximum number of extra balls per ball.

max_per_game:

Single value, type: `integer`. Defaults to empty.

Maximum number of extra balls per game.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Not used.

Related How To guides

- *Extra Balls*

fadecandy:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `fadecandy:` section of your config is where you configure your fadecandy hardware platform. Usually you can leave this at the defaults. See *How to configure a FadeCandy RGB LED Controller* for more details.

From the fadecandy documentation:

Fadecandy internally represents colors with 16 bits of precision per channel, or 48 bits per pixel. Why 48-bit color? In combination with our dithering algorithm, this gives a lot more color resolution. It's especially helpful near the low end of the brightness range, where stair-stepping and color popping artifacts can be most apparent.

Each pixel goes through the following processing steps in Fadecandy:

- 8 bit per channel framebuffer values are expanded to 16 bits per channel
- We interpolate smoothly from the old framebuffer values to the new framebuffer values
- This interpolated 16-bit value goes through the color LUT, which itself is linearly interpolated
- The final 16-bit value is fed into our temporal dithering algorithm, which results in an 8-bit color
- These 8-bit colors are converted to the format needed by OctoWS2811's DMA engine
- In hardware, the converted colors are streamed out to eight LED strings in parallel

The color lookup tables can be used to implement gamma correction, brightness and contrast, and white point correction. Each channel (RGB) has a 257 entry table. Each entry is a 16-bit intensity. Entry 0 corresponds to the 16-bit color 0x0000, entry 1 corresponds to 0x0100, etc. The 257th entry corresponds to 0x10000, which is just past the end of the 16-bit intensity space.

Since MPF cannot do any better we suggest that you use this instead of our software color correction (which is limited to 8-bit resolution here).

Optional settings

The following sections are optional in the `fadecandy:` section of your config. (If you don't include them, the default will be used).

console_log:

Single value, type: one of the following options: none, basic, full. Default: `none`

Log level for the console log for this platform.

dithering:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Enabled temporal dithering for 16bit color precision. You want to leave this enabled since it looks much nicer (especially at low brightness).

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this platform.

gamma:

Single value, type: `number` (will be converted to floating point). Default: `2.5`

Specifies the gamma correction value for the lights. The default is 2.5.

keyframe_interpolation:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Whatever the fadecandy should fade between keyframes. You usually want to leave this at true since it looks much nicer.

linear_cutoff:

Single value, type: `number` (will be converted to floating point). Default: `0.0`

This is best explained by quoting the FadeCandy documentation: By default, brightness curves are entirely nonlinear. By setting linearCutoff to a nonzero value, though, a linear area may be defined at the bottom of the brightness curve. The linear section, near zero, avoids creating very low output values that will cause distracting flicker when dithered. This isn't a problem when the lights are viewed indirectly such that the flicker is below the threshold of perception, but in cases where the flicker is a problem this linear section can eliminate it entirely at the cost of some dynamic range. To enable the linear section, set linearCutoff to some nonzero value. A good starting point is 1/256.0, corresponding to the lowest 8-bit PWM level.

linear_slope:

Single value, type: `number` (will be converted to floating point). Default: `1.0`

Specifies the slope (output / input) of the linear section of the brightness curve for the lights. The default is 1.0.

whitepoint:

List of one (or more) values, each is a type: `number` (will be converted to floating point). Default: `1.0, 1.0, 1.0`

Specifies the white point (or white balance) of your lights. Enter it as a list of three floating point values that correspond to the red, blue, and green light segments. These values are treated as multipliers to all incoming color commands. The default of *1.0, 1.0, 1.0* means that no white point adjustment is used. *1.0, 1.0, 0.8* would set the blue segment to be at 80% brightness while red and green are 100%, etc.

You can use this to affect the overall brightness of lights (e.g. `0.8, 0.8, 0.8` would be 80% brightness as every color would be multiplied by 0.8). You can also use this to affect the "tint" (lowering the blue, for example).

Related How To guides

- *How to configure a FadeCandy RGB LED Controller*

fast:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `fast:` section of your machine-wide config is where you configure hardware options that are specific to the FAST Pinball Controller. Note that we have a how to guide which includes *all the FAST-specific settings* throughout your entire config file, so be sure to read that if you have FAST hardware.

```
fast:
  ports: com3, com4, com5
```

Required settings

The following sections are required in the `fast:` section of your config:

default_normal_debounce_close:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Defaults to empty.

Specifies the default value for the debounce time for switches that are configured with `debounce: normal` when they close.

Even though this is listed as a required setting, this entry is in the `mpfconfig.yaml` file, (with a value of `10ms`), so you don't have to enter it here unless you want to override that.

Also, keep in mind that this setting is only a default. You can override it for any switch in that switch's config.

default_normal_debounce_open:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Defaults to empty.

Specifies the default value for the debounce time for switches that are configured with `debounce: normal` when they open.

Even though this is listed as a required setting, this entry is in the `mpfconfig.yaml` file, (with a value of `10ms`), so you don't have to enter it here unless you want to override that.

Also, keep in mind that this setting is only a default. You can override it for any switch in that switch's config.

default_quick_debounce_close:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Defaults to empty.

Specifies the default value for the debounce time for switches that are configured with `debounce: quick` when they close.

Even though this is listed as a required setting, this entry is in the `mpfconfig.yaml` file, (with a value of `2ms`), so you don't have to enter it here unless you want to override that.

Also, keep in mind that this setting is only a default. You can override it for any switch in that switch's config.

default_quick_debounce_open:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Defaults to empty.

Specifies the default value for the debounce time for switches that are configured with `debounce: quick` when they open.

Even though this is listed as a required setting, this entry is in the `mpfconfig.yaml` file, (with a value of `2ms`), so you don't have to enter it here unless you want to override that.

Also, keep in mind that this setting is only a default. You can override it for any switch in that switch's config.

ports:

List of one (or more) values, each is a type: `string`. Defaults to empty.

A comma-separated list of the serial port names your FAST controller uses.

Optional settings

The following sections are optional in the `fast:` section of your config. (If you don't include them, the default will be used).

baud:

Single value, type: `integer`. Default: `921600`

The baud rate for the FAST COM ports.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `none`

Log level for the console log for this platform.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

See the *documentation on the debug setting* for details.

dmd_buffer:

Single value, type: `integer`. Default: `3`

Max backlog for the DMD port to prevent overflows in the FAST CPU.

driverboards:

Single value, type: one of the following options: fast, wpc, None. Defaults to empty.

Which driverboards are you using? Most likely `fast`. Similar to `driverboards` in the *hardware:* section. Use this setting if you use multiple playforms (i.e. FAST and P3-Roc) in one machine.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this platform.

firmware_updates:

List of one (or more) values, each is a type: *fast_firmware_update*. Defaults to empty.

A list of firmware versions which can be installed using *mpf hardware (command-line utility)*.

hardware_led_fade_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `0`

Controls how quickly LEDs will fade to their new color when they receive a color instruction from MPF.

The default is 0, which means if you set an LED to be red, it will turn red instantly. But if you set `hardware_led_fade_time: 20`, that means that when an LED receives an instruction to turn RED, it will smoothly fade from whatever color it is now to red over a period of 20ms.

You can play with different settings to pick something you like. Some people prefer the instant 0ms snappiness that's possible with LEDs. Others like to set this value to something like `100ms` which gives LEDs the more gentle fade style reminiscent of incandescent bulbs.

ignore_rgb_crash:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Ignore if the RGB CPU crashes. It will restart and the light will mostly recovery within a few seconds. If you set this to `False` MPF will shutdown when this happens because the hardware state is undefined when this happens.

net_buffer:

Single value, type: `integer`. Default: `10`

Max backlog for the NET port to prevent overflows in the FAST CPU.

rgb_buffer:

Single value, type: `integer`. Default: `3`

Max backlog for the RGB port to prevent overflows in the FAST CPU.

watchdog:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `1000`

The FAST controllers include a "watchdog" timer. A watchdog is a timer that is continuously counting down towards zero, and if it ever hits zero, the controller shuts off all the power to the drivers. The idea is that every time MPF runs a game loop (so, 30 times a second or whatever), MPF tells the FAST controller to reset the watchdog timer. So this timer is constantly getting reset and never hits zero.

But if MPF crashes or loses communication with the FAST controller, then this watchdog timer won't be reset. When it hits zero, the FAST controller will kill the power to the drivers. This should prevent an MPF crash from burning up driver or somehow damaging your hardware in another way.

You can set the watchdog timer to whatever you want. (This is essentially the max time a driver could be stuck "on" if MPF crashes.) The default is 1 second which is probably fine for almost everyone, and you don't have to include this section in your config if you want to use the default.

Related How To guides

- *How to configure MPF for FAST Pinball hardware*

fast_coils:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

The `fast_coils:` section of your config is where you configure platform specific settings for coils in the FAST platform.

Optional settings

The following sections are optional in the `fast_coils:` section of your config. (If you don't include them, the default will be used).

connection:

Single value, type: one of the following options: network, local, auto. Default: `auto`

How is your coil connected? For WPC this might be `local` otherwise `network`.

recycle_ms:

Single value, type: `time string (ms)` (*Instructions for entering time strings*).

The cooldown time of a coil after each pulse. Any pulse during that time will be ignored to prevent overheating the coil.

Related How To guides

- *How to configure MPF for FAST Pinball hardware*

fast_firmware_update:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

The `firmware_updates:` section of your `fast:` config is where you list all your firmware images. Those can then be installed using *mpf hardware firmware_update*.

Required settings

The following sections are required in the `fast_firmware_update:` section of your config:

file:

Single value, type: `string`.

The path of your firmware file.

type:

Single value, type: one of the following options: net, rgb.

For which CPU is this firmware file?

version:

Single value, type: `string`.

The exact version of the firmware. MPF will check that if this is higher than the installed version reported by the FAST CPU.

Related How To guides

- *How to configure MPF for FAST Pinball hardware*

fast_switches:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

The `fast_switches:` section of your config is where you configure platform specific details about switches when using *fast hardware*.

```
switches:
  some_switch:
    number:
    platform_settings:
      debounce_close: 2ms
      debounce_open: 4ms
```

Please make sure to read *Debouncing in Pinball Machines* before changing those times.

Optional settings

The following sections are optional in the `fast_switches:` section of your config. (If you don't include them, the default will be used).

debounce_close:

Single value, type: `string`.

Set the switch debounce time for closing the switch.

debounce_open:

Single value, type: `string`.

Set the switch debounce time for opening the switch.

Related How To guides

- *How to configure MPF for FAST Pinball hardware*

flasher_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

Note: This section can also be used in a show file in the `flashers:` section of a step.

The `flasher_player:` section of your config is where you can flash lights. See *Flasher player* for details.

Optional settings

The following sections are optional in the `flasher_player:` section of your config. (If you don't include them, the default will be used).

color:

Single value, type: `string`. Default: `on`

Set a color for flashing, if the flasher supports RGB coloring.

Color values may be a hex string (e.g. `22FFCC`), a list of RGB values (e.g. `[50, 128, 206]`), a color name (e.g. `turquoise`), or a brightness value (i.e. `AA` or `120`). MPF knows 140+ standard web color names, and you can define your own custom colors in the *named_colors:* section of your config. If you use brightness on an RGB light MPF will use the brightness for every channel. For instance brigness `AA` will result in color `AAAAAA`.

ms:

Single value, type: ms_or_token. Default: `100ms`

Configures how long should that flasher be enabled.

Related How To guides

- *Flashers*

flashers:

Removed in 0.50.

In most cases flashers can be configured as *coils*. You can use *coil_player* to pulse/flash them. Alternatively, you can configure them as *lights* and use *light_player* or *flasher_player* to control them.

Here is an example:

```
# configure the flasher as coil
coils:
  flasher_01:
    number: 4                    # this number depends on your hardware
    default_pulse_ms: 40         # pulse duration to use if no specified elsewhere
    max_hold_power: 1.0          # needed if you want to use flasher and light_player
# you can flash the flasher using flasher player
coil_player:
```

(continues on next page)

(continued from previous page)

```
  flash_coil:
    flasher_01:
      action: pulse              # will use the default 40ms pulse
# create a light which is backed by a coil (optional if you want to use light_player and flasher_player)
lights:
  flasher_01:
    number: flasher_01           # name of your coil
    platform: drivers            # use a coil
# use the light to flash the flasher
flasher_player:
  flash_flasher_01:
    flasher_01: 100ms
```

flippers:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `flippers:` section of your config contains all the settings for the flippers in a pinball machine.

Here's an example from a *Judge Dredd* machine with four flippers. (Note *Judge Dredd* technically has four flipper buttons too, but it's the style where you push the button part way in to flip the lower flipper, and all the way in to flip the upper flipper too. But as far as the game code is concerned, it sees two separate switches in each flipper button—-one that's activated via the half-press, and the second via the full press.)

Also note that flippers are kind of complex and there are a lot of options. Read the *Flippers* tech note for details. (You should definitely read that first before digging into the configuration options here.)

Note: The `flippers:` section of the config is only used for controlled flippers in newer machines. Early solid-state (pre-WPC) machines used enable relays to enable the flippers, and those are configured elsewhere. (See the How To guides for details.)

```
flippers:
  lower_left:
    main_coil: c_flipper_lower_left_main
    hold_coil: c_flipper_lower_left_hold
    activation_switch: s_flipper_left
    eos_switch: flipperLwL_EOS
    label: Left Main Flipper
  lower_right:
    main_coil: c_flipper_lower_right_main
    hold_coil: c_flipper_lower_right_hold
    activation_switch: s_flipper_right
    eos_switch: flipperLwR_EOS
    label: Right Main Flipper
  upper_left:
    main_coil: flipperUpLMain
```

(continues on next page)

(continued from previous page)

```
    hold_coil: flipperUpLHold
    activation_switch: flipperUpL
    eos_switch: flipperUpL_EOS
    label: Upper Left Flipper
  upper_right:
    main_coil: flipperUpRMain
    hold_coil: flipperUpRHold
    activation_switch: flipperUpR
    eos_switch: flipperUpR_EOS
    label: Upper Right Flipper
```

Required settings

The following sections are required in the `flippers:` section of your config:

main_coil:

Single value, type: string name of a *coils* device. Defaults to empty.

The name of the main flipper coil. For flippers that only have single- wound coils, this is where you specify that coil. In that case you would also configure the lower-power hold option for this coil in the *coils:* section of your config.

Optional settings

The following sections are optional in the `flippers:` section of your config. (If you don't include them, the default will be used).

activation_switch:

Single value, type: string name of a *switches* device. Defaults to empty.

The switch that controls this flipper (i.e. the flipper button). This setting is optional because you can also use `sw_flip_enable` below but `activation_switch` is far more common and recommended instead.

ball_search_hold_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `1s`

How long this flipper will be activated for when it is activated during ball search.

ball_search_order:

Single value, type: `integer`. Default: `100`

A relative value which controls the order individual devices are pulsed when ball search is running. Lower numbers are checked first. See the *Ball Search* documentation for details.

disable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `ball_will_end, service_mode_entered`

(Note that if you add an entry here, it will replace the default. So if you also want the default value(s) to apply, add them too.)

Disables this flipper (meaning pushing the flipper button doesn't active the flipper).

enable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `ball_started`

(Note that if you add an entry here, it will replace the default. So if you also want the default value(s) to apply, add them too.)

Enables this flipper.

eos_active_ms_before_repulse:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `500`

If you specify `repulse_on_eos_open` MPF will wait this many milliseconds until issuing an EOS repulse. The rational for this is that we do not want to stress the main coil too much. For instance if the hold coil break we do not want to continuously pulse the coil.

eos_switch:

Single value, type: string name of a *switches* device. Defaults to empty.

EOS switch on this flipper (if there is one).

eos_switch_overwrite:

One or more sub-entries. Each in the format of `string` : `string`

One or more sub-entries, each in the format of `string` : `string` If you're using an end of stroke switch with this flipper, enter the switch name here.

hold_coil:

Single value, type: string name of a *coils* device. Defaults to empty.

The name of the hold coil winding for dual-wound flipper coils.

hold_coil_overwrite:

Single value, type: *coil_overwrites*. Defaults to empty.

Overwrites settings on the hold_coil. See *coil_overwrites:* for details.

include_in_ball_search:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Controls whether this flipper is included in ball search.

Usually flippers aren't included in ball search. However if you have upper flippers, it's probably good to include them in the ball search since it's often possible for an upper flipper to disable and hold a ball under the flipper. Usually this isn't an issue since the player can just flip to release the ball. However if the machine has tilted (or the flippers are otherwise disabled), then it's possible for a flipper to come down on the ball and get it stuck. So you definitely want to include upper flippers in ball search.

BTW, this is something that happened to us in *Wizard of Oz*, so that's how we thought to include an option for flippers in ball search. :)

main_coil_overwrite:

Single value, type: *coil_overwrites*. Defaults to empty.

Overwrites settings on the main_coil. See *coil_overwrites:* for details.

playfield:

Single value, type: string name of a *playfields* device. Default: `playfield`

change this value if you have more than one playfield and you're managing them separately.

power_setting_name:

Single value, type: `string`. Defaults to empty.

A *machine setting* to use to adjust the (relative) power. It can be used to allow the operator to adjust the power in service mode.

This is an example:

```
coils:
  c_flipper_main:
    number:
switches:
  s_flipper:
    number: 1
    tags: left_flipper
flippers:
  f_test_flippers_with_settings:
    main_coil: c_flipper_main
```

(continues on next page)

(continued from previous page)

```
power_setting_name: flipper_power
activation_switch: s_flipper
```

MPF comes with a *setting* called `flipper_power` by default and you can add additional ones.

repulse_on_eos_open:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Whether MPF should repulse the main coil of the flipper when the EOS reopens and the flipper buttons are still active. Not all platforms support this in hardware. MPF might emulate this in software for platforms which do not support this. Consult your platform manual if in doubt.

sw_flip_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

If the flipper is enabled this will flip the flipper from software. This will usually have some delay and jitter so use with care. In almost all cases it is prefered to use an `activation_switch` which will use hardware rules internally to flip the flipper.

sw_release_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Disables a flipper from software. Use this together with `sw_flip_events`.

switch_overwrite:

One or more sub-entries. Each in the format of `string` : `string`

One or more sub-entries, each in the format of `string` : `string` Overwrites settings on the activation_switch. See *switch_overwrites:* for details.

use_eos:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Controls whether an EOS switch is used to disable the main winding or to switch to lower-power pwm mode.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

See the *documentation on the debug setting* for details.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

A descriptive name for this device which will show up in the service menu and reports.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Special / reserved tags for flippers: *None*

See the *documentation on tags* for details.

Related How To guides

- *Flippers*

game:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `game:` section of the machine config holds settings related to the game play.

```
game:
  balls_per_game: 3
  max_players: 4
```

Optional settings

The following sections are optional in the `game:` section of your config. (If you don't include them, the default will be used).

add_player_event:

Single event. The device will add an handler for this event. Defaults to empty.

An event name which will request to add a player. Same as `add_player_switch_tag` but using an event instead of a switch tag (see below).

add_player_switch_tag:

Single value, type: `string`. Default: `start`

The tag of the switch that's used to request to add a player to an existing game. (We say "request to add a player" instead of "add a player" because it's possible that adding a player is not allowed. For example, if the machine is set to require credits and there are not enough credits available, or the game already has the maximum number of players.)

This is the name of the tag in the `tags:` section of one of your switches.

allow_start_with_ball_in_drain:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Controls whether it's possible to start a game when a ball is in a ball device that's tagged with `drain` but not `home` or `trough`. (This is needed in some older machines that have non-standard trough/drain device configurations.

allow_start_with_loose_balls:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Controls whether it's possible to start a game when balls are not all in ball devices tagged with `home`.

balls_per_game:

Single value, type: `integer` or `template` (*Instructions for entering templates*). Default: `3`

How many balls the game is. Typically it's 3 or 5 but it can be anything. MPF doesn't care.

Also note that you can use *dynamic values* here if you want to do math or use settings to make this configurable.

end_ball_event:

Single event. The device will add an handler for this event. Default: `end_ball`

When this event is handled by the game it will end the current ball. This is similar to the last ball draining. Use with care if there are still balls in play.

end_game_event:

Single event. The device will add an handler for this event. Default: `end_game`

When this event is handled by the game it will end the game. This is similar to slam tilt but bonus mode, match mode etc will still run.

max_players:

Single value, type: `integer` or `template` (*Instructions for entering templates*). Default: `4`

Controls the maximum number of players that can play a game.

Also note that you can use *dynamic values* here if you want to do math or use settings to make this configurable.

start_game_event:

Single event. The device will add an handler for this event. Defaults to empty.

Event to request to start a game. Same as `start_game_switch_tag` but using an event instead of a switch tag (see below for details).

start_game_switch_tag:

Single value, type: `string`. Default: `start`

The tag of the switch that's used to request to start a game. (We say "request to start a game" instead of "start a game" because it's possible that starting a game is not allowed. For example, if the machine is set to require credits and there are not enough credits available.)

This is the name of the tag in the `tags:` section of one of your switches.

wait_for_empty_playfields_on_ball_start:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Todo: *Help us to write it*

Related How To guides

- *Game Logic*

gi_player:

Removed in 0.50. Use *light_player* instead.

gis:

> **Warning:** As of MPF 0.50, `matrix_lights`, `flashers` and `leds` have been combined into a single `lights` configuration. See *lights:* for details.

You would configure GIs as normal lights with `subtype: gi` (see your platform documentation for details about `subtype`).

Here's an example from *Judge Dredd*:

```
lights:
  gi01:    # lower backglass
    number: G01
    subtype: gi
  gi02:    # mid backglass and rear playfield
    number: G02
    subtype: gi
  gi03:    # upper left backglass and slings, variable
    number: G03
    subtype: gi
  gi04:    # upper right backglass and Deadworld globe, variable
    number: G04
    subtype: gi
  gi05:    # coin slot lights & side cabinet fire buttons
    number: G05
    subtype: gi
```

See *lights:* for details about the `lights` section.

hardware:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `hardware:` section of your machine config file is where you configure the options for the *physical hardware controller boards* that MPF will use.

If you intend to use MPF with physical hardware, at a minimum you'll have a `platform:` and `driverboards:` section in your machine config, like this:

```
hardware:
  platform: fast
  driverboards: fast
```

Device-specific defaults

The following optional settings can be used to set default platforms for a specific class of devices. Note that `virtual` and `smart_virtual` are valid options for all of these, though they are not included in the lists below. Also note that those lists are not exhaustive.

Note: The list of platforms is incomplete here. See the *MPF compatible control systems / hardware* for details which platforms are supported by MPF.

Optional settings

The following sections are optional in the `hardware:` section of your config. (If you don't include them, the default will be used).

accelerometers:

List of one (or more) values, each is a type: `string`. Default: `default`

See *DMD Platforms in MPF* for supported platforms.

coils:

List of one (or more) values, each is a type: `string`. Default: `default`

For instance:

- `p_roc`
- `p3_roc`
- `fast`
- `opp`
- `apc`
- `snux`

Almost all platforms in *MPF compatible control systems / hardware* are supported here.

dmd:

List of one (or more) values, each is a type: `string`. Default: `default`

See *DMD Platforms in MPF* for supported platforms.

driverboards:

Single value, type: `string`. Defaults to empty.

Specifies the default type of driver boards you're using. If you have a home brew machine, this will probably match your platform. If you're using an existing machine, then this will be whatever type of driverboard is installed in the machine.

- pdb P-ROC Driver Boards, PD-16, PD-8x8, etc.)
- `fast` FAST IO boards (0804, 1616, 3208, etc.)
- opp OPP wing boards

- `wpc95` Williams WPC-95
- `wpc` Williams WPC
- `wpcAlphaNumeric` Williams WPC with alphanumeric 14-pin connected segmented display
- `sternSAM` Stern SAM
- `sternWhitestar` Stern Whitestar

hardware_sound_system:

List of one (or more) values, each is a type: `string`. Default: `default`

See *MPF compatible control systems / hardware* for supported platforms.

i2c:

List of one (or more) values, each is a type: `string`. Default: `default`

See *I2C Platforms in MPF* for supported platforms.

lights:

List of one (or more) values, each is a type: `string`. Default: `default`

Almost all platforms in *MPF compatible control systems / hardware* are supported here.

platform:

List of one (or more) values, each is a type: `string`. Default: `virtual`

Specifies the default platform that will be used by all devices in the config. We say this is the "default" platform, because it's possible to use more than one platform at time. (Maybe you use a P-ROC for coils and switches and a FadeCandy for RGB LEDs, etc.) See the *Mixing-and-Matching hardware platforms* for more details on this.

See *MPF compatible control systems / hardware* for a complete list.

rgb_dmd:

List of one (or more) values, each is a type: `string`. Default: `default`

See *DMD Platforms in MPF* for supported platforms.

segment_displays:

List of one (or more) values, each is a type: `string`. Default: `default`

See *Segment Display Platforms in MPF* for supported platforms.

servo_controllers:

List of one (or more) values, each is a type: `string`. Default: `default`

See *Servo Platforms in MPF* for supported platforms.

stepper_controllers:

List of one (or more) values, each is a type: `string`. Default: `default`

See *Stepper Platforms in MPF* for supported platforms.

switches:

List of one (or more) values, each is a type: `string`. Default: `default`

Almost all platforms in *MPF compatible control systems / hardware* are supported here.

Related How To guides

- *MPF compatible control systems / hardware*

hardware_sound_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

Note: This section can also be used in a show file in the `hardware_sounds:` section of a step.

The `hardware_sound_player:` section of your config is where you can control external sound modules (e.g. in LISY).

This is an example:

```
hardware_sound_systems:
  default:
    label: Default external sound system
hardware_sound_player:
  event_posted_elsewhere1:
    2:
      action: play
  ball_started:
    3: play
  test_stop: stop
```

Optional settings

The following sections are optional in the `hardware_sound_player:` section of your config. (If you don't include them, the default will be used).

action:

Single value, type: one of the following options: play, play_file, text_to_speech, set_volume, increase_volume, decrease_volume, stop. Default: `play`

`play` will play a sound. Depending on the hardware this might stop previous sounds. Also loop behaviour depends on the hardware and might be different per sound.

`stop` will stop all sounds.

platform_options:

Single value, type: dict. Defaults to empty.

Todo: *Help us to write it*

---

sound_system:

Single value, type: string name of a *hardware_sound_systems* device. Default: `default`

In case you got multiple hardware_sound platforms you can expliticly select one here.

track:

Single value, type: `integer`. Default: `1`

The track number to play this sound on. What this means depends on your hardware. Usually, there are one or two tracks.

value:

Single value, type: `string`. Defaults to empty.

The number of your sound.

Related How To guides

- *Arduino Pinball Controller*
- *How to use MPF with the LISY platform*

hardware_sound_systems:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `hardware_sound_systems:` section of your config is where you configure external sound systems. For instance, this is used in the *LISY platform*.

Optional settings

The following sections are optional in the `hardware_sound_systems:` section of your config. (If you don't include them, the default will be used).

platform:

Single value, type: `string`. Defaults to empty.

Overwrite the default platform.

platform_settings:

Single value, type: dict. Defaults to empty.

Todo: *Help us to write it*

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Not used.

Related How To guides

- *Arduino Pinball Controller*
- *How to use MPF with the LISY platform*

high_score:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**YES**

The `high_score:` section of your config is where you configure the built-in high score mode. See *High Scores* for details.

Required settings

The following sections are required in the `high_score:` section of your config:

categories:

Ordered list for one (or more) sub-settings. Each in the format of `string` : `list` (*Instructions* for entering lists)

An ordered map of categories which contain a list of awards. See *High Scores* for an example.

Optional settings

The following sections are optional in the `high_score:` section of your config. (If you don't include them, the default will be used).

award_slide_display_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `4s`

How long should the award slide be displayed?

defaults:

One or more sub-entries. Each in the format of `string` : `list` (*Instructions* for entering lists)

A map of categories with a list of player/score tuples. See *High Scores* for an example.

enter_initials_timeout:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `20s`

Timeout for the player to enter his/her initials.

reset_high_scores_events:

List of one (or more) events. The device will add handlers for those events. Default: `high_scores_reset, factory_reset`

Event to reset high scores. The default is used by the service mode.

reverse_sort:

List of one (or more) values, each is a type: `string`. Defaults to empty.

A list of categories where the sort should be inverted. Usually the highest score is the best but sometimes you want the shortest time or least amount of shots to be the best score.

Related How To guides

- *High Scores*

image_pools:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `image_pools:` section of your config is where you. . .

Todo: *Help us to write it*

images:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `images:` section of your config is where you configure non-default parameter values for any image assets you want to use in your game. Note: You do *not* have to have an entry for every single image you want to use, rather, you only need to add individual assets to your config file that have settings which are different from other assets in that asset's folder. (This section is part of the MPF media controller and only available if you're using MPF-MC for your media controller.)

More information on working with assets is in the *Assets* section of the documentation.

Each sub-entry in your `image:` section is the name that MPF will use to refer to that asset. (In other words it's how you specify that asset in other areas of your config files.) The asset manager works by first scanning the file system to build up a list of asset files it finds. Then it looks at the config to see if there are any additional settings specified for each asset.

For example:

```
images:
  insert_coin:
    load: preload
  hello_face:
    file: hello_face_300.jpg
    load: None
```

So in the example above, if the asset manager found a file called `insert_coin.jpg` on disk, then it will also see the `insert_coin` entry in the config file and know that those two match. (The "match" is just based on the part of the file name without the extension, so the settings entry for `insert_coin:` would match `insert_coin.jpg` and `insert_coin.png`. In other words, don't name two files with the same name if you want to keep them straight.)

Optional settings

The following sections are optional in the `images:` section of your config. (If you don't include them, the default will be used).

file:

Single value, type: `string`. Defaults to empty.

Sometimes you might want to name a file one thing on disk but refer to it as another thing in your game and config files. In this case, you can create an `file:` setting in an asset entry. (Note the file: `hello_face_300.jpg` setting in the example above, and note that it includes the file extension.) In this example, you would refer to that image asset as `hello_face` even though the file is `hello_face_300`.

You might be wondering why this exists? Why not just change the file name to be whatever you want and/or who cares what the name is? The reason this function exists is because it allows for the separation of the actual file on disk from the way it's called in the game. For example, you could use this to create two sets of assets—one for a traditional DMD and one for a color DMD—and then you could refer to the asset by its generic name throughout your configs. (In other words, you could swap out assets for different physical machine types without having to update your display code.) That said, we expect that 99% of people won't use this `file:` setting, which is fine.

frame_skips:

List of one (or more) values, each is a type: images_frame_skips. Defaults to empty.

Todo: *Help us to write it*

image_template:

Single value, type: *images*. Defaults to empty.

Todo: *Help us to write it*

load:

Single value, type: `string`. Defaults to empty.

Specifies when this asset should be loaded. (See the *Assets* documentation for an explanation on loading.)

- *preload* (The asset is loaded when MPF boots and stays in memory as long as MPF is running.)
- *mode_start* (The asset is loaded when the mode starts and is unloaded when the mode ends. This option is only valid for asset files that are in mode folders, not machine-wide assets.)
- Anything else (or nothing at all) means that the asset it loaded "on demand" when it's first called for. (At this point, assets loaded on demand stay in memory forever, but at some point we'll change that so they get unloaded on demand too.)

Note that you can configure `load:` options in the *assets:* section of your config files. It's nice to be able to override those on an asset-by-asset basis. For example, you might configure your assets for a mode to all load when the mode starts, but you could also create a few entries in your config files with `load: preload` for the assets that are needed for the intro show of the mode. That way that show can play while the other assets are loading in the background. (Of course you could also create a subfolder for the assets that you want to preload and specific an `assets:` entry for that folder rather than specifying entries in your config for specific assets. The choice is up to you.)

Related How To guides

- *Image Widget*

info_lights:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `info_lights:` section of a machine config file allows you to configure the "Info Lights" plugin to automatically set "status" lights based on different things that are happening in the game. This is very common in EM and older solid state machines, since they use lights to tell the player whose turn it is, what ball they're on, etc.

Here's an example `info_lights:` section from a machine configuration file:

```
info_lights:
  match_00:
    light: match00
  match_10:
    light: match10
  match_20:
    light: match20
  match_30:
    light: match30
  match_40:
    light: match40
  match_50:
    light: match50
  match_60:
    light: match60
  match_70:
    light: match70
  match_80:
    light: match80
  match_90:
    light: match90
  ball_1:
    light: bip1
  ball_2:
    light: bip2
  ball_3:
    light: bip3
  ball_4:
    light: bip4
  ball_5:
    light: bip5
  player_1:
    light: player1
  player_2:
    light: player2
  tilt:
    light: tilt
  game_over:
    light: gameOver
```

The way info lights work is pretty simple. There are sub-sections that represent different lights that may be in your machine, and then under each of them you map them to the name of the light.

Then they pretty much just work automatically.

Note that the the `light:` entry in each of these refers to a device in the *lights:* section.

match_XX:

This section is for the match lights, with the "XX" replaced with the number of the match light. In the example configuration above, the machine has match lights that count up by tens (10, 20, 30. . .) which is why the match_xx entries here are `match_10`, `match_20`, `match_30`, etc. If your machine matches by the ones digit, then you'd enter these items as `match_1`, `match_2`, etc.

ball_XX:

This maps the ball-in-play number to the light.

player_XX:

This maps the current player to the number in the light. This plugin turns on each light when a new player joins a game. So it doesn't show which player is up, rather, if you have a two-player game then both the `player_1` and `player_2` lights are lit. (So how does a player know that it's his turn? That's handled by the score reel lights.)

tilt:

Turns this light on when the machine tilts.

game_over:

Flashes this light when a game is not in progress at a rate of 1/2 sec on, 1/2 sec off.

keyboard:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `keyboard:` section of your config is used to configure options for how you map computer keyboard keys to pinball machine switches and events. This is useful for testing your game from your computer when you're not around your physical machine.

You might also want to implement some virtual switches in your machine which can be only used via a keyboard for debugging.

Options for each key & key combination

Once you enter the key and/or key combination, then you need to create a subsection which defines what this key or key combination does when it's hit. There are several options:

Optional settings

The following sections are optional in the `keyboard:` section of your config. (If you don't include them, the default will be used).

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Todo: *Help us to write it*

event:

Single event. This device will be posted by the device. Defaults to empty.

You can specify an event name to be posted when this key is pressed. This is useful for testing when you want to test some part of your game code based on an event. For example, you could map a keyboard key to *clockwise_orbit_hit* event instead of having to hit the *left_orbit_enter* key quickly followed by the *right_orbit_enter* key. Events entered here are transmitted posted by the MPF core engine process.

invert:

Single value, type: `boolean` (`true`/`false`). Default: `false`

If True, then this key is inverted, meaning the associated switch is active when you're not pushing the key down, and it's inactive when you're holding the key.

mc_event:

Single event. This device will be posted by the device. Defaults to empty.

This is similar to the *event:* entry, except an *mc_event* is posted as events in the media controller process, rather than in the MPF process.

params:

One or more sub-entries. Each in the format of `string` : `string`

This section contains subsections which are a list of parameters that are posted along with the *event* or *mc_event* specified above. Using the following configuration file snippet as an example:

```
keyboard:
  4:
    event: advance_reel_test
    params:
      reel_name: score_1p_10
      direction: 1
```

This keyboard entry will post the event *advance_reel_test* when the *4* key is pressed, and it will pass the parameters *reel_name=score_1p_10* and *direction=1*.

switch:

Single value, type: string name of a *switches* device. Defaults to empty.

The switch name of the pinball machine switch you want this key (or key combination) to control.

toggle:

Single value, type: `boolean` (`true`/`false`). Default: `false`

If True, then the key acts like a "push on / push off" key, where you just have to tap it once to hold the switch active. This is useful for switches in ball devices, since you don't want to have to hold down the keys on your keyboard forever whenever a ball is locked in a device. Default is *False*. You might want to create multiple entries for the same switch for different key combinations. For example:

```
  1:
    switch: trough1
  shift+1:
    switch: trough1
    toggle: true
```

In the above code, you can momentarily "tap" the *trough1* switch by hitting the *1* key, but if you want to lock that switch on, then you can push *Shift+1*.

Related How To guides

- *Connecting Your Computer Keyboard to MPF Switches*

kickbacks:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `kickbacks:` section of your machine config is used to define *kickback mechanisms* which are a type of *autofire coil* that kicks the ball back into play, typically located in an outlane.

This is an example:

```
switches:
  s_kickback:
    number: 1
coils:
  c_kickback:
    number: 1
    default_pulse_ms: 20ms
kickbacks:
  left_kickback:
    coil: c_kickback
    switch: s_kickback
```

Since kickbacks are a type of autofire coil, they have the same settings as *autofire_coils:*. See that documentation for a list of all the settings and options.

Required settings

The following sections are required in the `kickbacks:` section of your config:

coil:

Single value, type: string name of a *coils* device. Defaults to empty.

The name of the coil you want to fire. (Actually, perhaps we should phrase it as the name of the coil you want to change the state on, because you can also use these kickback coil rules to cause coils to stop firing based on a switch change.)

switch:

Single value, type: string name of a *switches* device. Defaults to empty.

The name of the switch which will trigger the kickback coil. More precisely, this switch is used together with the `coil` in the hardware rules which will instruct your pinball hardware to pulse the coil.

Optional settings

The following sections are optional in the `kickbacks:` section of your config. (If you don't include them, the default will be used).

ball_search_order:

Single value, type: `integer`. Default: `100`

A relative value which controls the order individual devices are pulsed when ball search is running. Lower numbers are checked first. Set to `0` if you do not want this device to be included in the ball search. See the *Ball Search* documentation for details.

coil_overwrite:

Single value, type: *coil_overwrites*. Defaults to empty.

You can overwrite `recycle`, `pulse_ms`, `pulse_power` or `hold_power` of the coil for this device.

This is an example:

```
switches:
  s_kickback:
    number: 1
coils:
  c_kickback:
```

(continues on next page)

(continued from previous page)

```
    number: 1
    default_pulse_ms: 10ms
kickbacks:
  left_kickback:
    coil: c_kickback
    switch: s_kickback
    coil_overwrite:
      pulse_ms: 20ms
```

In this example we increase `pulse_ms` of the kickback.

coil_pulse_delay:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `0`

This setting will delay the pulse of your `coil` by a certain milliseconds after your `switch` has activated. Please note that this has to be supported in your hardware platform and not all platforms do that.

disable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `ball_will_end, service_mode_entered`

Disables this kickback coil by clearing the hardware rule from the pinball controller hardware.

enable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Enables this kickback coil by writing the hardware rule to the pinball controller hardware.

playfield:

Single value, type: string name of a *playfields* device. Default: `playfield`

The name of the playfield that this kickback device is on. The default setting is "playfield", so you only have to change this value if you have more than one playfield and you're managing them separately.

reverse_switch:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Boolean which controls whether this kickback device fires when the switch is active or inactive. The default behavior is that the coil is fired when the switch goes to an active state. If you want to reverse that, so the coil fires when the switch goes to inactive, then set this to False. (This is what you would use if you have an opto.) Default is *False*.

switch_overwrite:

One or more sub-entries. Each in the format of `string` : `string`

You can overwrite the debounce setting of your switch in this device.

timeout_disable_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `0`

To prevent machine gunning of your kickback coils you can define a windows `timeout_watch_time`. If more than `timeout_max_hits` hits to your switch (and thus responses by your coil) are seen by MPF it will disable the hardware rule for `timeout_disable_time` and reinstall it afterwards.

timeout_max_hits:

Single value, type: `integer`. Default: `0`

To prevent machine gunning of your kickback coils you can define a windows `timeout_watch_time`. If more than `timeout_max_hits` hits to your switch (and thus responses by your coil) are seen by MPF it will disable the hardware rule for `timeout_disable_time` and reinstall it afterwards.

timeout_watch_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `0`

To prevent machine gunning of your kickback coils you can define a windows `timeout_watch_time`. If more than `timeout_max_hits` hits to your switch (and thus responses by your coil) are seen by MPF it will disable the hardware rule for `timeout_disable_time` and reinstall it afterwards.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

See the *documentation on the debug setting* for details.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

The plain-English name for this device that will show up in operator menus and trouble reports.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Special / reserved tags for kickbacks: *None*

See the *documentation on tags* for details.

Related How To guides

- *Kickbacks*
- *autofire_coils:*

kivy_config:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `kivy_config:` section of your config is where you configure kivy.

You can directly configure kivy here. Usually you don't need this but in some cases it allows some additional tweaking (e.g. for embedded workloads). All options are documented in the kivy config documentation.

This is an example:

```
kivy_config:
  kivy:
    desktop: 1
    exit_on_escape: true
  graphics:
    borderless: false
    fbo: hardware  # hardware, software, force-hardware
    fullscreen: false
    multisamples: 2
    position: auto  # auto, custom
    show_cursor: true
    resizable: true
```

Related How To guides

- *Using multiple screens*

led_player:

led_player and *matrix_light_player* were replaced with *light_player* in MPF 0.50. See *lights:* for details.

light_stripes:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

A "led_stripe" will create "count" leds for you starting the number at "number_start". If you need a prefix or suffix for the number you can use "number_template". All settings in "led_template" will be applied to all LEDs. The only difference between *led_stripes* and *light_rings* is how the x/y coordinates are computed.

Here's an example:

```
#config_version=5

light_stripes:
  stripe1:
    number_start: 10
    light_template:
      tags: test
    count: 5
    debug: True
  stripe2:
    number_start: 200
    number_template: 7-{}
    count: 5
    direction: 90
    start_x: 10
    start_y: 20
    distance: 5
    debug: True
  stripe3:
    start_channel: ABC-123
    count: 5
    direction: 90
    start_x: 10
    start_y: 20
    distance: 5
    debug: True
    light_template:
      type: rgbw

light_rings:
  ring1:
    number_start: 20
    count: 12
    radius: 3
```

(continues on next page)

(continued from previous page)

```
    start_angle: 90
    center_x: 100
    center_y: 50
    debug: True

neoseg_displays:
  neoSeg_0:
    start_channel: 0-0-0
    size: 8digit
    light_template:
      type: w
      subtype: led
  neoSeg_1:
    start_channel: 0-0-120
    size: 2digit
    light_template:
      type: w
      subtype: led
```

Required settings

The following sections are required in the `light_stripes:` section of your config:

count:

Single value, type: `integer`. Defaults to empty.

The integer value for how many LEDs are in the stripe.

light_template:

Single value, type: *lights*. Defaults to empty.

This is a list of sub-settings (indented) that are regular settings from the *lights:* section of your machine config. Any settings that are valid there are valid here, and they're applied to all the LEDs in the stripe.

Optional settings

The following sections are optional in the `light_stripes:` section of your config. (If you don't include them, the default will be used).

direction:

Single value, type: `number` (will be converted to floating point). Defaults to empty.

The angle (in degrees, 0-360) the this LED stripe is positioned on the playfield. This is used for the calculation of x/y positions of individual LEDs only.

distance:

Single value, type: `number` (will be converted to floating point). Defaults to empty.

The distance between individual LEDs (in relative size to the x/y coordinates of the `start_x:` and `start_y:` positions. This is used for the calculation of x/y positions of individual LEDs only.

number_start:

Single value, type: `integer`. Default: `0`

The integer value for the number for the first LED in the stripe. (MPF assumes that all the LEDs in the stripe are numbered sequentially.)

number_template:

Single value, type: `string`. Defaults to empty.

MPF automatically configures the LEDs in a stripe. The first one uses the `number_start:` value, and then it counts up from there up through the `count:` value.

However, many hardware numbers for LEDs are not just vanilla numbers, rather they also include a board number or channel or something like that. The `number_template:` is where you specify what that number value looks like. Just use braces `{}` for the part you want replaced by a number.

The example config with a number template of `7-{}` with a number start of 200 and a count of 5 will create 5 LEDs with the numbers 7-200, 7-201, 7-202, 7-203, and 7-204.

previous:

Single value, type: string name of a *lights* device. Defaults to empty.

Todo: *Help us to write it*

start_channel:

Single value, type: `string`. Defaults to empty.

Todo: *Help us to write it*

start_x:

Single value, type: `number` (will be converted to floating point). Defaults to empty.

The "x" position of the first LED. (This is not used in MPF yet.)

start_y:

Single value, type: `number` (will be converted to floating point). Defaults to empty.

The "y" position of the first LED.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Unused.

Related How To guides

Todo: *Help us to write it*

light_rings:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

A "light_rings" will create "count" lights for you starting the number at "number_start". If you need a prefix or suffix for the number you can use "number_template". All settings in "light_template" will be applied to all lights. The only difference between *light_stripes* and *light_rings* is how the x/y coordinates are computed.

```
#config_version=5

light_stripes:
  stripe1:
    number_start: 10
    light_template:
      tags: test
    count: 5
    debug: True
  stripe2:
    number_start: 200
    number_template: 7-{}
    count: 5
    direction: 90
    start_x: 10
    start_y: 20
    distance: 5
    debug: True
  stripe3:
    start_channel: ABC-123
    count: 5
    direction: 90
    start_x: 10
    start_y: 20
    distance: 5
    debug: True
    light_template:
      type: rgbw

light_rings:
  ring1:
    number_start: 20
    count: 12
    radius: 3
    start_angle: 90
    center_x: 100
    center_y: 50
    debug: True

neoseg_displays:
  neoSeg_0:
    start_channel: 0-0-0
    size: 8digit
    light_template:
      type: w
      subtype: led
  neoSeg_1:
    start_channel: 0-0-120
    size: 2digit
    light_template:
```

(continues on next page)

(continued from previous page)

```
    type: w
    subtype: led
```

Required settings

The following sections are required in the `light_rings:` section of your config:

count:

Single value, type: `integer`. Defaults to empty.

The integer value for how many LEDs are in the ring.

light_template:

Single value, type: *lights*. Defaults to empty.

This is a list of sub-settings (indented) that are regular settings from the *lights:* section of your machine config. Any settings that are valid there are valid here, and they're applied to all the LEDs in the ring.

Optional settings

The following sections are optional in the `light_rings:` section of your config. (If you don't include them, the default will be used).

center_x:

Single value, type: `number` (will be converted to floating point). Defaults to empty.

The "x" position of the center of the ring. (This is not used in MPF yet.)

center_y:

Single value, type: `number` (will be converted to floating point). Defaults to empty.

The "y" position of the center of the ring.

number_start:

Single value, type: `integer`. Default: `0`

The integer value for the number for the first LED in the ring. (MPF assumes that all the LEDs in the ring are numbered sequentially.)

number_template:

Single value, type: `string`. Defaults to empty.

MPF automatically configures the LEDs in a ring. The first one uses the `number_start:` value, and then it counts up from there up through the `count:` value.

However, many hardware numbers for LEDs are not just vanilla numbers, rather they also include a board number or channel or something like that. The `number_template:` is where you specify what that number value looks like. Just use braces `{}` for the part you want replaced by a number.

The example config with a number template of `7-{}` with a number start of 200 and a count of 5 will create 5 LEDs with the numbers 7-200, 7-201, 7-202, 7-203, and 7-204.

previous:

Single value, type: string name of a *lights* device. Defaults to empty.

Todo: *Help us to write it*

radius:

Single value, type: `number` (will be converted to floating point). Defaults to empty.

The radius of the ring (in relative size to the x/y coordinates of the `center_x:` and `center_y:` positions. This is used for the calculation of x/y positions of individual LEDs only.

start_angle:

Single value, type: `number` (will be converted to floating point). Default: `0`

The angle (in degrees, 0-360) of the first LED in the right. This is used for the calculation of x/y positions of individual LEDs only.

start_channel:

Single value, type: `string`. Defaults to empty.

Todo: *Help us to write it*

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Not used.

Related How To guides

Todo: *Help us to write it*

lisy:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `lisy:` section of your config is where your lisy platform. See *How to use MPF with the LISY platform* for details.

Optional settings

The following sections are optional in the `lisy:` section of your config. (If you don't include them, the default will be used).

baud:

Single value, type: `integer`. Defaults to empty.

Baudrate when connecting to LISY using a serial port.

connection:

Single value, type: one of the following options: network, serial. Default: `network`

Whatever to use a network or serial connection.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `none`

Log level for the console log for this platform.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

See the *documentation on the debug setting* for details.

disable_dtr:

Single value, type: `boolean` (`true`/`false`). Default: `true`

If set to `True` MPF will try to prevent your operating system from toggling the DTR line of your serial. This is needed for *APC* and some other controllers which would reset when this happens. If in doubt check the documentation of your controller.

display_flash_duty:

Single value, type: `number` (will be converted to floating point). Default: `0.5`

Todo: *Help us to write it*

---

display_flash_frequency:

Single value, type: `number` (will be converted to floating point). Default: `1.0`

How fast should the displays flash? Defaults to once per second or 1Hz.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this platform.

max_led_batch_size:

Single value, type: `integer`. Default: `12`

How many LEDs can be batched on your controller? This might differ on different controllers. If in doubt check the documentation of your controller.

network_host:

Single value, type: `string`. Defaults to empty.

Host to connect when connecting to LISY via network.

network_port:

Single value, type: `integer`. Defaults to empty.

Port to connect when connecting to LISY via network.

poll_hz:

Single value, type: `integer`. Default: `100`

How fast should MPF poll LISY for switch changes? Defaults to 1000Hz

port:

Single value, type: `string`. Defaults to empty.

Serial port when connecting to LISY using serial.

send_length_after_command:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Some controllers require an additional length byte after the command.

Related How To guides

- *How to use MPF with the LISY platform*
- *Arduino Pinball Controller*

leds:

Config file section

> **Warning:** As of MPF 0.50, `matrix_lights`, `flashers` and `leds` have been combined into a single `lights` configuration. See *lights:* for details.

lights:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `lights:` section of your config is where you configure physical lights for your hardware platform.

Note: As of MPF 0.50, all lights have been combined into this single `lights` configuration. If you are using 0.33 or earlier, please see *matrix_lights:* for incandescent bulbs and *leds:* for LEDs.

Concepts

MPF supports white, single-color or multi-color lights. Traditional GIs are single white. Similar, single-color red lights are possible (i.e. red inserts). RGBW lights are possible as well. They maintain an additional white channel for better color reproduction.

To support all those different kinds of lights with a single interface for various hardware generations MPF abstracts two concepts: Light numbers and channel numbers.

Light Numbers

Configuring the `number` of a light is often the simplest way. Internally, your hardware platform will turn this into one or multiple channels (see below) depending on the `subtype` configured. For instance, if lights are usually RGB the platform will parse the number into three channels.

This is an example:

```
lights:
  my_led:
    number: 7   # might also be 8-7 or 8-1-0 depending on your platform
```

This is often the easiest way to start and will work in most cases.

Channel Numbers

Channel numbers can be configured in `channels` and describe the number for a single light channel each. This channel number is then used when the talking to the hardware. For single-color or white light this can be the same as `number`. However, for some serial LED platforms this might be also `number * 3` or a more complex conversion.

This is an example:

```
lights:
  rainbow_star:
    type: rgb
    channels:
      red:
        number: 9-29
      green:
        number: 9-30
      blue:
        number: 9-40     # this light is not sequential to the previous
```

This syntax allows the greatest flexibility but is also the most verbose one.

You can either use `channels` to arbitrarily map channels to colors or you can use `start_channel` + `type` (color order) to define the first channel and then map colors sequentially to the following channels as defined in the color order. Instead of `start_channel` you can also chain lights by configuring the `previous` light and let MPF (with help by the hardware platform) figure out the channel number.

This is an example:

```
lights:
  rainbow_star:    # this will use red: 9-29, green: 9-30 and blue: 9-31
    type: rgb
    start_channel: 9-29

  rainbow_star2:   # this will use red: 9-33, green: 9-32 and blue: 9-34
    type: grb      # notice the changed order here
    previous: rainbow_star
```

This syntax covers almost all practical cases and is beneficial with serial LEDs as the above channels syntax is very verbose. It allows the service mode to disable broken LEDs if they were removed from a serial chain. Numbers will then be recalculated omitting disabled LEDs. The syntax also works for parallel LEDs and other types of lights.

See the *documentation page of your hardware platform* for more details about numbers and channels.

Optional settings

The following sections are optional in the `lights:` section of your config. (If you don't include them, the default will be used).

channels:

Single value, type: dict. Defaults to empty.

Instead of a single `number` address for a light, you can enter channels corresponding to the multi-color channels of an RGB or RGBW LED. Each channel entry can contain any of the `lights` parameters listed on this page, but at least `number` is required.

```
lights:
  rainbow_star:
    type: rgb
    channels:
      red:
        number: 9-29
      green:
        number: 9-30
      blue:
        number: 9-31
```

Note that a light must have either `channels` or `number` defined, but cannot have both. See *LEDs* for more details about how to configure channels for different types of LEDs.

color_correction_profile:

Single value, type: `string`. Defaults to empty.

If provided, a color correction profile will be applied to all color settings this light receives. By order of operations, the light will be set to the requested color first and then the color correction profile will be applied on top.

default_on_color:

Single value, type: `color` (*color name*, *hex*, or list of values *0-255*). Default: `ffffff`

For multi-color LEDs, the color defined here will be used when the light is enabled via "on" (as opposed to being enabled with a specific color). Not intended for single-color lights.

Color values may be a hex string (e.g. *22FFCC*), a list of RGB values (e.g. *[50, 128, 206]*), or a color name (e.g. *turquoise*). MPF knows 140+ standard web color names, and you can define your own custom colors in the *named_colors:* section of your config.

fade_ms:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Defaults to empty.

When this light receives instructions to change color, it can interpolate from its current value to the new value over a fade time. If no value is provided, the machine default will be used. If this light is part of a show that defines a fade time, the show's value will supercede this light's setting.

number:

Single value, type: `string`. Defaults to empty.

This is the number of the light which specifies which output the hardware bulb or LED is physically connected to. The exact format used here will depend on which control system you're using and how the light is connected.

See the *How to configure "number:" settings* guide for details.

Note that a light must have either `channels` or `number` defined, but cannot have both.

platform:

Single value, type: `string`. Defaults to empty.

Name of the platform this LED is connected to. The default value of `None` means the default hardware platform will be used. You only need to change this if you have multiple different hardware platforms in use and this coil is not connected to the default platform.

See the *Mixing-and-Matching hardware platforms* guide for details.

There is a special platform `drivers` which will reference a driver which has to be configured in the `number` setting. It can be used if you got a light which is connected to a driver in your platform. That might be the case for *GIs* for example. This is an example for a driver as light:

```
coils:
  light_connected_to_a_driver:
    number: 42           # number depends on your platform
    allow_enable: true   # this will allow 100% enable without pwm
lights:
  light_on_a_driver:
    number: light_connected_to_a_driver    # map this light to a driver
    platform: drivers
```

platform_settings:

Single value, type: dict. Defaults to empty.

Platform-specific light settings. Consult your platform documentation for details.

previous:

Single value, type: string name of a *lights* device. Defaults to empty.

Instead of specifying the number for each light in a chain you can also use the previous setting. To do this only specify the number of the first light in the chain and then link all consequent light using the previous setting:

```
lights:
  led_0:
    number: 0
    subtype: led
    type: rgb
  led_1:
    previous: led_0
    subtype: led
    type: rgbw
  led_2:
    previous: led_1
    subtype: led
    type: rgbw
```

MPF will then calculate the number based on the light of the previous light. Make sure MPF knows how many channel each light has (i.e. by specifying the type parameter). This is not supported in all platforms but in most of them.

start_channel:

Single value, type: `string`. Defaults to empty.

In most platforms MPF will calculate the internal address of a light and how many channels it has using the number parameter. If you got unusual types of lights (such as RGBW LEDs) you can instead provide this internal address and the number of channels (i.e. using type). This is an example:

```
lights:
  led_0:
    start_channel: 0-0
    subtype: led
    type: rgbw
```

Consult the manual of your platform for details.

subtype:

Single value, type: `string`. Defaults to empty.

If you hardware platform supports multiple types of lights you need to set a `subtype` to tell your platform how to address this light (to prevent `number` collisions). Typical values are `led`, `matrix` or `gi`. Consult your platform documentation for details.

type:

Single value, type: `string`. Defaults to empty.

Default value is `rgb`.

This describes the channel order of an LED. Can be 1 to many channels (if supported by hardware). Valid channels: r (red), g (green), b (blue), w (white=minimum of red, green and blue), + (always on), - (always off).

When using serial LEDs (e.g. with FAST or Fadecandy), use *rgb* for WS2812 and *grb* for WS2811 LEDs.

x:

Single value, type: `number` (will be converted to floating point). Defaults to empty.

This is used for display_light_player to determine the position of this light on the playfield and use it as a huge display.

y:

Single value, type: `number` (will be converted to floating point). Defaults to empty.

This is used for display_light_player to determine the position of this light on the playfield and use it as a huge display.

z:

Single value, type: `number` (will be converted to floating point). Defaults to empty.

Currently not used anywhere.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

If `True`, this light will log its configuration and color changes to the debug log.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of the light in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Lights can be referenced by their tags in light_players. Typical tags are *gi* for all GIs or *playfield_inserts* for all inserts on the playfield.

Related How To guides

- *Lights*

light_segment_displays:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `platform_settings:` of your `segment_displays` section is where you map segment displays to lights when using the *light segment displays platform*.

Optional settings

The following sections are optional in the `light_segment_displays:` section of your config. (If you don't include them, the default will be used).

display_flash_duty:

Single value, type: `number` (will be converted to floating point). Default: `0.5`

For 7segment your segments are: a, b, c, d, e, f, g and dp (see: 7-Segment Displays in Wikipedia for details) For BCD your segments are: x0, x1, x2, x3 and dp (see: Binary Coded Decimal in Wikipedia for details) For 14segment your segments are: l, m, n, k, j, h, g2, g1, f, e, d, c, b, a and dp (see: 14 Segment Displays in Wikipedia for details) For 16segment your segments are: u, t, s, r, p, n, m, k, h, g, f, e, d, c, b, a and dp (see: 16 Segment Displays in Wikipedia for details)

dp is an optional decimal point per display.

display_flash_frequency:

Single value, type: `number` (will be converted to floating point). Default: `1.0`

How fast should the displays flash? Defaults to once per second or 1Hz.

Related How To guides

- *How to Connect Segment Displays as Lights to MPF*
- *Alpha-Numeric / Segment Displays*
- *Segment Display Platforms in MPF*

light_settings:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `light_settings:` section of your config is where you configure default settings for lights in your machine.

If you are using LEDs in your machine you probably want to set `default_fade_ms` to make them look softer. Otherwise, they will turn on and off very sharply and might look flickery. For instance, Stern uses a value of about 40ms for LEDs on modern machines:

```
light_settings:
  default_fade_ms: 40
```

Depending on your hardware your color might look a bit off by default. Different color channels might achive different brightnesses and white might look pinkish or blueish for example. You can set a color_correction_profile to compensate for that:

```
light_settings:
  default_color_correction_profile: correction_profile_less_red
  color_correction_profiles:
    correction_profile_less_red:
      whitepoint: [0.9, 1.0, 1.0]
      gamma: 2.5
      linear_slope: 1.0
      linear_cutoff: 0.0
```

Human perception is also not linear. Therefore, `linear_slope` is used to translate perceived brightness to brightness (you can configure that). If you see flickering at very low brightnesses you can increase `linear_cutoff` to compensate for that (see below for details).

You can also define more than one profile and configure them per *light* in the `color_correction_profile` setting. This might be useful if you use different types of lights in your machine:

```
light_settings:
  default_color_correction_profile: correction_profile_less_red
  color_correction_profiles:
    correction_profile_less_red:
      whitepoint: [0.9, 1.0, 1.0]
      gamma: 2.5
      linear_slope: 1.0
      linear_cutoff: 0.0
    correction_profile_less_blue:
      whitepoint: [1.0, 1.0, 0.9]
      gamma: 2.5
      linear_slope: 0.8
      linear_cutoff: 0.1
lights:
  special_led:
    number: 42
    color_correction_profile: correction_profile_less_blue
```

Please note, that some hardware platforms (such as the *fadecandy*) support color correction in hardware. If possible, we advice you to use the hardware correction because it gives you more dynamic range (since they use 16bit values internally).

Optional settings

The following sections are optional in the `light_settings:` section of your config. (If you don't include them, the default will be used).

color_correction_profiles:

One or more sub-entries. Each in the format of `string` : *color_correction_profile*

The `color_correction_profile:` section of your config is where you configure named color correction profiles which you can then apply to lights. You could create a single profile here which you use for all of them, or create different ones for different groups of lights.

default_color_correction_profile:

Single value, type: `string`. Defaults to empty.

The name of the color correction profile that applies to an light by default if that light doesn't have a profile configured for it.

default_fade_ms:

Single value, type: `integer`. Default: `0`

This is the default *fade_ms* that will be applied to individual lights that don't have fade_ms settings configured. If you configure an individual light's *fade_ms*, it will override this setting.

Related How To guides

- *lights:*

light_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

Note: This section can also be used in a show file in the `lights:` section of a step.

The `light_player:` section of your config is where you can control lights in config or shows. Example in config:

```
light_player:
  some_event:
    led1:
      color: red
      fade: 200ms
    led2:
      color: ff0000
      fade: 2000ms
```

```
shows:
  rainbow:
    - lights:
        (leds): red
    - lights:
        (leds): orange
    - lights:
```

(continues on next page)

(continued from previous page)

```
      (leds): yellow
  - lights:
      (leds): green
  - lights:
      (leds): blue
  - lights:
      (leds): purple
```

Optional settings

The following sections are optional in the `light_player:` section of your config. (If you don't include them, the default will be used).

color:

Single value, type: `string`. Default: `white`

Set a color to this light. Color values may be a hex string (e.g. `22FFCC`), a list of RGB values (e.g. `[50, 128, 206]`), a color name (e.g. `turquoise`), or a brightness value (i.e. `AA` or `120`). MPF knows 140+ standard web color names, and you can define your own custom colors in the *named_colors:* section of your config. If you use brightness on an RGB light MPF will use the brightness for every channel. For instance brigness `AA` will result in color `AAAAAA`.

There is a special color `stop` which will remove the current light entry from the light stack and the current show will become transparent to underlying shows as if the light has never been used in this show.

fade:

Single value, type: ms_or_token. Defaults to empty.

Time to fade this light in ms. Use this to achieve smooth transitions between colors.

priority:

Single value, type: int_or_token. Default: `0`

Relative priority of this entry in the light stack.

Related How To guides

- *Light player*

logic_blocks:

Logic blocks moved one level up in MPF 0.50. Instead of

```
logic_blocks:
  counters:
    your_counter:
      count_events: count_it_up
```

just use:

```
counters:
  your_counter:
    count_events: count_it_up
```

There are three type of logic blocks:

- *accruals:*
- *counters:*
- *sequences:*

Click each of the links above for details and settings for each type of logic block.

logging:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

In the logging section you can configure which how verbose parts of MPFs should log.

```
logging:
  console:
    asset_manager: none
    ball_controller: none
    ball_search: basic
    bcp: basic
    bcp_client: basic
    bcp_interface: basic
    bcp_server: basic
    clock: none
    config_players: none       # todo
    data_manager: none       # todo subclasses
    delay_manager: none
    device_manager: none
    event_manager: none
    file_manager: none       # todo
    logic_blocks: none
    machine_controller: basic
    mode_controller: basic
    placeholder_manager: none
    platforms: none       # todo
    players: basic       # todo
    plugins: none       # todo
    score_reel_controller: none
```

(continues on next page)

(continued from previous page)

```
    scriptlets: none       # todo
    service_controller: basic
    settings_controller: none
    show_controller: none
    switch_controller: basic
    timers: none
  file:
    asset_manager: basic
    ball_controller: basic
    ball_search: basic
    bcp: basic
    bcp_client: basic
    bcp_interface: basic
    bcp_server: basic
    clock: none
    config_players: basic
    data_manager: basic
    delay_manager: none
    device_manager: basic
    event_manager: basic
    file_manager: basic
    logic_blocks: basic
    machine_controller: basic
    mode_controller: basic
    placeholder_manager: basic
    platforms: basic
    players: full
    plugins: basic
    score_reel_controller: basic
    scriptlets: basic
    service_controller: basic
    settings_controller: basic
    show_controller: basic
    switch_controller: full
    timers: none
```

Related How To guides

Todo: *Help us to write it*

machine:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `machine:` section of your config is where you configure defails about the number of balls in your machine.

```
machine:
  balls_installed: 6
  min_balls: 3
```

Optional settings

The following sections are optional in the `machine:` section of your config. (If you don't include them, the default will be used).

balls_installed:

Single value, type: `integer`. Default: `1`

The (maximum) number of balls which should be installed in your machine.

min_balls:

Single value, type: `integer`. Default: `1`

The minimum number of balls required to start a game. If less than `min_balls` are present MPF will refuse to stat a game.

It's super annoying if you walk up to a pinball machine on location and can't start a game because it's missing a ball. So this setting lets you specify the minimum number of balls that need to be installed in order for a game to start. Note that it's up to you to make sure your game code can handle fewer balls than you might be expecting.

Related How To guides

- *How to configure an older style trough with two coils and only one ball switch*

machine_vars:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `machine_vars:` section of your machine-wide config file lets you specify the initial state of machine variables that are set when MPF starts up.

Example:

```
#config_version=5

player_vars:
  some_var:
    initial_value: 4
  some_float:
```

(continues on next page)

(continued from previous page)

```
    initial_value: 4
    value_type: float
  some_string:
    initial_value: 4
    value_type: str
  some_other_string:
    initial_value: hello
    value_type: str  # required for non-ints

machine_vars:
  test1:
    initial_value: 4
    value_type: int
  test2:
    initial_value: '5'
    value_type: str

# below is the min config we need to be able to start a game

game:
    balls_per_game: 3

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:

switches:
    s_start:
        number:
        tags: start
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch_launcher:
        number:

playfields:
    playfield:
        default_source_device: bd_launcher
        tags: default

ball_devices:
    bd_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2
        debug: true
        confirm_eject_type: target
        eject_targets: bd_launcher
        tags: trough, drain, home
    bd_launcher:
        eject_coil: eject_coil2
```

(continues on next page)

(continued from previous page)

```
        ball_switches: s_ball_switch_launcher
        debug: true
        confirm_eject_type: target
        eject_timeouts: 2s
```

Required settings

The following sections are required in the `machine_vars:` section of your config:

initial_value:

Single value, type: `string`. Defaults to empty.

The initial value of this machine variable that you're setting. This is set when MPF starts.

Optional settings

The following sections are optional in the `machine_vars:` section of your config. (If you don't include them, the default will be used).

persist:

Single value, type: `boolean` (`true`/`false`). Default: `true`

True/False value which controls whether this machine variable will be persisted to when MPF shuts down.

value_type:

Single value, type: one of the following options: str, float, int. Default: `int`

Select one of the options from this list: `int` (integer), `float`, or `str` (string). The default is "int", and there is no intelligence to try to detect which type of value you have, so if you have a floating point number or a string, you also need to set the value_type.

Related How To guides

- *Machine Variables*

magnets:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `magnets:` section of your machine config is used to define magnet mechanisms from coils and (optionally) switches. There are settings that control the timing of grabbing, releasing, and "flinging" the ball.

Example:

Listing 1: /config/config.yaml

```
#config_version=5

coils:
  magnet_coil1:
    number:
    default_pulse_ms: 100
    default_hold_power: 0.375
  magnet_coil2:
    number:
    default_pulse_ms: 100
    default_hold_power: 0.375
  magnet_coil3:
    number:
    default_pulse_ms: 100
    default_hold_power: 0.375

switches:
  grab_switch1:
    number:
  grab_switch2:
    number:
  grab_switch3:
    number:

magnets:
  magnet1:
    magnet_coil: magnet_coil1
    grab_switch: grab_switch1
    enable_events: magnet1_enable
    disable_events: magnet1_disable
    release_ball_events: magnet1_release
    fling_ball_events: magnet1_fling

  magnet_ball_save:
    magnet_coil: magnet_coil2
    grab_switch: grab_switch2
    enable_events: magnet_ball_save_enable
    disable_events: magnet_magnet_ball_save_grabbed_ball
    fling_ball_events: magnet_magnet_ball_save_grabbed_ball

  magnet_auto_enable:
    magnet_coil: magnet_coil3
    grab_switch: grab_switch3

ball_saves:
  magnet_save:
    balls_to_save: 1
    active_time: 5s
```

(continues on next page)

(continued from previous page)

```
    enable_events: magnet_magnet_ball_save_grabbing_ball
```

Required settings

The following sections are required in the `magnets:` section of your config:

magnet_coil:

Single value, type: string name of a *coils* device. Defaults to empty.

Note that is any of the magnet activation times are longer than 255ms and the magnet pulse power is 100%, then you will need to add `allow_enable: true` to the coil's entry in the `coils:` section of the machine config.

Optional settings

The following sections are optional in the `magnets:` section of your config. (If you don't include them, the default will be used).

disable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `ball_will_end, service_mode_entered`

These events mean the magnet will no longer try to grab a ball if the `grab_switch:` is activated.

enable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `ball_started`

These events enable the magnet to grab a ball based on the `grab_switch:` being activated.

fling_ball_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events to trigger flinging a ball.

fling_drop_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `250ms`

How long the magnet is deactivated for before the "fling_regrab_time" when it's flinging a ball.

fling_regrab_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `50ms`

How long the "second" (fling) pulse is for when a magnet is flinging a ball after its dropped it.

grab_ball_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

These events cause the magnet to immediately attempt to grab a ball. The magnet will be activated for the `grab_time:`.

grab_switch:

Single value, type: string name of a *switches* device. Defaults to empty.

The switch which activates grabbing a ball.

grab_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `1.5s`

How long the magnet will be energized when attempting to grab a ball.

playfield:

Single value, type: string name of a *playfields* device. Default: `playfield`

The playfield on which this magnet is.

release_ball_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

These events cause the magnet to deactivate for the `release_time:` setting.

release_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `500ms`

How long the magnet disables to release a ball.

reset_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `machine_reset_phase_3, ball_starting`

These events release a grabbed ball and disable the magnet.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

A list of tags. Not used for any logic.

Related How To guides

- *Magnets*

matrix_lights:

> **Warning:** As of MPF 0.50, `matrix_lights` and `leds` have been combined into a single `lights` configuration. See *lights:* for details.

mc_custom_code:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `mc_custom_code:` section of your config is a list where you register your custom code classes for MC.

Related How To guides

- MPF developer documentation.

mc_scriptlets:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `mc_scriptlets:` section of your config is where you list you custom code scriptlets for MC. This has been deprecated with 0.50+. Use *mc_custom_code:* instead. Scriptlets still work but will be removed eventually.

Related How To guides

- MPF developer documentation.

mode:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**YES**

The `mode:` section of a mode config file is used to specify settings for a that mode.

Note that this `mode:` section is different than the `modes:` section. (The `modes:` section is a machine-wide setting where you list all the modes that are made available to MPF when it boots up. The `mode:` section we're talking about here goes in a mode-specific config and holds the settings for that specific mode.)

Let's take a look at an example `mode:` section from a multiball mode:

```
##! mode: mode1
mode:
  start_events: ball_starting
  stop_events: timer_mode_timer_complete, shot_right_ramp
  priority: 300
```

Optional settings

The following sections are optional in the `mode:` section of your config. (If you don't include them, the default will be used).

asset_paths:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Todo: *Help us to write it*

code:

Single value, type: `string`. Defaults to empty.

If you want to write some custom Python code for this mode, you can specify the name of your file as well as the class (a child class of Mode). This entry is completely optional. If you don't need to write custom Python code for this mode (i.e. if you can do everything you need to do with config files which will probably be the case 90% of the time, then you can skip this setting.)

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this mode.

events_when_started:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

Events which will be posted when this mode has been started.

events_when_stopped:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

Events which will be posted when this mode has been stopped.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this mode.

game_mode:

Single value, type: `boolean` (`true`/`false`). Default: `true`

A mode can only access player state if `game_mode` is set to `True`. You can set this to `False` to allow a mode to run outside of a game. On example for such a mode is the attract mode. Game modes are automatically stopped at the end of a game.

path:

Single value, type: `string`. Defaults to empty.

Todo: *Help us to write it*

priority:

Single value, type: `integer`. Default: `100`

This is the numeric value that this mode will run at. (Note that this cannot be changed once the mode is running.) This priority affects two things:

- The priority order of the modes which affects the order shots and other "blockable" events are processed.
- The default priority that other things from this mode run at (shows, slides, sounds, etc.).

Our best practices are that you should have a 100-point separation between modes. (i.e. run your base mode at 100, a game mode at 200, maybe your extra ball awarded mode at 10,000, etc.) The reason for this is that with big spacing between modes, you still have room to adjust the relative priorities of things that happen within a mode without the risk of those things affecting other modes.

Warning: Keep your mode priorities between 100 and 1000000. MPF needs some built-in modes to run above and below your modes, so it has some things that run under 100 and over 1 million.

restart_on_next_ball:

Single value, type: `boolean` (`true`/`false`). Default: `false`

If you set this to *true*, a mode that was running when the ball ended that was also configured to stop on ball end will automatically start for the next ball this player has. This is managed on a per-player basis via a player variable *_restart_modes_on_next_ball* which maintains a list of the modes to be restarted.

start_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Events in this list, when posted, cause this mode to start.

If the mode is already running when one of the start events is posted, that's ok. (i.e. It won't start over or break.)

For modes that you want to start when the player's ball starts (like for your base mode, ball save, or skillshot, you'd enter *ball_starting* here. For modes that should start when some progress has been made in the game, enter the name of the event that represents when you want to start the mode. This could be the event from a shot being made, the resultant event from a logic block being completed, etc.

start_priority:

Single value, type: `integer`. Default: `0`

Allows you to fine-tune the order that modes are started in.

By default, modes register their start event handlers based on their mode priority, meaning if two modes are both configured to start on the `ball_starting` event, the higher-priority one will start first.

This `start_priority:` setting allows you to specify a relative value that will be added to the mode's `priority:` for the purpose of controlling the start order. (You can specify positive or negative values here.)

Note that the `start_priority:` setting only matters when you have multiple modes that are set to start on the same event.

stop_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Events in this list, when posted, cause the mode to stop which will remove itself from the list of active modes. All of the things you configured in this mode's config file will be unloaded. (i.e. slides and shows won't play, scoring and shot events are removed, etc.)

In the skillshot mode from the example above, there are two `stop_events:`. The first entry is the event that's posted when a timer called "mode_timer" is complete. (In this case this is a timed mode, so when that timer expires, the mode ends.) The second event is when the skillshot is made (the right ramp) in this case. (This is because once the skillshot is made, you want to remove this mode.)

If a mode is stopped and another one of the stop_events is posted, that's ok. The mode will remain stopped.

stop_on_ball_end:

Single value, type: `boolean` (`true`/`false`). Default: `true`

The default behavior for modes in MPF is that they're automatically stopped when the ball ends. Some modes (like the built-in *game* and *credit* modes) need to stay running even when the ball ends, so to support that you can add `stop_on_ball_end: false`.

Another use of this option is to retain the mode's progress towards completion after draining a ball; allowing the next player to start their ball where the previous player left off in the mode. To enable this behavior, you can add `stop_on_ball_end: false`.

However, it is very likely that a mode will be left unfinished (open) after the final ball, causing MPF to shutdown unexpectedly. You will get an error similar to this:

```
AssertionError('Mode terra_2 is not supposed to run outside of game.',)
```

To avoid this unexpected crash of MPF, add `game_ending` to the `stop_events`:

```
##! mode: mode1
mode:
  start_events: mode_terra_2_start
  stop_events: mode_complete, game_ending
  stop_on_ball_end: false
  game_mode: false
```

However, a mode with `stop_on_ball_end: False` set must be a non game mode (i.e. `game_mode: False` is also set). To prevent crashes you cannot use all player functionality (such as accessing player variable) in this mode.

stop_priority:

Single value, type: `integer`. Default: `0`

Control the order that modes stop.

By default, modes register their stop handlers at the level the mode is operating plus one. (Why +1? Because if you have one mode set to stop at an event and another mode set to start on the same event, automatically adding +1 to the stop event handler guarantees that the old mode will stop before the new mode starts.)

If you add stop priority, it's relative and added on top of the priority of the mode plus the +1. So if you have one mode you want to stop before another mode, you can simply add `stop_priority: 1` to that mode, and if other modes don't have a stop_priority set then they'll stop after it. (A higher number means that mode stops first.)

If you have a mode you want to stop last, then don't enter a *stop_priority* for it but enter *stop_priority: 1* for all the other modes you want to stop first. You can add different *stop_priority* values for different modes, and they will all stop in order, highest numeric value to lowest. Note that the *stop_priority* setting only matters when you have multiple modes that are set to end on the same stop_event.

use_wait_queue:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Specifies whether this mode should "pause" the flow of MPF while this mode is running. This only works if the mode is started via a "queue" event (something like ball_ending, game_ending, etc.). When set to true, game flow will be halted as long as this mode is running. Game flow proceeds when this mode ends.

This is useful for things like bonus modes where you want the mode to finish before the game flow moves on with the next player's turn, or modes like match or high score entry where you want those to finish before the attract mode starts again.

Related How To guides

- *How to design a game in MPF using Modes*
- *Tutorial step 14: Add your first game mode*
- *Modes*

mode_settings:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**YES**

The `mode_settings:` section of your config is a generic section that contains settings that you might want to use in a specific mode. It's nice because it's pretty much ignored by the general MPF config processing, meaning you can put whatever settings you want in here for a specific mode.

In fact, several of the built-in MPF modes make use of the `mode_settings:` section, including:

- *End of Ball Bonus mode*

Related How To guides

- *End of Ball Bonus mode*
- *Mode Selection*
- *Carousel*
- *End of Ball Bonus*

modes:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `modes:` section of your config is where you configure which modes can be loaded in your machine.

This is an example:

```
modes:
  - my_mode1
  - my_mode2
```

See *Modes* and *How to design a game in MPF using Modes* for details about modes.

Related How To guides

- *How to design a game in MPF using Modes*
- *Tutorial step 14: Add your first game mode*
- *Modes*

motors:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `motors:` section of your config is where you configure motors with position switches.

MPF supports two types of motor devices:

1. Motor can only move into one direction. The device mechanically changes the direction or moves in cycles.
2. Motor can move in two directions.

Motors devices are controlled using *digital_outputs* which can map to either light or driver outputs.

Device which can only move in one direction

This is an example for a motorized drop target bank which is mounted to a camshaft. When the motor is running it constantly moves up and down. Two position switches are used to detect the current position.

```
switches:
  s_motorized_drop_target_bank_position_up:
    number:
  s_motorized_drop_target_bank_position_down:
    number:
digital_outputs:
  c_motorized_drop_target_bank_run:
    number:
    type: driver
motors:
  motorized_drop_target_bank:
    motor_left_output: c_motorized_drop_target_bank_run
    position_switches: !!omap
      - up: s_motorized_drop_target_bank_position_up
      - down: s_motorized_drop_target_bank_position_down
```

(continues on next page)

(continued from previous page)

```
    reset_position: down
    go_to_position:
      move_bank_up: up
      move_bank_down: down
```

Device which can move in two directions

The slimer in Stern Ghostbusters is an example for a motor which can move in two directions. Both digital outputs are connected to light outputs. Again two position switches are used to detect the current position. In this setup the first and last switches are also considered as limit switches and the motor will stop once it hit one of them.

```
switches:
  s_slimer_home:
    number: 8-1
  s_slimer_away:
    number: 8-2
digital_outputs:
  c_slimer_motor_forward:
    number: 8-3
    type: light
  c_slimer_motor_backward:
    number: 8-4
    type: light
motors:
  ghostbusters_slimer:
    motor_left_output: c_slimer_motor_forward
    motor_right_output: c_slimer_motor_backward
    position_switches: !!omap
      - home: s_slimer_home
      - away: s_slimer_away
    reset_position: home
    go_to_position:
      slimer_home: home
      slimer_away: away
```

Another example of such a device would be the claw in Stern Batman DK (or also Stern Batman 66). It has more position switches but the mechanics are similar:

```
switches:
  s_claw_home:
    number:
  s_claw_position1:
    number:
  s_claw_position2:
    number:
  s_claw_position3:
    number:
  s_claw_position4:
    number:
  s_claw_position5:
    number:
```

(continues on next page)

(continued from previous page)

```
digital_outputs:
  c_claw_forward:
    number:
    type: driver
  c_claw_backward:
    number:
    type: driver
motors:
  batman_claw:
    motor_left_output: c_claw_forward
    motor_right_output: c_claw_backward
    position_switches: !!omap
      - home: s_claw_home
      - pos1: s_claw_position1
      - pos2: s_claw_position2
      - pos3: s_claw_position3
      - pos4: s_claw_position4
      - pos5: s_claw_position5
    reset_position: home
    go_to_position:
      stop_claw: home
      go_pos1: pos1
      go_pos2: pos2
      go_pos3: pos3
      go_pos4: pos4
      go_pos5: pos5
```

Required settings

The following sections are required in the `motors:` section of your config:

position_switches:

Ordered list for one (or more) sub-settings. Each in the format of `string` : string name of a *switches* device

Ordered map of name of the position and the switch which becomes active once this position is reached.

For example:

```
position_switches:  !!omap
    - home: s_claw_home
    - pos1: s_claw_position1
    - pos2: s_claw_position2
```

`home`, `pos1` and `pos2` are the names of your positions (you can choose them freely). `s_claw_home`, `s_claw_position1` and `s_claw_position2` are the switches to detect the position.

The order is important when the motor can move in two directions. For instance, if the device is at `home` and should move to `pos1` it will move right. However, if it is at `pos2` it will move left.

The same position logic applies when working with a motor that has a "home" position on the right instead of the left:

```
position_switches:  !!omap
    - pos2: s_claw_position2
    - pos1: s_claw_position1
    - home: s_claw_home
```

If it is not at any position and also does not know its previous position it will move left until it reaches a known position and may then change its direction again (usually this should not happen since it will move to a known position during reset).

reset_position:

Single value, type: `string`. Defaults to empty.

The position the device should move to on reset (as defined in `position_switches`).

Optional settings

The following sections are optional in the `motors:` section of your config. (If you don't include them, the default will be used).

go_to_position:

One or more sub-entries. Each in the format of `string` : `string`

A mapping of events to positions. Once an event in the mapping is posted the motor will move to the corresponding position.

For instance:

```
go_to_position:
    stop_claw: home
    go_pos1: pos1
    go_pos2: pos2
```

If you post `stop_claw` the motor will move to the position called `home` (as defined in `position_switches`).

include_in_ball_search:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Whether the motor should be included in ball search.

motor_left_output:

Single value, type: string name of a *digital_outputs* device. Defaults to empty.

Digital output to enable to move the motor left. You need to configure at least `motor_left_output` or `motor_right_output` if you motor can only move in one direction or both if it can move in both directions.

motor_right_output:

Single value, type: string name of a *digital_outputs* device. Defaults to empty.

Digital output to enable to move the motor right. You need to configure at least `motor_left_output` or `motor_right_output` if you motor can only move in one direction or both if it can move in both directions.

reset_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `machine_reset_phase_3, ball_starting`

Events on which the motor should move to its `reset_position`. You usually do not have to configure this.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Not used.

Related How To guides

- *Motors*

mpf:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `mpf:` section of your config is where you configure global MPF settings.

Optional settings

The following sections are optional in the `mpf:` section of your config. (If you don't include them, the default will be used).

allow_invalid_config_sections:

Single value, type: `boolean` (`true`/`false`). Default: `false`

MPF will not raise a fatal error when on invalid section when you set this to true. This might be useful when you are developing a new feature and do not want to constantly update config_spec (the file which describes allowed sections).

auto_create_switch_events:

Single value, type: `boolean` (`true`/`false`). Default: `true`

MPF will post switch_event_active and switch_event_inactive (see below) when this is enabled.

config_players:

Unknown type. See description below.

A list of config players which will be loaded.

core_modules:

Unknown type. See description below.

A list of core modules which will be loaded.

default_ball_search:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Default value for whether ball search is enabled or disabled on all playfields (unless you overwrite it on that playfield).

default_light_hw_update_hz:

Single value, type: `integer`. Default: `50`

Default light update hz. Can be overwritten per platform.

default_platform_hz:

Single value, type: `number` (will be converted to floating point). Default: `100`

For all non-tickless platforms we poll this often. This usually means how often we will read switches. Reducing this setting might reduce the amounts of CPU significantly. We recommand to keep this at least at 50Hz or you will loose switch hits. For smooth game play aim at 100Hz. Everything above that will mostly only reduce switch latency.

default_pulse_ms:

Single value, type: `integer`. Default: `10`

Default default_pulse_ms for all coils when not overwritten. This will be used when you do not specify any pulse_ms in your coil.

default_show_sync_ms:

Single value, type: `integer`. Default: `0`

Default sync_mc for all shows when not specified otherwise.

default_timed_enable_ms:

Single value, type: `integer`. Default: `0`

Todo: *Help us to write it*

---

device_modules:

Unknown type. See description below.

A list of device modules which will be loaded.

paths:

Unknown type. See description below.

Paths for all additional files loaded in MPF.

platforms:

Unknown type. See description below.

A list of platforms which will be loaded.

plugins:

Unknown type. See description below.

A list of plugins which will be loaded.

report_crashes:

Single value, type: one of the following options: ask, never, always. Default: `ask`

Todo: *Help us to write it*

---

save_machine_vars_to_disk:

Single value, type: `boolean` (`true`/`false`). Default: `true`

If set to true MPF will persist machine_vars to disk in a background writer.

switch_event_active:

Single value, type: `string`. Default: `%_active`

If auto_create_switch_events is set to true this event will be posted after a switch turned active.

switch_event_inactive:

Single value, type: `string`. Default: `%_inactive`

If auto_create_switch_events is set to true this event will be posted after a switch turned inactive.

switch_tag_event:

Single value, type: `string`. Default: `sw_%`

This event will be posted for all tags after a switch turned active.

Related How To guides

- *How to configure LEDs (FAST Pinball)*

mpf-mc:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `mpf-mc:` section of your config is where you configure options for the MC itself.

Required Settings

All of these settings are required in the `mpf-mc:` section. However, MPF-MC includes a default config file called `mcconfig.yaml` which includes all these settings with their defaults. So you only need to add/enter these if you want to change something from the default.

bcp_port:

Single integer value, default is `5050`.

This is the TCP port that the MC listens on for incoming BCP connections. If you change this from the

bcp_interface:

String, default is `localhost`.

The interface to bind for the BCP connection.

fps:

Single integer value, default is `30`.

Limit the frames per second to `fps`. This prevents using excessive CPU for MPF MC.

allow_invalid_config_sections:

Single boolean value, default is `True`.

Allow sections which are not known to MPF.

multiball_locks:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**YES**

The `multiball_locks:` section of your config is used to configure ball locks which will lock balls for multiball. Note that if you only want to hold a ball temporarily (like to play a show for an award) and then release it, use the *ball_holds:* section instead.

Multiball lock devices are smart. They work with physical ball devices but track the number of balls locked virtually which is not necessarily the same as the number of balls that are physically contained in a ball device.

When a ball is locked, it will add a new ball into play from the ball device which is set in `default_source_device` of your playfield unless the device that just locked it is full, in which case it will eject a ball from the full device. The events that control the ball ejections are queue events, so you can interrupt the delivery of a new ball with the *queue_relay_player:* (for example, to have a mode selection screen before returning to play).

Whenever a new ball is locked, the event *multiball_lock_<name>_locked_ball* is posted with an argument "total_balls_locked". When the lock is full, it will post *multiball_lock_<name>_full*, which you can use as a start event for a related *multiballs:* to start multiball. (And since the multiball lock tracks the "virtual" ball lock count on a per-player basis, this will still work even if another player previously emptied out the lock. (In that case, the multiball will add any additional balls it needs from the trough.)

Here's an example:

```
ball_devices:
  bd_bunker:
    eject_coil: c_eject
    ball_switches: s_ball1
##! mode: mode1
multiball_locks:
  bunker:
    balls_to_lock: 3
    lock_devices: bd_bunker
```

Each sub-entry under the `multiball_locks:` section is the name of the multiball lock ("bunker") in the example above. Then each named ball lock has the following settings:

Required settings

The following sections are required in the `multiball_locks:` section of your config:

balls_to_lock:

Single value, type: `integer`. Defaults to empty.

The number of balls this ball lock should hold. If one of the associated lock devices receives a ball and this logical ball lock is full, then the ball device will just release the ball again.

lock_devices:

List of one (or more) values, each is a type: string name of a *ball_devices* device. Defaults to empty.

A list of one (or more) ball devices that will collect balls which will count towards this lock.

Optional settings

The following sections are optional in the `multiball_locks:` section of your config. (If you don't include them, the default will be used).

balls_to_replace:

Single value, type: `integer`. Default: `-1`

By default a multiball lock will immediately replace every ball it locks with a new ball from the default device (i.e. the trough). With this setting you can instruct the lock to replace only up to a certain number of locked balls. A value of 0 means the lock will never replace balls, and -1 means it will always replace balls (default).

This setting is useful for machines that physically lock multiple balls in a lock and replace them from the trough. When a full lock starts a multiball, for example, you may not want the game to add another ball from the trough. Usually this setting will be used in tandem with replace-balls-in-play from *multiballs:*.

Caution: an improperly configured setting can lead the player to a state where no balls are active on the playfield and the game becomes stuck. See *How to create a multiball with a traditional ball lock* for instructions and examples.

blocking_facility:

Single value, type: `string`. Defaults to empty.

Todo: *Help us to write it*

---

disable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None` (Note that if you add an entry here, it will replace the default. So if you also want the default value(s) to apply, add them too.)

Event(s) which disable this ball lock, meaning that balls that enter one of the lock devices don't count towards the lock. If you want to set up a ball lock that requires the player to "re-light" the lock after locking a ball, you can set this ball lock's "ball_locked" event as a disable event for this lock and then set some other shot that re-enables the lock as an enable event.

empty_lock_devices_on_ball_end:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Todo: *Help us to write it*

---

enable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None` (Note that if you add an entry here, it will replace the default. So if you also want the default value(s) to apply, add them too.)

Event(s) which enable this ball lock. If this multiball lock is disabled, then a ball entering one of its ball devices does not count towards the lock. You can use this in situations where a player has to hit some other shot to first re-light the lock before a ball can be locked. (In that case you'd use the event posted by the light lock shot as one of the enable_events here.

locked_ball_counting_strategy:

Single value, type: one of the following options: virtual_only, min_virtual_physical, physical_only, no_virtual. Default: `virtual_only`

See the *general multiball lock documentation* for an explanation of how each of these works.

priority:

Single value, type: `integer`. Default: `1`

Relative priority when claiming balls entering a device. This can be used to give one *ball_hold* or *multiball_lock* preference when claiming balls.

reset_all_counts_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Event(s) which reset the locked ball counts for all players.

reset_count_for_current_player_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Event(s) which reset the locked ball count for the current player.

source_devices:

List of one (or more) values, each is a type: string name of a *ball_devices* device. Defaults to empty.

Select the source device to use when replacing balls. By default this will use the device defined in lock_devices. If this setting is defined and the defined device does not have a ball the lock will fall back to the default playfield source device.

source_playfield:

Single value, type: string name of a *ball_devices* device. Default: `playfield`

The name of the playfield that feeds balls to this lock. If you only have one playfield (which is most games), you can leave this setting out. Default is the playfield called *playfield*.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

See the *documentation on the debug setting* for details.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Not used.

Related How To guides

- *Multiballs*

multiballs:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `multiballs:` section of your config is where you can configure multiball devices. Multiball devices are "abstract" devices in that they're more of a concept rather than a physical device on the playfield. The multiball "device" is used to start multiball. This section can be used in your machine-wide config files. This section can be used in mode-specific config files.

Here's an example which contains several different multiball configs. (In the real world, you'd probably only have one multiball for each mode.)

```
multiballs:
  add_a_ball:
    ball_count: 1
    ball_count_type: add
    shoot_again: 30s
    enable_events: mb4_enable
    disable_events: mb4_disable
    start_events: mb4_start
    stop_events: mb4_stop
  quick_2_ball:
    ball_count: 2
    ball_count_type: total
    shoot_again: 20s
    start_events: mb11_start
    ball_locks: bd_lock
  release_all_locked_balls:
    ball_count: current_player.lock_mb6_locked_balls
    ball_count_type: add
    shoot_again: 20s
    start_events: mb12_start
    ball_locks: bd_lock
  quick_add_2_ball:
    ball_count: 2
    ball_count_type: add
    shoot_again: 0
    start_events: mb6_start
    ball_locks: bd_lock
  full_ball_save:
    ball_count: 2
    shoot_again: 30s
    hurry_up_time: 10s
    grace_period: 5s
    add_a_ball_events: add_ball
    add_a_ball_shoot_again: 20s
    add_a_ball_hurry_up_time: 5s
    add_a_ball_grace_period: 10s
    start_events: mb20_start
```

Required settings

The following sections are required in the `multiballs:` section of your config:

ball_count:

Single value, type: `integer` or `template` (*Instructions for entering templates*). Defaults to empty.

The number of balls this multiball should eject (and maintain during shoot again period). This is a template so you can use *dynamic values* to calculate this during runtime.

Optional settings

The following sections are optional in the `multiballs:` section of your config. (If you don't include them, the default will be used).

add_a_ball_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, will add one ball into play. Posting an event multiple times will add one ball for each time the event is posted.

This is useful for "add-a-ball" functionality (which you can combine with a counter and/or conditional events if you want to cap how many total balls can be added into play).

add_a_ball_grace_period:

Single value, type: `time string (ms) or template` (*Instructions for entering time strings* and *Instructions for entering templates*). Default: `0`

The "secret" time (in MPF time string format) the ball save is still active after any shows or effects that are triggered end. This is added onto the add_a_ball_shoot_again.

add_a_ball_hurry_up_time:

Single value, type: `time string (ms) or template` (*Instructions for entering time strings* and *Instructions for entering templates*). Default: `0`

The time before the add-a-ball ball save ends (in MPF time string format) that will cause the multiball_<name>_hurry_up event to be posted. Use this to change the script for the light or trigger other effect.

add_a_ball_shoot_again:

Single value, type: `time string (ms) or template` (*Instructions for entering time strings* and *Instructions for entering templates*). Default: `5s`

Specifies a time period for "shoot again" when an add-a-ball event is posted. This is a sort of automatic ball save for multiballs. The timer will start when this multiball starts, and any balls that drain during this time will be re-added into play.

ball_count_type:

Single value, type: one of the following options: add, total. Default: `total`

Set this to either `total` or `add`. Default is `total`.

This setting controls the behavior of how the multiball calculates the number of balls it should add into play. Adjusting this setting is useful when you have multiple (or stacked) multiballs and you want to control how the combined counts work.

total Means the `ball_count:` setting will provide a target for the total number of balls that should be in play when this multiball starts. So if this multiball has a `ball_count: 3`, and it starts when 2 balls are live on the playfield, then this multiball will only add 1 more ball to bring the total to 3.

add Means that the `ball_count:` setting will specify the number of balls that are added into play on top of whatever number of balls are already in play. So if this multiball is set to `ball_count: 2` and there are already 2 balls in play, then this multiball will add 2 more balls for a total of 4 balls live.

ball_locks:

List of one (or more) values, each is a type: string name of a *ball_devices* device. Defaults to empty.

Use those devices first when ejecting balls to the playfield on multiball start. On start all balls from all locks will be ejected (maybe more than ball_count). If there are not enough balls in the lock more balls will be requested to the source_playfield.

disable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, disable this multiball. When disabled, the other events (like start and add a ball) do not work. If this multiball is in a mode config, then it will also be disabled when the mode it's in stops.

enable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, enable this multiball. Note that enabling a multiball is not the same as starting it, but the other events (like to start the multiball or, or add a ball, etc.) do not work unless this multiball is enabled.

Note that if you do not add any `enable_events:` (which is the default), this multiball will be automatically enabled when the mode it's in starts.

grace_period:

Single value, type: `time string (ms) or template` (*Instructions for entering time strings* and *Instructions for entering templates*). Default: `0`

The "secret" time (in MPF time string format) the ball save is still active after any shows or effects that are triggered end. This is added onto the shoot_again.

hurry_up_time:

Single value, type: `time string (ms) or template` (*Instructions for entering time strings* and *Instructions for entering templates*). Default: `0`

The time before the ball save ends (in MPF time string format) that will cause the multiball_<name>_hurry_up event to be posted. Use this to change the script for the light or trigger other effect.

replace_balls_in_play:

Single value, type: `boolean` (`true`/`false`). Default: `false`

This setting controls whether the multiball should include existing balls in play when counting the number of balls to add to the playfield. Specifically for machines which physically lock multiple balls, this setting should be used in tandem with balls-to-replace from *multiball_locks:* to accurately populate the multiball when it starts.

See *How to create a multiball with a traditional ball lock* for detailed instructions on using this setting.

reset_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `machine_reset_phase_3, ball_starting`

Event(s) that reset this multiball, which means they disable it as well as disabling shoot again and resetting the ball add counts to 0.

shoot_again:

Single value, type: `time string (ms) or template` (*Instructions for entering time strings* and *Instructions for entering templates*). Default: `10s`

Specifies a time period for "shoot again" which is a sort of automatic ball save for multiballs. The timer will start when this multiball starts, and any balls that drain during this time will be re-added into play.

source_playfield:

Single value, type: string name of a *ball_devices* device. Default: `playfield`

The name of the playfield (from the `playfields:` section of your machine config that this multiball will add balls to. You don't have to worry about this unless you have multiple playfields that you're managing separately (which is rare, usually only in head-to-head type games).

start_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, start the multiball. Note that these events will only have an effect if this multiball is enabled.

start_or_add_a_ball_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, will either start the multiball, or, if it's started, will add another ball.

stop_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events in this list, when posted, stop the multiball. If there are multiball balls on the playfield, there's nothing that can be done about that (unless you want to disable the flippers). However stopping the multiball will cut off the "shoot again" period.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

See the *documentation on the debug setting* for details.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Unused.

Related How To guides

- *Multiballs*

mypinballs:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `mypinballs:` section of your config is where your mypinballs segment display controller. See *MyPinballs Segment Display Controller* for details.

Required settings

The following sections are required in the `mypinballs:` section of your config:

port:

Single value, type: `string`. Defaults to empty.

Serial port to use.

Optional settings

The following sections are optional in the `mypinballs:` section of your config. (If you don't include them, the default will be used).

baud:

Single value, type: `integer`. Default: `115200`

Baud rate to use on the serial port.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set to true to see more debug output.

Related How To guides

- *MyPinballs Segment Display Controller*

named_colors:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The named_colors: section of your config is where you define color names that can be used for RGB lights throughout your machine code. Anywhere in lights: or light_player: where a color can be specified, named colors can be used.

Your named colors can be an array of R/G/B values or a hex string of hex values (which can also include a brightness percentage, like all hex color strings).

This is an example:

```
named_colors:
  custom_blue: [24, 65, 226]
  troll_green: 4a9b22
  troll_green_dark: 4a9b22%50
lights:
  troll_target:
    number: 10
    default_on_color: troll_green
  l_jackpot:
    number: 20
light_player:
  trolls_disabled:
    troll_target: troll_green_dark
  jackpot_lit:
    l_jackpot:
      color: custom_blue
      fade: 10
```

Related How To guides

Todo: *Help us to write it*

open_pixel_control:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The open_pixel_control: section of your config is where you configure a openpixel light controller. This is usually used together with a *fadecandy* but can also be used standalone. Usually, you don't have to change anything.

Optional settings

The following sections are optional in the open_pixel_control: section of your config. (If you don't include them, the default will be used).

console_log:

Single value, type: one of the following options: none, basic, full. Default: `none`

Log level for the console log for this platform.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see more debug log lines.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this platform.

host:

Single value, type: `string`. Default: `localhost`

Hostname of the openpixel server to connect.

port:

Single value, type: `integer`. Default: `7890`

Port of the openpixel server to connect.

Related How To guides

- *How to configure a FadeCandy RGB LED Controller*

opp:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `opp:` section of your config is where you configure your OPP platform. See *How to configure Open Pinball Project (OPP) hardware for MPF* for details.

This is an example:

```
hardware:
  platform: opp
  driverboards: gen2
opp:
  ports: COM7
```

Required settings

The following sections are required in the `opp:` section of your config:

ports:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Serial ports to use.

Optional settings

The following sections are optional in the `opp:` section of your config. (If you don't include them, the default will be used).

baud:

Single value, type: `integer`. Default: `115200`

Baud rate to use on the serial port.

chains:

One or more sub-entries. Each in the format of `string` : `string`

This is an example:

```
opp:
  ports: /dev/ttyOPP0, /dev/ttyOPP1
  chains:
    0: /dev/ttyOPP0
    1: /dev/ttyOPP1
```

If you switch was number `1-3` before it will be `0-1-3` or `1-1-3` afterwards.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `none`

Log level for the console log for this platform.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true if you want to see more debug output.

driverboards:

Single value, type: one of the following options: gen2. Default: `gen2`

Similar to `driverboards` in the *hardware:* section. Use this setting if you use multiple playforms (i.e. FAST and OPP) in one machine.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this platform.

incand_update_hz:

Single value, type: `integer`. Default: `25`

The update rate for incandecant bulbs. Do not set this too high or you might saturate the OPP bus.

poll_hz:

Single value, type: `integer`. Default: `100`

How many times per section the OPP hardware is polled for switch changes. Default is 100.

Related How To guides

- *How to configure Open Pinball Project (OPP) hardware for MPF*

opp_coils:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `opp_coils:` section of your config is where you configure platform specific settings for OPP coils.

Optional settings

The following sections are optional in the `opp_coils:` section of your config. (If you don't include them, the default will be used).

recycle_factor:

Single value, type: `integer`. Default: `None`

The recycle_factor is used in OPP to determine the cool down time of a coil after a pulse in relation to default_pulse_ms. For instance, with recycle_factor of 2 and a default_pulse_ms of 20ms the coil will cool down for at least 40ms after each pulse.

osc:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `osc:` section of your config is where you configure the *osc platform*.

Optional settings

The following sections are optional in the `osc:` section of your config. (If you don't include them, the default will be used).

events_to_send:

List of one (or more) events. The device will add handlers for those events. Defaults to empty.

You can list all events which you want to be forwarded to your OSC remote. This is an example:

```
hardware:
  platform: osc

osc:
  remote_ip: 127.0.0.1
  remote_port: 8000

  events_to_send:
    - player_score
    - some_non_osc_switch_active
    - some_non_osc_switch_inactive
```

listen_ip:

Single value, type: `string`. Default: `127.0.0.1`

The IP MPF should use to listen for incoming UDP OSC connections. You can also set this to `0.0.0.0` if you want MPF to listen on all interfaces instead of just on loopback (local connections only).

listen_port:

Single value, type: `integer`. Default: `9000`

The port MPF shoud use to listen for incoming UDP OSC connections.

remote_ip:

Single value, type: `string`. Default: `127.0.0.1`

The IP address of your remote OSC server. MPF will send all messages to this IP.

remote_port:

Single value, type: `integer`. Default: `8000`

The port of your remote OSC server.

Related How To guides

- *How to use MPF with OSC Devices or Hardware*

p_roc:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `p_roc:` section of your config is where you configure hardware specific bits about the P-Roc or P3-Roc. In most cases you can omit this config and stick with the defaults.

Optional settings

The following sections are optional in the `p_roc:` section of your config. (If you don't include them, the default will be used).

console_log:

Single value, type: one of the following options: none, basic, full. Default: `none`

Log level for the console log for this platform.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to `True` if you want to know what is going on under the hood. We will usually ask you to set this if you experience any hardware related problems and send us your log.

display_flash_duty:

Single value, type: `number` (will be converted to floating point). Default: `0.5`

Todo: *Help us to write it*

display_flash_frequency:

Single value, type: `number` (will be converted to floating point). Default: `1.0`

Todo: *Help us to write it*

dmd_timing_cycles:

List of one (or more) values, each is a type: `integer`. Defaults to empty.

Only P-Roc (not P3-Roc).

Those values determine the timing to drive the different shades of your DMD. See *How to configure mono/traditional DMD (P-ROC)* for details.

dmd_update_interval:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `33ms`

Only P-Roc (not P3-Roc).

The update interval of your DMD. Usually you do not have to change this.

driverboards:

Single value, type: one of the following options: wpc, wpcAlphanumeric, wpc95, sternSAM, sternWhitestar, pdb, custom, None. Defaults to empty.

Similar to `driverboards` in the *hardware:* section. Use this setting if you use multiple playforms (i.e. FAST and P3-Roc) in one machine.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this platform.

lamp_matrix_strobe_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `100ms`

Default: `100ms`

The column strobe time for your lamp matrix. See *How to configure Matrix Lights (P-ROC/P3-ROC)* for details.

pd_led_boards:

One or more sub-entries. Each in the format of `integer` : *pd_led_boards*

A map of PD-LED boards with their ID as key and a *configuration map* as value. This can be used to configure indivdual features per board.

See *Servos on a PD-LED (P-ROC/P3-ROC)*, *Steppers on a PD-LED (P-ROC/P3-ROC)* or *How to configure LEDs on the PD-LED (P-ROC/P3-ROC)* for details.

trace_bus:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Log all calls to libpinproc. This will cause a lot of additional log lines and might considerably slow down MPF. Use only during debugging.

use_separate_thread:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Whether MPF should spawn a separate thread to talk to the P/P3-Roc or not. If you set this to `False` any IO to the P/P3-Roc will block the game loop which might cause lags unrelated to the hardware. This has a small overhead but should be enabled in most cases.

use_watchdog:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Enable or disable the watchdog. Usually you want to keep this enabled.

watchdog_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `1s`

Watchdog timeout. The P/P3-Roc will disable all coils when the watchdog expires.

Related How To guides

- *How to configure Multimorphic (P-ROC & P3-ROC) hardware*

pd_led_boards:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

The `pd_led_boards:` section of your config is where you configure your PD-LED boards connected to your *P-Roc or P3-Roc*. See *Servos on a PD-LED (P-ROC/P3-ROC)*, *Steppers on a PD-LED (P-ROC/P3-ROC)* or *How to configure LEDs on the PD-LED (P-ROC/P3-ROC)* for details.

Optional settings

The following sections are optional in the `pd_led_boards:` section of your config. (If you don't include them, the default will be used).

lpd880x_0_first_address:

Single value, type: `integer`. Default: `100`

First LED address to map to lpd880x_0. This will be the LED number on the PD-LED for your first LED in the chain. If you set this to 100 it will be the first LED in your chain. 101 will be the second in chain and so on.

lpd880x_0_last_address:

Single value, type: `integer`. Default: `249`

Last LED address to map to lpd880x_0. This will determine how many LEDs map to your chain. The more LEDs you have in your chain the lower the update rate will be.

lpd880x_1_first_address:

Single value, type: `integer`. Default: `250`

First LED address to map to lpd880x_1. This will be the LED number on the PD-LED for your first LED in the chain. If you set this to 100 it will be the first LED in your chain. 101 will be the second in chain and so on.

lpd880x_1_last_address:

Single value, type: `integer`. Default: `399`

Last LED address to map to lpd880x_1. This will determine how many LEDs map to your chain. The more LEDs you have in your chain the lower the update rate will be.

lpd880x_2_first_address:

Single value, type: `integer`. Default: `400`

First LED address to map to lpd880x_2. This will be the LED number on the PD-LED for your first LED in the chain. If you set this to 100 it will be the first LED in your chain. 101 will be the second in chain and so on.

lpd880x_2_last_address:

Single value, type: `integer`. Default: `549`

Last LED address to map to lpd880x_2. This will determine how many LEDs map to your chain. The more LEDs you have in your chain the lower the update rate will be.

max_servo_value:

Single value, type: `integer`. Default: `250`

Max clock cycles in a servo duty cycle. 300 will rougly map to 2ms.

stepper_speed:

Single value, type: `integer`. Default: `13524`

Clock cycles for a stepper half step (at 32MHz). This might need some tuning depending on your stepper.

use_lpd880x_0:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Enable the first LPD880x serial LED chain on connector J8 pin 13 (clock) and pin 14 (data). If you enable this you cannot use LEDs 79 and 80 on the board.

use_lpd880x_1:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Enable the second LPD880x serial LED chain on connector J8 pin 9 (clock) and pin 12 (data). If you enable this you cannot use LEDs 77 and 78 on the board.

use_lpd880x_2:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Enable the third LPD880x serial LED chain on connector J8 pin 7 (clock) and pin 8 (data). If you enable this you cannot use LEDs 75 and 76 on the board.

use_servo_0:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Set to true to enable servo 0 on connector J8 pin 2. If you enable this you cannot use LED 72 on the board.

use_servo_1:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Set to true to enable servo 1 on connector J8 pin 3. If you enable this you cannot use LED 73 on the board.

use_servo_10:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Set to true to enable servo 10 on connector J8 pin 18. If you enable this you cannot use LED 82 on the board.

use_servo_11:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Set to true to enable servo 11 on connector J8 pin 19. If you enable this you cannot use LED 83 on the board.

use_servo_2:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Set to true to enable servo 2 on connector J8 pin 4. If you enable this you cannot use LED 74 on the board.

use_servo_3:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Set to true to enable servo 3 on connector J8 pin 7. If you enable this you cannot use LED 75 on the board.

use_servo_4:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Set to true to enable servo 4 on connector J8 pin 8. If you enable this you cannot use LED 76 on the board.

use_servo_5:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Set to true to enable servo 5 on connector J8 pin 9. If you enable this you cannot use LED 77 on the board.

use_servo_6:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Set to true to enable servo 6 on connector J8 pin 12. If you enable this you cannot use LED 78 on the board.

use_servo_7:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Set to true to enable servo 7 on connector J8 pin 13. If you enable this you cannot use LED 79 on the board.

use_servo_8:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Set to true to enable servo 8 on connector J8 pin 14. If you enable this you cannot use LED 80 on the board.

use_servo_9:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Set to true to enable servo 9 on connector J8 pin 17. If you enable this you cannot use LED 81 on the board.

use_stepper_0:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Set to true to enable stepper 0 on connector J8 pin 12 (sleep), pin 13 (pulse) and pin 14 (direction). If you enable this you cannot use LEDs 78, 79 and 80 on the board.

use_stepper_1:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Set to true to enable stepper 1 on connector J8 pin 7 (sleep), pin 8 (pulse) and pin 9 (direction). If you enable this you cannot use LEDs 75, 76 and 77 on the board.

use_ws281x_0:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Enable the first WS281x serial LED chain on connector J8 pin 19. If you enable this you cannot use LED 83 on the board.

use_ws281x_1:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Enable the second WS281x serial LED chain on connector J8 pin 18. If you enable this you cannot use LED 82 on the board.

use_ws281x_2:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Enable the third WS281x serial LED chain on connector J8 pin 17. If you enable this you cannot use LED 81 on the board.

ws281x_0_first_address:

Single value, type: `integer`. Default: `100`

First LED address to map to ws281x_0. This will be the LED number on the PD-LED for your first LED in the chain. If you set this to 100 it will be the first LED in your chain. 101 will be the second in chain and so on.

ws281x_0_last_address:

Single value, type: `integer`. Default: `249`

Last LED address to map to ws281x_0. This will determine how many LEDs map to your chain. The more LEDs you have in your chain the lower the update rate will be.

ws281x_1_first_address:

Single value, type: `integer`. Default: `250`

First LED address to map to ws281x_1. This will be the LED number on the PD-LED for your first LED in the chain. If you set this to 100 it will be the first LED in your chain. 101 will be the second in chain and so on.

ws281x_1_last_address:

Single value, type: `integer`. Default: `399`

Last LED address to map to ws281x_1. This will determine how many LEDs map to your chain. The more LEDs you have in your chain the lower the update rate will be.

ws281x_2_first_address:

Single value, type: `integer`. Default: `400`

First LED address to map to ws281x_2. This will be the LED number on the PD-LED for your first LED in the chain. If you set this to 100 it will be the first LED in your chain. 101 will be the second in chain and so on.

ws281x_2_last_address:

Single value, type: `integer`. Default: `599`

Last LED address to map to ws281x_2. This will determine how many LEDs map to your chain. The more LEDs you have in your chain the lower the update rate will be.

ws281x_end_bit_time:

Single value, type: `integer`. Default: `40`

Clock cycles for the end bit in a WS281x chain (at 32MHz). Usually this does not have to be changed.

ws281x_high_bit_time:

Single value, type: `integer`. Default: `24`

Clock cycles for a high bit in a WS281x chain (at 32MHz). Usually this does not have to be changed.

ws281x_low_bit_time:

Single value, type: `integer`. Default: `13`

Clock cycles for a low bit in a WS281x chain (at 32MHz). Usually this does not have to be changed.

ws281x_reset_bit_time:

Single value, type: `integer`. Default: `1603`

Clock cycles for a reset bit in a WS281x chain (at 32MHz). Usually this does not have to be changed.

pin2dmd:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `pin2dmd:` section of your config is where you configure your *PIN2DMD RGB DMD display*.

Optional settings

The following sections are optional in the `pin2dmd:` section of your config. (If you don't include them, the default will be used).

console_log:

Single value, type: one of the following options: none, basic, full. Default: `none`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

panel:

Single value, type: one of the following options: rgb, rbg. Default: `rgb`

The order of the LEDs in your panels. If your blue and green appear to be swapped change this.

resolution:

Single value, type: one of the following options: 128x32, 192x64. Default: `128x32`

The resolution of your panel. PIN2DMD XL is `192x64` and the standard PIN2DMD is `128x32`.

Related How To guides

- *How to configure a PIN2DMD RGB LED DMD*

pkone:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `pkone:` section of your machine-wide config is where you configure hardware options that are specific to the Penny K Pinball PKONE Controller. Note that we have a how to guide which includes *all* *the PKONE-specific settings* throughout your entire config file, so be sure to read that if you have Penny K Pinball PKONE hardware.

```
pkone:
  port: com3
```

Required settings

The following sections are required in the `pkone:` section of your config:

port:

Single value, type: `string`. Defaults to empty.

The serial port name your PKONE controller uses.

Optional settings

The following sections are optional in the `pkone:` section of your config. (If you don't include them, the default will be used).

baud:

Single value, type: `integer`. Default: `115200`

Baud rate to use on the serial port.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `none`

Log level for the console log for this platform.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

See the *documentation on the debug setting* for details.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this platform.

watchdog:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `1000`

The PKONE controller includes a “watchdog” timer. A watchdog is a timer that is continuously counting down towards zero, and if it ever hits zero, the controller shuts off all the power to the drivers and turns off all the lights. The idea is that every time MPF runs a game loop (so, 30 times a second or whatever), MPF tells the FAST controller to reset the watchdog timer. So this timer is constantly getting reset and never hits zero.

But if MPF crashes or loses communication with the PKONE hardware, then this watchdog timer won’t be reset. When it hits zero, the PKONE controller will kill the power to the coils and servos and turn off all lights. This should prevent an MPF crash from burning up a coil or somehow damaging your hardware in another way.

You can set the watchdog timer to whatever you want (up to 10 seconds). This is essentially the maximum time a coil could be stuck “on” if MPF crashes. The default is 1 second which is probably fine for almost everyone, and you don’t have to include this section in your config if you want to use the default.

Related How To guides

- *How to configure MPF for Penny K Pinball PKONE hardware*

player_vars:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `player_vars:` section of your machine-wide config file lets you specify the initial state of player variables that are set for a player when the game starts.

Example:

```
#config_version=5

player_vars:
  some_var:
    initial_value: 4
  some_float:
    initial_value: 4
    value_type: float
  some_string:
    initial_value: 4
    value_type: str
  some_other_string:
    initial_value: hello
    value_type: str  # required for non-ints

machine_vars:
```

(continues on next page)

(continued from previous page)

```
  test1:
    initial_value: 4
    value_type: int
  test2:
    initial_value: '5'
    value_type: str

# below is the min config we need to be able to start a game

game:
    balls_per_game: 3

coils:
    eject_coil1:
        number:
    eject_coil2:
        number:

switches:
    s_start:
        number:
        tags: start
    s_ball_switch1:
        number:
    s_ball_switch2:
        number:
    s_ball_switch_launcher:
        number:

playfields:
    playfield:
        default_source_device: bd_launcher
        tags: default

ball_devices:
    bd_trough:
        eject_coil: eject_coil1
        ball_switches: s_ball_switch1, s_ball_switch2
        debug: true
        confirm_eject_type: target
        eject_targets: bd_launcher
        tags: trough, drain, home
    bd_launcher:
        eject_coil: eject_coil2
        ball_switches: s_ball_switch_launcher
        debug: true
        confirm_eject_type: target
        eject_timeouts: 2s
```

Required settings

The following sections are required in the `player_vars:` section of your config:

initial_value:

Single value, type: `string`. Defaults to empty.

The initial value of this player variable that you're setting. This is set when the player is created.

Optional settings

The following sections are optional in the `player_vars:` section of your config. (If you don't include them, the default will be used).

value_type:

Single value, type: one of the following options: str, float, int. Default: `int`

Select one of the options from this list: `int` (integer), `float`, or `str` (string). The default is "int", and there is no intelligence to try to detect which type of value you have, so if you have a floating point number or a string, you also need to set the value_type.

Related How To guides

- *Tutorial step 15: Add scoring*
- *Player Variables Reference*
- *Scoring*
- *Player Variables*

playfield_transfers:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `playfield_transfers:` section of your config is where you configure devices which transfer balls between *playfields*.

This is an example:

```
switches:
  s_transfer:
    number:
playfield_transfers:
  transfer1:
    ball_switch: s_transfer
    captures_from: playfield1
    eject_target: playfield2
  transfer2:
    transfer_events: transfer_ball
```

(continues on next page)

(continued from previous page)

```
    captures_from: playfield1
    eject_target: playfield2
playfields:
  playfield1:
    label: Playfield 1
    default_source_device: None
  playfield2:
    label: Playfield 2
    default_source_device: None
```

Required settings

The following sections are required in the `playfield_transfers:` section of your config:

captures_from:

Single value, type: string name of a *ball_devices* device. Defaults to empty.

Source playfield for the transfer.

eject_target:

Single value, type: string name of a *ball_devices* device. Defaults to empty.

Target playfield for the transfer.

Optional settings

The following sections are optional in the `playfield_transfers:` section of your config. (If you don't include them, the default will be used).

ball_switch:

Single value, type: string name of a *switches* device. Defaults to empty.

Ball switch which triggers the transfer.

transfer_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Events in this list, when posted, will trigger a ball transfer.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see more debug output.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Tags of the device. Not used currently.

Related How To guides

- *Playfields*

playfields:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `playfields:` section of your config is where you configure your *playfields* in your machine. You can have multiple playfields and MPF will track balls per playfield. One playfield should contain the tag *default* so that the game knows which playfield to use.

Required settings

The following sections are required in the `playfields:` section of your config:

default_source_device:

Single value, type: string name of a *ball_devices* device. Defaults to empty.

The source ball device to use to feed balls to this playfield. This source device must be able to eject directly to the playfield. Usually this is your launcher ball device. If you do not have a launcher use the trough device.

Optional settings

The following sections are optional in the `playfields:` section of your config. (If you don't include them, the default will be used).

ball_search_block_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `flipper_cradle`

Event to block ball search. Used by flipper cradle.

ball_search_disable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Event to disable ball search.

ball_search_enable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Event to enable ball search.

ball_search_failed_action:

Single value, type: `string`. Default: `new_ball`

When ball search failed this action is taken. Either `new_ball` which will eject a new ball from the default default source device or `end_game` which will end the game.

ball_search_interval:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `150ms`

The delay after each fired coil/searched device.

ball_search_phase_1_searches:

Single value, type: `integer`. Default: `3`

Ball search will run in multiple phases with increasing intensity. For instance, in phase 1, only ball devices without a ball will be pulsed. This defines how many time phase 1 is repeated until ball_search proceeds to phase 2.

ball_search_phase_2_searches:

Single value, type: `integer`. Default: `3`

Ball search will run in multiple phases with increasing intensity. For instance, in phase 2, all ball devices except the trough will try to dejam. This defines how many time phase 2 is repeated until ball_search proceeds to phase 3.

ball_search_phase_3_searches:

Single value, type: `integer`. Default: `4`

Ball search will run in multiple phases with increasing intensity. For instance, in phase 3, all ball devices except the trough pulse their coil. This defines how many time phase 3 is repeated until ball search gives up.

ball_search_timeout:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `15s`

ball_search_timeout configures the time of inactivity which has to pass until ball search starts.

ball_search_unblock_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `flipper_cradle_release`

Event to unblock ball search. Used by flipper cradle.

ball_search_wait_after_iteration:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `5s`

Extra delay after each iteration.

enable_ball_search:

Single value, type: `boolean` (`true`/`false`). Defaults to empty.

Enable ball_search by default. Use with care during development since coils may hurt you. Should be enabled in any production machine.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Turn on/off debugging.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Label for service menu.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Set tag *default* to your default playfield. The game will use the default playfield to eject balls.

Related How To guides

- *Playfields*

playlist_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

Note: This section can also be used in a show file in the `playlists:` section of a step.

The `playlist_player:` section of your config is where you specify actions to perform on playlists when MPF events are received. Additional information may be found in the *playlist_player* documentation.

Examples:

```
playlist_player:
  play_attract_music:
    playlist:
      playlist: attract_music
      action: play
  advance_playlist:
    playlist:
      action: advance
  stop_playlist:
    playlist:
      action: stop
```

Basic usage:

```
playlist_player:
  <triggering_event_name>:
    <playlist track name>:
      action: <action name>
      <optional settings>
  <triggering_event_name>:
    <playlist track name>:
      action: <action name>
      <optional settings>
```

Optional settings

The following sections are optional in the `playlist_player:` section of your config. (If you don't include them, the default will be used).

action:

Single value, type: one of the following options: play, stop, advance, set_repeat. Default: `play`

Todo: *Help us to write it*

Related How To guides

Todo: *Help us to write it*

playlists:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The playlists: section of your config is where you configure non-default parameter values for any playlist assets you want to use in your game. (This section is part of the MPF media controller and only available if you're using MPF-MC for your media controller.)

Here is an example:

```
# ---------------------
# SOUNDS::PLAYLIST
# ---------------------
playlists:
  playlistIntro:
    shuffle: false
    repeat: false
    sounds:
      - voiceAnnouncerNewsFlash1
      - voiceAnnouncerMessage1
      - voiceAnnouncerAliensAttack1
  playlistHighScore:
    shuffle: true
    repeat: true
    crossfade_mode: override
    crossfade_time: 5s
    sounds:
      - soundHighScore001
      - soundHighScore002
      - soundHighScore003
      - soundHighScore004
# ---------------------
# PLAYLIST::PLAYER
# ---------------------
playlist_player:
  # -------------------
  # ADDED SURPRISE VOICE DURING ATTRACT MODE
  playlistAttention:
    trackplaylist:
      playlist: playlistIntro
      action: play
  # -------------------
  # MUSIC DURING HIGH SCORE ENTRY
  high_score_enter_initials:
    trackplaylist:
      playlist: playlistHighScore
      shuffle: true
      repeat: true
      action: play
  mode_attract_started:
    trackplaylist:
      action: stop
```

Required settings

The following sections are required in the playlists: section of your config:

sounds:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Note: If you want to use a sound that has spaces in its name, the name of the sound must be in quotes:

```
playlists:
  mode_music:
    sounds:
      - song_01
      - song_02
      - "song 03" # example of a sound with a space in its name using quotes
      - song_04
```

Optional settings

The following sections are optional in the `playlists:` section of your config. (If you don't include them, the default will be used).

crossfade_mode:

Single value, type: one of the following options: use_track_setting, override. Default: `use_track_setting`

The `crossfade_mode:` of a playlist determines whether the playlist uses the track `crossfade_time` setting or the `crossfade_time` specified in the playlist. Options for `crossfade_mode:` are:

- `use_track_settings` - Use the `crossfade_time` specified in the playlist track.
- `override` - Use the `crossfade_time` specified in the playlist.

crossfade_time:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `0`

The number of seconds over which to crossfade between sounds in the playlist. This value is ignored when `crossfade_mode:` is set to `use_track_setting`.

events_when_looping:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A list of one or more names of events that MPF will post when this playlist loops back to the beginning while playing. The playlist will only loop if `repeat:` is set to `True`. Enter the list in the MPF config list format. These events are posted exactly as they're entered.

events_when_played:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A list of one or more names of events that MPF will post when this playlist is played. Enter the list in the MPF config list format. These events are posted exactly as they're entered.

events_when_sound_changed:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A list of one or more names of events that MPF will post when a new sound is played while the playlist is played. Enter the list in the MPF config list format. These events are posted exactly as they're entered.

events_when_sound_stopped:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A list of one or more names of events that MPF will post when a playlist sound has finished playing. Enter the list in the MPF config list format. These events are posted exactly as they're entered.

events_when_stopped:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A list of one or more names of events that MPF will post when this playlist has finished playing. Enter the list in the MPF config list format. These events are posted exactly as they're entered.

repeat:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Flag indicating whether or not the playlist will repeat when all sounds have been played or just stop.

scope:

Single value, type: one of the following options: machine, player. Default: `machine`

Whatever this playlist should be persisted per player or machine-wide.

shuffle:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Flag indicating whether or not the playlist will be played in order (`shuffle: True` or randomized (`shuffle: False`) for playback.

Related How To guides

- *Playlist player*

plugins:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `plugins:` section of your config is where you list all plugin classes to load. By default it contains:

- *info_lights:*
- *switch_player:*
- *auditor:*

pololu_maestro:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `pololu_maestro:` section of your config is where you configure the serial port that a Pololu Maestro servo controller is connected to.

When you attach a Pololu Maestro, two serial ports will appear. You want to specific the first (lower numbered) port here. For example:

```
pololu_maestro:
  port: COM5
```

Note that there are a few other settings you need to configure in other areas to use a Pololu Maestro servo controller. See the *How To guide* for details.

Required settings

The following sections are required in the `pololu_maestro:` section of your config:

port:

Single value, type: `string`. Defaults to empty.

The name of the serial port.

Optional settings

The following sections are optional in the `pololu_maestro:` section of your config. (If you don't include them, the default will be used).

console_log:

Single value, type: one of the following options: none, basic, full. Default: `none`

Log level for the console log for this platform.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

See the *documentation on the debug setting* for details.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this platform.

servo_max:

Single value, type: `integer`. Default: `9000`

Quarter-microseconds to use for the max servo value. The default (9000) translates to 2250 microseconds or 2.25ms.

servo_min:

Single value, type: `integer`. Default: `3000`

Quarter-microseconds to use for the max servo value. The default (3000) translates to 750 microseconds or 0.75ms.

Related How To guides

- *Pololu Maestro Servo Controller*

pololu_tic:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `pololu_tic:` section of your config is where you configure your *Pololu Tic Stepper Controller*.

See *tic_stepper_settings:* for `platform_settings` in your steppers.

Optional settings

The following sections are optional in the `pololu_tic:` section of your config. (If you don't include them, the default will be used).

console_log:

Single value, type: one of the following options: none, basic, full. Default: `none`

Log level for the console log for this platform.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this platform.

Related How To guides

- *How to use Pololu Tic in MPF*

psus:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `psus:` section of your config is where you power supply units. See *Voltages and Power in Pinball Machines* for details about voltages in pinball machines and some electric details. Then specify to which PSU your *coils* are connected. This is used for power management. In some cases, MPF ,ay deliberately delay coil pulses to prevent too many coils from firing and drawing to much current from your PSU. *Ball devices* and *drop target* do this by default to ensure more consistent pulses.

Optional settings

The following sections are optional in the `psus:` section of your config. (If you don't include them, the default will be used).

max_amps:

Single value, type: `integer`. Defaults to empty.

Maximum ampers which can be provided by this PSU. Currently not used.

release_wait_ms:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `10`

Time to wait after a coil pulse.

voltage:

Single value, type: `integer`. Defaults to empty.

Voltage of your PSU. Only informal.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Currently unused.

Related How To guides

- *Power Management in Software*

queue_event_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

Note: This section can also be used in a show file in the `queue_events:` section of a step.

The `queue_event_player:` section of your config file is similar to the `event_player:`, except it posts *queue events* instead of regular events.

This section is particularly useful with the *queue_relay_player:*.

Here's an example:

```
queue_event_player:
  some_event:
    queue_event: my_queue
    events_when_finished: my_queue_done
```

In the example above, when the regular event *some_event* is posted, a new queue event called *my_queue* will be posted. After all the handlers for *my_queue* are done, the event *my_queue_done* will be posted. (This could be immediately if none of the handlers blocked it, or it could be awhile if one of those handlers is doing something else first.)

Required settings

The following sections are required in the `queue_event_player:` section of your config:

queue_event:

Single event. The device will add an handler for this event. Defaults to empty.

The name of the queue event that will be posted when the parent event is posted. (required)

Optional settings

The following sections are optional in the `queue_event_player:` section of your config. (If you don't include them, the default will be used).

args:

One or more sub-entries. Each in the format of `string` : `string`

A sub-configuration of key:value pairs that will be posted with the event. This setting is optional.

events_when_finished:

Single event. This device will be posted by the device. Defaults to empty.

The event name that will be posted when all the handlers of this queue event are done processing it. This setting is optional.

Related How To guides

- *How to Drain All Balls on the Playfield and Serve One Back Without Ending the Current Ball*
- *Queue Event player*

queue_relay_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The queue_relay_player lets you "pause" queue event processing until some other event is posted, at which time the original queue event processing continues.

Here's an example:

```
queue_relay_player:
  game_ending:
    post: start_my_mode
    wait_for: my_mode_done
```

This entry will watch for the *game_ending* event to be posted. (*game_ending* is a queue event.) When it's posted, the queue relay player will pause the processing of the *game_ending* event and post a new event, the *start_my_mode* in this case.

You can use that new event to do whatever you want, like start some custom mode you want to run at game end before the machine goes back to the attract mode.

When your mode is done, you would configure it to post *my_mode_done* (or whatever the `wait_for:` is set to, and that will release the queue and progress will continue. If your mode doesn't need to do anything, it can simply post the `wait_for:` event and exit.

> **Warning:** If the `wait_for:` event is never posted, you will break your game since MPF will wait forever.

Note that each entry under `queue_event_player:` (the `game_ending:` in the example above) must be for a queue event. (You can see which events are queue events in the *event reference*.) You can also use the *queue_event_player:* to "convert" a regular event into a queue event.

Required settings

The following sections are required in the `queue_relay_player:` section of your config:

post:

Single value, type: `string`. Defaults to empty.

The name of the event to post to trigger your action once the queue event has been posted. (required)

wait_for:

Single value, type: `string`. Defaults to empty.

The name of the event this queue will wait for to continue. In other words, this is the event you need to post for the queue event to continue. (required)

Optional settings

The following sections are optional in the `queue_relay_player:` section of your config. (If you don't include them, the default will be used).

args:

One or more sub-entries. Each in the format of `string` : `string`

A sub-configuration of key:value pairs that will be posted with the event. This setting is optional.

pass_args:

Single value, type: `boolean` (`true`/`false`). Default: `false`

If `True` pass on the arguments from the event in `wait_for` to the event posted in `post`.

Related How To guides

- *Queue Relay player*
- *Mode Selection*
- *Game End Modes*
- *How to Drain All Balls on the Playfield and Serve One Back Without Ending the Current Ball*

random_event_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

Note: This section can also be used in a show file in the `random_events:` section of a step.

The `random_event_player:` section of your config is where you can play a random event out of a list based on an event.

```
# in your global config:
random_event_player:
  play_random_event_global:
    scope: machine
    events:
      - event1
      - event2
      - event3
##! mode: base
# in your mode:
random_event_player:
  play_random_event:
    events:
      - event1
      - event2
      - event3
  play_random_event_with_weight:
    events:
      unlikely_event1: 2
      unlikely_event2: 3
      likely_event1: 45
      likely_event2: 50
  play_random_event_with_weight_and_conditional:
    events:
      event1{mode.field.active}: 25
      event2{device.ball_devices.bd_ramp_lock.balls==2}: 25
      event3{device.accruals.base_locking_engaged.completed}: 10
      event4{device.counters.health.value>9}: 30
      event5{current_player.hearts < current_player.hearts_max}: 10
    fallback_event: event_posts_if_everything_above_false
```

Optional settings

The following sections are optional in the `random_event_player:` section of your config. (If you don't include them, the default will be used).

disable_random:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Disable random.

events:

Unknown type. See description below.

List the events to choose from. If you use a list all events will be equiprobable. You can also use a dict with `eventname: probablity`. See the example above.

You can also use *conditional events* here.

fallback_event:

Single value, type: `string`. Defaults to empty.

If all of the events in the random_event_player are conditional and none of them are true, this event name will be posted instead. If not defined, no event will be posted.

force_all:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Enforce that all events are posted once before a event is posted a second time.

force_different:

Single value, type: `boolean` (`true`/`false`). Default: `true`

If set to true it will enforce that the same entry will never appear twice in a row. When setting `force_all` to true this will prevent that the last event is the same as the first of the next iteration.

scope:

Single value, type: one of the following options: player, machine. Default: `player`

The scope of the random selection for `force_different` and `force_all`. When setting to `player` this is enforced per player and persisted between balls.

Related How To guides

- *Random event player*
- *Mystery Awards*

raspberry_pi:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `raspberry_pi:` section of your config is where you configure your Raspberry Pi running pigpio. See *Raspberry PI (pigpio)* for details.

This is an example:

```
raspberry_pi:
  ip: localhost
  port: 8888
```

Required settings

The following sections are required in the `raspberry_pi:` section of your config:

ip:

Single value, type: `string`. Defaults to empty.

IP of your Raspberry Pi. MPF will connect to this IP. Hostname does not work here.

Optional settings

The following sections are optional in the `raspberry_pi:` section of your config. (If you don't include them, the default will be used).

console_log:

Single value, type: one of the following options: none, basic, full. Default: `none`

Log level for the console log for this platform.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this platform.

port:

Single value, type: `integer`. Default: `8888`

Port of the pigpio daemon on your Raspberry Pi (in case you change it).

Related How To guides

- *Raspberry PI (pigpio)*

rgb_dmds:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `rgb_dmds:` section of your machine config is where you configure the settings for physical RGB DMDs (dot matrix displays). You only need this section if you have a RGB DMD connected via USB. If you have a mono DMD, configure that in the *dmds:* section.

If you want to show a virtual RGB DMD in an on-screen window, you configure that as a display widget with a dot filter. You don't need to use this `rgb_dmds:` section to do that.

Note there are no *height* and *width* settings here. The pixel size of your DMD is determined by the size of the `source:` display which drives the content for this DMD.

Here's an example:

```
displays:
  dmd:
    width: 128
    height: 32
rgb_dmds:
  smartmatrix:  # name of this DMD which can be whatever you want
    hardware_brightness: .5
    fps: 25
    gamma: 2.5
```

Note that this section is called `rgb_dmds:` (plural). Just like "switches" and "coils" and most everything else in MPF, this is a section that contains all your DMDs. Now since this is a DMD, you probably only have one, (though MPF can support as many as you want), but it's important to note that you add a `rgb_dmds:` section to your config, then under that you add an entry for a specific DMD (which can be whatever you want), and then you enter one or more of the following settings:

Optional settings

The following sections are optional in the `rgb_dmds:` section of your config. (If you don't include them, the default will be used).

brightness:

Single value, type: `number` (will be converted to floating point). Default: `1.0`

Brightness value multiplied in software (as an OpenGL shader in MC). Using `hardware_brightness` is preferred if your hardware supports it.

channel_order:

Single value, type: `string` (case-insensitive). Default: `rgb`

Channel order of your rgb dmd. Change this if colors are swapped on your hardware. Any order (such as `rgb`, `grb`, `brg` and so) will work.

fps:

Single value, type: `integer`. Default: `30`

How many frames per second this DMD will be updated. Note that some RGB DMDs cannot handle the full 30fps, so you might have to dial this back to around 25 or so or else the DMD won't be able to keep up and will get behind.

gamma:

Single value, type: `number` (will be converted to floating point). Default: `2.2`

Sets the gamma of the DMD. See *Gamma correction in MPF* for details.

Note that the default setting of `2.2` will probably be ok, though if your RGB DMD does its own internal gamma correction, you'll want to set the gamma to `1.0` (which is effectively disabling it).

Note that gamma is closely related to brightness (below). You'll probably want to adjust both of them together.

Important: Gamma setting is important!

We can't stress enough that setting the gamma for your DMD is important for making it look right. So click the link above and make the adjustment. It's a one-time thing.

---

hardware_brightness:

Single value, type: `number` or `template` (will be converted to floating point; *Instructions for entering templates*). Default: `1.0`

A brightness multiplier for the DMD (because RGB DMDs are crazy bright). Note that brightness is closely related to gamma (see above). You'll probably want to adjust both of them together.

Also note that you can use *dynamic values* here if you want to do math or use settings to make this configurable.

only_send_changes:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Specifies whether every frame is sent to the DMD, or only changed frames.

platform:

Single value, type: `string`. Defaults to empty.

Name of the platform this DMD is connected to. The default value of `None` means the default hardware platform will be used. You only need to change this if you have multiple different hardware platforms in use and this coil is not connected to the default platform.

See the *Mixing-and-Matching hardware platforms* guide for details.

source_display:

Single value, type: `string`. Default: `dmd`

The name of the display (from the `displays:` section of your machine config) that is the source for this physical DMD. Whatever's on the source display will be displayed on the DMD. If you don't specify a source, MPF will automatically use a source display called "dmd".

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Not used currently.

Related How To guides

- *DMD Platforms in MPF*
- *Using an RGB full-color LED DMD*

rpi_dmd:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `rpi_dmd:` section of your config is where you configure a *RPi DMD*. All settings are directly passed to the rpi-rgb-led-matrix library. Read their documentation (or the source) if in doubt.

Optional settings

The following sections are optional in the `rpi_dmd:` section of your config. (If you don't include them, the default will be used).

brightness:

Single value, type: `integer`. Default: `100`

Brightness value to use between 0 and 100%.

chain_length:

Single value, type: `integer`. Default: `1`

Number of panels in your chain. Longer chains mean less frames per second.

cols:

Single value, type: `integer`. Default: `32`

How many columns of LEDs does your matrix have?

daemon:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Leave this at `False` of thing will go wrong.

disable_hardware_pulsing:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Disable hardware pulsing. Only useful for debugging.

drop_privileges:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Drop root rights after opening the hardware. It is highly recommended to leave it this way.

gpio_slowdown:

Single value, type: `integer`. Default: `1`

Slow down the GPIOs a bit. Otherwise the RPi might be too fast for your LEDs.

hardware_mapping:

Single value, type: one of the following options: regular, adafruit-hat, adafruit-hat-pwm. Default: `regular`

Select which hardware you are using. Consult manual if in doubt.

inverse_colors:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Inverse colors. You know it if you see it.

led_rgb_sequence:

Single value, type: `string`. Default: `RGB`

The color order of your LEDs. You know it if you see that colors are mixed up.

multiplexing:

Single value, type: `integer`. Default: `0`

Select your multiplexing settings. Consult manual.

parallel:

Single value, type: `integer`. Default: `1`

How many chains to run in parallel?

pixel_mapper_config:

Single value, type: `string`. Default: `""`

Select your pixel mapper. Consult manual.

pwm_bits:

Single value, type: `integer`. Default: `11`

How many bits to use for PWM?

pwm_lsb_nanoseconds:

Single value, type: `integer`. Default: `130`

On-time for your LEDs. Lower means more fps but potential less quality.

row_address_type:

Single value, type: `integer`. Default: `0`

Row address type. Consult manual.

rows:

Single value, type: `integer`. Default: `32`

How many rows of LEDs does your matrix have?

scan_mode:

Single value, type: `integer`. Default: `0`

Scan mode. 0 = progressive; 1 = interlaced. Consult manual.

show_refresh_rate:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Print refresh rate on terminal.

Related How To guides

- *Raspberry PI DMD (rpi-rgb-led-matrix)*

score_queue_player:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**YES**

The `score_queue_player:` section of your config is where you configure your SS style scoring. This is an example:

```
coils:
  c_chime_1000:
    number:
  c_chime_100:
    number:
  c_chime_10:
    number:
score_queues:
  score:
    chimes: c_chime_1000, c_chime_100, c_chime_10,  None
    debug: true
##! mode: my_mode
# in your mode
score_queue_player:
  score_2k:
    score: 2000
  score_200:
    score: 200
```

Optional settings

The following sections are optional in the `score_queue_player:` section of your config. (If you don't include them, the default will be used).

int:

Single value, type: `integer` or `template` (*Instructions for entering templates*). Defaults to empty.

Score value to add to the queue.

Related How To guides

- *How to implement solid state game style score queues in MPF*

score_queues:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `score_queues:` section of your config is where you configure SS style scoring queues in MPF. This is an example:

```
coils:
  c_chime_1000:
    number:
  c_chime_100:
    number:
  c_chime_10:
    number:
score_queues:
  score:
    chimes: c_chime_1000, c_chime_100, c_chime_10,  None
##! mode: my_mode
# in your mode
score_queue_player:
  score_2k:
    score: 2000
  score_200:
    score: 200
```

Required settings

The following sections are required in the `score_queues:` section of your config:

chimes:

List of one (or more) values, each is a type: string name of a *coils* device. Defaults to empty.

A list of chimes to pulse when adding score via the score queue. Start from the left the right on your digits. You might use None if a certain digit does not have a chime. Example: `c_chime_1000, c_chime_100, c_chime_10, None`

Optional settings

The following sections are optional in the `score_queues:` section of your config. (If you don't include them, the default will be used).

delay:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `200ms`

The delay between adding scores (and pulsing a chime).

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Not used.

Related How To guides

- *How to implement solid state game style score queues in MPF*

score_reel_groups:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `score_reel_groups:` section of your config is where you configure groups of *score reels*. Every reel only displays one digits so they have to be grouped to display longer scores. See *How to Configure Score Reels* for more details.

Required settings

The following sections are required in the `score_reel_groups:` section of your config:

reels:

List of one (or more) values, each is a type: string name of a *score_reels* device. Defaults to empty.

List the *score reels* which make up this group. Start with the highest digit. The last entry will be the right most digit. You may use None if there is no reel for a digit.

Optional settings

The following sections are optional in the `score_reel_groups:` section of your config. (If you don't include them, the default will be used).

chimes:

List of one (or more) values, each is a type: string name of a *coils* device. Defaults to empty.

List the *coils* driving the chime which are rung when the reel overflows. Start with the highest digit. The last entry will be the right most digit. You may use None if there is no chime for a digit.

lights_tag:

Single value, type: `string`. Defaults to empty.

Lights to turn on when this group is active.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Tag groups with the player which uses it. Add `player1` to use this reel for player 1. Use `player2` for player 2 and so on. A reel can be used for more than one player.

Related How To guides

- *How to Configure Score Reels*

score_reels:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `score_reels:` section of your config is where you configure your score reels. See *How to Configure Score Reels* for more details.

Optional settings

The following sections are optional in the `score_reels:` section of your config. (If you don't include them, the default will be used).

coil_inc:

Single value, type: string name of a *coils* device. Defaults to empty.

Coil to fire to increment this reel.

hw_confirm_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `20`

How long does the switch have to stay active until counted.

limit_hi:

Single value, type: `integer`. Default: `9`

The highest digit on your reel.

limit_lo:

Single value, type: `integer`. Default: `0`

The lowest digit on your reel.

repeat_pulse_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `200`

How long to wait after a pulse before pulsing the coil again.

switch_0:

Single value, type: string name of a *switches* device. Defaults to empty.

Switch which indicates that the reel is showing a 0.

switch_1:

Single value, type: string name of a *switches* device. Defaults to empty.

Switch which indicates that the reel is showing a 1.

switch_10:

Single value, type: string name of a *switches* device. Defaults to empty.

Switch which indicates that the reel is showing a 10.

switch_11:

Single value, type: string name of a *switches* device. Defaults to empty.

Switch which indicates that the reel is showing a 11.

switch_12:

Single value, type: string name of a *switches* device. Defaults to empty.

Switch which indicates that the reel is showing a 12.

switch_2:

Single value, type: string name of a *switches* device. Defaults to empty.

Switch which indicates that the reel is showing a 2.

switch_3:

Single value, type: string name of a *switches* device. Defaults to empty.

Switch which indicates that the reel is showing a 3.

switch_4:

Single value, type: string name of a *switches* device. Defaults to empty.

Switch which indicates that the reel is showing a 4.

switch_5:

Single value, type: string name of a *switches* device. Defaults to empty.

Switch which indicates that the reel is showing a 5.

switch_6:

Single value, type: string name of a *switches* device. Defaults to empty.

Switch which indicates that the reel is showing a 6.

switch_7:

Single value, type: string name of a *switches* device. Defaults to empty.

Switch which indicates that the reel is showing a 7.

switch_8:

Single value, type: string name of a *switches* device. Defaults to empty.

Switch which indicates that the reel is showing a 8.

switch_9:

Single value, type: string name of a *switches* device. Defaults to empty.

Switch which indicates that the reel is showing a 9.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set to true to get more debug output in the log.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Tags of this reel.

Related How To guides

- *How to Configure Score Reels*

scriptlets:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `scriptlets:` section of your config is where you list you custom code scriptlets. This has been deprecated with 0.50+. Use *custom_code:* instead. Scriptlets still work but will be removed eventually.

Related How To guides

- MPF developer documentation.

segment_display_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

Note: This section can also be used in a show file in the `segment_displays:` section of a step.

The `segment_display_player:` section of your config is a *Config Players* which controls *segment_displays:*. See *Alpha-Numeric / Segment Displays* for details.

Optional settings

The following sections are optional in the `segment_display_player:` section of your config. (If you don't include them, the default will be used).

action:

Single value, type: one of the following options: add, remove, flash, no_flash, flash_match, flash_mask, set_color. Default: `add`

- `add` - Add a text to the segment_display.
- `remove` - Remove a text from the segment_display by key. If a `transition_out:` setting is used, then that transition will be started.
- `no_flash` - Stop flashing this segment display.
- `flash` - Flash this segment display.
- `flash_match` - Flash the last two characters of the segment display.
- `flash_mask` - Use the `flash_mask` parameter value to determine which characters of the segment display to flash.
- `set_color` - Set the color(s) of the characters in the segment display (for platforms that support it).

color:

List of one (or more) values, each is a type: `color` (*color name*, *hex*, or list of values *0*-*255*). Defaults to empty.

The color for each character in the display (if the platform supports it). If a single color is supplied, all characters in the display will be set to that color. See *Specifying Colors in Config Files* for more information on specifying colors in config files.

expire:

Single value, type: ms_or_token. Defaults to empty.

Only used with `action` `add`. Text will be removed after `expire` ms.

flash_mask:

Single value, type: `string`. Defaults to empty.

Only used with the `flash_mask` action (or with `add` when the `flashing` parameter is set to `mask`. Determines which characters of the segment display will be flashed. Each character of the flash mask string represents a character in the display. Character positions with an `F` character (must be upper-case) will be flashed while positions containing any other character will not flash. For example, in a segment display of length 16, to flash the first 8 characters use a `flash_mask` parameter value of `FFFFFFFF________`. You can use whatever character you wish for the non-flashing character positions.

flashing:

Single value, type: one of the following options: off, all, match, mask, not_set. Default: `not_set`

- `off` - Stop flashing this segment display.
- `all` - Flash all characters in this segment display.
- `match` - Flash the last two characters of the segment display.
- `mask` - Use the `flash_mask` parameter value to determine which characters of the segment display to flash.

Only used with the `add` action.

key:

Single value, type: `string`. Defaults to empty.

Key to use with `action` `add` and `remove` to reference a text on the segment display.

priority:

Single value, type: int_or_token. Default: `0`

Priority of this text. The segment display will maintain a stack and show the text on top (highest priority).

text:

Single value, type: `string`. Defaults to empty.

Text to show. You can use *Text Templates*.

transition:

Unknown type. See description below.

Note: Be sure the `segment_display` `size` parameter has been properly set for the segment display or the transition effects may not be calculated and displayed properly.

transition_out:

Unknown type. See description below.

```
segment_display_player:
  jackpot_completed:
    display1:
      text: JACKPOT
      priority: 1000
      expire: 2s
      transition:
        type: push
        direction: right
        text: " *** "
      transition_out:
        type: push
        direction: right
        text: " *** "
```

There can only be one transition between text entries, so if outgoing text has a `transition_out` set, and an incoming text entry has a `transition` set, then the incoming transition will take precedence.

Related How To guides

- *Alpha-Numeric / Segment Displays*
- *Segment Display Platforms in MPF*

segment_displays:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `segment_displays:` section of your config is where you define your *segment displays*. This can be 7-segment or alphanumeric displays which are typically used in older machines.

Required settings

The following sections are required in the `segment_displays:` section of your config:

number:

Single value, type: `string`. Defaults to empty.

The number of the display. The meaning depends on the hardware platform.

Optional settings

The following sections are optional in the `segment_displays:` section of your config. (If you don't include them, the default will be used).

default_color:

List of one (or more) values, each is a type: `color` (*color name*, *hex*, or list of values *0-255*). Default: `white`

Todo: *Help us to write it*

---

default_transition_update_hz:

Single value, type: float_or_token. Default: `30`

The speed (steps per second) at which text transition effects will be updated in the display.

integrated_commas:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Determines whether or not the physical segment display has integrated commas in each character rather than taking up an entire character. When set to `true`, commas are collapsed with the preceding character when calculating text transition effects.

integrated_dots:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Determines whether or not the physical segment display has integrated dots/periods in each character rather than taking up an entire character. When set to `true`, dots/periods are collapsed with the preceding character when calculating text transition effects.

platform:

Single value, type: `string`. Defaults to empty.

This can be used to overwrite the platform which is defined in the *hardware* section for segment_displays.

platform_settings:

Single value, type: dict. Defaults to empty.

Platform specific settings. See your *segment platform documentation*.

size:

Single value, type: `integer`. Default: `7`

The number of characters in the segment display. This value should be set to match the number of characters in your physical hardware (or virtual emulator). It is important to set this number correctly for the text transition effects.

use_dots_for_commas:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Todo: *Help us to write it*

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Not used.

Related How To guides

- *Alpha-Numeric / Segment Displays*
- *Segment Display Platforms in MPF*

servo_controllers:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `servo_controllers:` section of your config is where you configure PCA9685/PCA9635-based I2C servo controllers. See *I2C Servo Controllers* for details.

I2C Address

When you configure an I2C servo controller you have to address it on the I2C bus. The default of the chip is `0x40` which is `64` in decimal. There might be some prefix depending on your I2C interface.

Optional settings

The following sections are optional in the `servo_controllers:` section of your config. (If you don't include them, the default will be used).

console_log:

Single value, type: one of the following options: none, basic, full. Default: `none`

Log level for the console log for this platform.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see more debug output.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this platform.

platform:

Single value, type: `string`. Defaults to empty.

Name of the platform this servo controller is connected to. The default value of `None` means the default hardware platform will be used. You only need to change this if you have multiple different hardware platforms in use and this coil is not connected to the default platform.

See the *Mixing-and-Matching hardware platforms* guide for details.

servo_max:

Single value, type: `integer`. Default: `600`

The controller is driving the servo using PWM. `servo_max` defines thes the upper PWM limit. It will use a duty cycle of `value/4096` at 50Hz.

servo_min:

Single value, type: `integer`. Default: `150`

The controller is driving the servo using PWM. `servo_min` defines thes the lower PWM limit. It will use a duty cycle of `value/4096` at 50Hz.

Related How To guides

- *I2C Servo Controllers*

servos:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `servos:` section of your config is where you specify any servo devices in your machine, as well as configuring their range of motion and a mapping of events that will cause the servos to move to certain positions.

Here's an example `servos:` section, with two servos defined called *servo1* and *servo2*:

```
servos:
  servo1:
    servo_min: 0.1
    servo_max: 0.9
    positions:
      0.0: servo1_down
      0.8: servo1_up
    reset_position: 0.5
    reset_events: reset_servo1
    number: 1
  servo2:
    positions:
      0.2: servo2_left
      1.0: servo2_home
    reset_position: 1.0
    reset_events: reset_servo2
    number: 2
```

Then for each servo in your `servos:` section, the following settings apply:

Required settings

The following sections are required in the `servos:` section of your config:

number:

Single value, type: `string`. Defaults to empty.

This is the number of the servo which specifies which driver output the servo is physically connected to. The exact format used here will depend on which control system you're using and how the servo is connected.

See the *How to configure "number:" settings* guide for details.

Optional settings

The following sections are optional in the `servos:` section of your config. (If you don't include them, the default will be used).

acceleration_limit:

Single value, type: `number` (will be converted to floating point). Default: `-1.0`

Acceleration limit for your servo. The unit of this value depends on your platform.

ball_search_max:

Single value, type: `number` (will be converted to floating point). Default: `1.0`

The value of the second position that this servo will go to in ball search.

ball_search_min:

Single value, type: `number` (will be converted to floating point). Default: `0.0`

The value of the initial position that this servo will go to in ball search.

First position in ball search

ball_search_wait:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `5s`

How long this servo will pause in each position (min and max) before moving to the other position while ball search is active.

include_in_ball_search:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Controls whether this servo is included in ball search.

platform:

Single value, type: `string`. Defaults to empty.

Name of the platform this servo is connected to. The default value of `None` means the default hardware platform will be used. You only need to change this if you have multiple different hardware platforms in use and this coil is not connected to the default platform.

See the *Mixing-and-Matching hardware platforms* guide for details.

platform_settings:

Single value, type: dict. Defaults to empty.

Todo: *Help us to write it*

positions:

One or more sub-entries. Each in the format of `number` (will be converted to floating point) : `string`

This is a sub-section mapping of servo positions to MPF event names. For example:

```
positions:
  0.1: servo1_down
  0.9: servo1_up
  0.45: servo1_mid
```

In MPF, servo ranges of motion are represented as numbers between 0.0 and 1.0. So 0.0 puts the servo at the extreme end of its range on one side as set by the servo_min: discussed below, and 1.0 moves it to the end of motion on the other side as set by the servo_max: as set below. You can use positions in between with as much precision as your servo controller will allow. (For example, a value of .4444 will tell the servo to move to 44.44% of the way between its minimum and maximum position.

The values in this `positions:` list represent MPF events that, when posted, tell this servo to move to a certain position. So in the example above, when the *servo1_up* event is posted, this servo will move to position 0.9 (90% of the way between its min and max).

You can add as many events here as you want, and the same event can be used for multiple servos.

reset_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `machine_reset_phase_3, ball_starting, ball_will_end, service_mode_entered`

Default: `None`

Events in this list, when posted,

Default: `machine_reset_phase_3, ball_starting, ball_will_end, service_mode_entered`

A list of events, or a list of events with delays, that cause the servo to move to its reset position (discussed below).

Note that by default, *ball_starting* is a reset event, so if you don't want the servo to reset on the start of each ball, you can override that like this:

```
reset_events: machine_reset_phase_3, ball_will_end, service_mode_entered
```

reset_position:

Single value, type: `number` (will be converted to floating point). Default: `0.5`

The position the servo will move to when its reset.

servo_max:

Single value, type: `number` (will be converted to floating point). Default: `1.0`

A numerical value that's sent to the servo which represents the servo's max position in relation to the servo_max: set in the controllers configuration. The actual value for this is normalized to 0.0 to 1.0 here. The controllers will convert it for the corresponding hardware.

Note that the position settings earlier are always 0.0 to 1.0, and the max (and min, discussed below) are used to calculate what actual values are sent to the servo.

So if you have `servo_max: 1.0` and `servo_min: 0.5`, and then you set the servo position to 0.5, the actual value sent will be 0.75. That position will be converted to an actual position in the hardware controller.

servo_min:

Single value, type: `number` (will be converted to floating point). Default: `0.0`

Like `servo_max:` above, except the minimum lower-end setting for values that are sent to the servo controller.

speed_limit:

Single value, type: `number` (will be converted to floating point). Default: `-1.0`

The maximum speed of this servo. The unit of this value depends on your platform.

stop_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Todo: *Help us to write it*

stop_timeout_after_last_move:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Defaults to empty.

Todo: *Help us to write it*

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Enables more detailed debug information to be added to the log (when verbose logging is enabled).

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

A friendly name for this servo that will be used in reports and the service menu.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Tags work like tags for any device. Nothing special here.

Related How To guides

- *Servo Platforms in MPF*
- *Servos*

settings:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `settings:` section of your config is where you configure settings which are configurable in *service mode*.

This is an example:

```
settings:
  replay_score:
    label: Replay Score
    values:
      500000: "500000 (default)"
      1000000: "1000000"
      1500000: "1500000"
    default: 500000
    key_type: int
    sort: 100
```

Required settings

The following sections are required in the `settings:` section of your config:

default:

Single value, type: `string`. Defaults to empty.

Default value to use if not changed or on reset. Must be included in `values`.

label:

Single value, type: `string`. Defaults to empty.

Label to use in service mode for this setting.

sort:

Single value, type: `integer`. Defaults to empty.

Sort in service mode.

values:

One or more sub-entries. Each in the format of `string` : `string`

Values for this setting in the format `value: label`. `value` will be assigned to the `machine_var` and `label` will be shown in service mode.

Optional settings

The following sections are optional in the `settings:` section of your config. (If you don't include them, the default will be used).

key_type:

Single value, type: one of the following options: str, float, int. Default: `str`

Type of the key. If you want to do math with the variable you need either `float` or `int`.

machine_var:

Single value, type: `string`. Defaults to empty.

Name of the machine variable to use. If this is not set it will use the name of this setting as machine variable.

settingType:

Single value, type: `string`. Default: `standard`

Todo: *Help us to write it*

Related How To guides

- *Service Mode*
- *Service Mode*

sequences:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

See also *sequences*.

The structure of sequence logic blocks is like this:

```
sequences:
   the_name_of_this_logic_block:
      <settings>
   some_other_logic_block:
      <settings>
   a_third_logic_block:
      <settings>
```

Note that the actual name of the logic block doesn't really matter. Mainly they're just used in the logs.

Required settings

The following sections are required in the `sequences:` section of your config:

events:

List of one (or more) events. The device will add handlers for those events. Defaults to empty.

The events section of a sequence logic block is where you define the events this logic block will watch for in order to make progress towards completion.

The real power of logic blocks is that you can enter more than one event for each step, and *only one* of the of the events of that step has to happen for that step to be complete.

Another way to look at it is that there's an *AND THEN* between all the steps. For the Sequence to complete, you need Step 1 *AND THEN* Step 2 *AND THEN* Step 3. But since you can enter more than one event for each step, you could think of those like *OR*s. So you have Step 1 (event1 *OR* event2) *AND THEN* Step 2 (event3) *AND THEN* Step 3 (event4 *OR* event5), like this:

```
##! mode: mode1
sequences:
  my_sequence:
    events:
      - event1, event2
      - event3
      - event4, event5
```

It might seem kind of confusing at first, but you can build this up bit-by-bit and figure them out as you go along.

You can enter anything you want for your events, whether it's one of MPF's built-in events or a made-up event that another logic block posts when it completes. (This is how you chain multiple logic blocks together to form complex logic.)

For example:

```
##! mode: mode1
sequences:
  logic_block_1:
    events:
      - event1
      - event2
      - event3
      - event4
      - event5
    events_when_complete: logic_block_1_done
  logic_block_2:
    events:
      - event1, event2, event3
      - event4
      - event5
    events_when_complete: logic_block_2_done
```

In the example above, there are two logic blocks. The first one just has five steps that need to complete (in 1-2-3-4-5 exact order since we're dealing with sequence logic blocks), and each step only has one event that will mark is as complete.

In the second example, if event 1, 2, or 3 is posted, that will count for step 1, and then both events 4 and 5 need to be posted for steps 2 and 3. (Again, in order, so event 1, 2, or 3 has to be posted before the logic block will even start looking for event 4.)

So in the second one, you could get event2, event4, then event5 posted, for example, and that will lead to *logic_block_2_done* being posted.

Note that you can have two logic blocks with the same events at the same time, and MPF will track the state of each logic block separately.

Optional settings

The following sections are optional in the `sequences:` section of your config. (If you don't include them, the default will be used).

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Currently unused.

Optional settings

The following sections are optional in the `logic_blocks_common:` section of your config. (If you don't include them, the default will be used).

disable_events:

List of one (or more) device control events (*Instructions for entering device control events*).

Event(s) that will disable this logic block.

A logic block must be enabled to track hits, progress, and to post events.

disable_on_complete:

Single value, type: `boolean` (`true`/`false`). Default: `true`

True/False (or Yes/No) which controls whether this logic block disables itself once it completes. This does not reset the current value.

enable_events:

List of one (or more) device control events (*Instructions for entering device control events*).

Event(s) that will enable this logic block.

A logic block must be enabled to track hits, progress, and to post events.

If you don't have any enable_events listed, then the logic block will automatically be enabled when the player's ball starts.

events_when_complete:

List of one (or more) events.

Events that will be posted when this device is completed.

events_when_hit:

List of one (or more) events.

Events that will be posted when this device is hit or advanced.

persist_state:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Boolean setting (yes/no or true/false) which controls whether this logic block remembers where it was from ball-to-ball. If `False`, then this logic block will reset itself whenever a new ball starts. If `True`, then this logic block will be saved to the player variable *<logic_block_name>_state*.

Note that logic block state is reset on mode end when this is `False` and, as normal modes stop at the end of a ball, the state is always maintained on a per-player basis, regardless of what this setting is configured for.

reset_events:

List of one (or more) device control events (*Instructions for entering device control events*).

Event(s) that will reset this logic block back to its original value. This has no effect on the enabled/disabled state of the block.

Note that there are also `reset_on_complete:` and `persist_state:` settings which also affect how and when the logic block is reset.

You can reset a logic block regardless of whether it's enabled.

reset_on_complete:

Single value, type: `boolean` (`true`/`false`). Default: `true`

True/False (or Yes/No) which controls whether this logic block resets itself once it completes. This just resets the current value or progress. It does not change the enabled or disabled state.

Note, `disable_on_complete` default is `true`, which may seem like reset isn't working. For something like a counter that automatically starts again change `disable_on_complete` to `false`.

restart_events:

List of one (or more) device control events (*Instructions for entering device control events*).

List of one (or more) events which, when posted, will restart this logic block. A restart is a reset, then an enable, combined into a single action.

start_enabled:

Single value, type: `boolean` (`true`/`false`).

If `true` this device will start enabled. If `false` this device will start disabled. If you omit this the device will start enabled unless you specify `enable_events` in which case the device will start disabled.

Related How To guides

- *Sequence Logic Blocks*
- *Integrating Logic_Blocks and Shows*

sequence_shots:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `sequence_shots:` section of your config is where you configure switch or event sequences which should trigger an event.

A *sequence_shots* is a device with multiple switches to hit, in order, for the sequence_shots to be registered as being hit/completed. You can optionally specify a time limit for these switches (i.e. the sequence must be completed within the time limit) with the `sequence_timeout:` setting.

When the first switch in a sequence is activated, the sequence_shot will start watching for the next one. When that one is activated, it looks for the next, and so on. Once the last switch is activated, the shot is considered "hit" and the device posts `your_sequence_shot_hit` (if your shot is called `your_sequence_shot`).

```
sequence_shots:
  left_orbit:
    switch_sequence: left_rollover, top_right_opto
    sequence_timeout: 3s
  weak_right_orbit:
    switch_sequence: top_right_opto, top_center_rollover
    sequence_timeout: 3s
```

Notice in the example above that there are two different shots with the same switches, but the order of the switches is inverted between the two. This is because the *left orbit* and *right orbit* shots in this machine use the same two switches, but the order the switches are activated in dictates which shot was just made.

Shots in MPF are able to track multiple simultaneous sequences in situations which is nice when multiple balls are on the playfield. If the first switch in a sequence is hit twice before the sequence completes, MPF will start tracking two sequences. Then when the next switch is it, it will only advance one sequence. If the next switch is hit again, it will advance the other sequence. But if the next switch is never hit a second time, then the second shot will not complete.

Here is an example with events:

```
sequence_shots:
  my_event_based_sequence_shot:
    event_sequence:
      - event1
      - event2
      - event3
    cancel_events: cancel
    delay_event_list:
      delay1: 1s
    sequence_timeout: 3s
```

And one with switches:

```
sequence_shots:
  my_switch_based_sequence_shot:
    switch_sequence:
      - seq2_1
      - seq2_2
```

(continues on next page)

(continued from previous page)

```
    - seq2_3
  cancel_switches: seq2_cancel
  delay_switch_list:
    seq2_delay: 1s
  sequence_timeout: 3s
```

Optional settings

The following sections are optional in the `sequence_shots:` section of your config. (If you don't include them, the default will be used).

cancel_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Those events will cancel the current sequence. Same as `cancel_switches` but with events.

cancel_switches:

List of one (or more) values, each is a type: string name of a *switches* device. Defaults to empty.

A switch (or list of switches) that will cause any in-progress switch sequence tracking to be canceled. (Think of it like a cancel "abort" switch.) If you enter more than one switch here, any of them being hit will cause the sequence tracking to reset. If MPF is currently tracking multiple in-process sequences, a cancel_switch hit will cancel all of them.

delay_event_list:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Events which will temporarily prevent new sequences from starting. Same as `delay_switch_list` but with events.

delay_switch_list:

One or more sub-entries. Each in the format of string name of a *switches* device : `time string (ms)` (*Instructions for entering time strings*)

Switches which will temporarily prevent new sequences from starting. This lets you specify a switch along with a time value that will prevent this shot from tracking from being hit. In other words, the shot only counts if the delay_switch was *not* hit within the time specified.

event_sequence:

List of one (or more) events. The device will add handlers for those events. Defaults to empty.

A sequence of events which will complete the sequence.

playfield:

Single value, type: string name of a *playfields* device. Default: `playfield`

The playfield this sequence is on.

sequence_timeout:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `0`

Timeout starting when the sequence starts (e.g. after the first switch was hit). This is the time limit the switches in the `switch_sequence:` section have to be activated in, from start to finish, in order for the sequence to be hit/completed. You can enter values with "s" or "ms" after the number, like *200ms* or *3s*. If you just enter a number then the system assumes you mean seconds. If you do not enter a time, or you enter a value of 0, then there is no timeout (i.e. the player could literally take multiple minutes between switch activations and the shot would count.)

switch_sequence:

List of one (or more) values, each is a type: string name of a *switches* device. Defaults to empty.

A sequence of switches which will complete the sequence.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Not used.

Related How To guides

- *Sequence Shots*
- *Loops / Orbits / Ramps*

shot_control_events:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

Shot can contain `control_events:` which can move the shot to a specific state.

Required settings

The following sections are required in the `shot_control_events:` section of your config:

events:

One or more values, type: `string`. Use commas to separate multiple values.

A list of one or more events, which triggers the move of the shot to a specified state.

state:

Single value, type: `int`.

The integer that is provided for the state the shot should move to when the event is posted. States are indexed at `0`, which means the first state is `0`, second state is `1`, etc.

Optional settings

The following sections are optional in the `shot_control_events:` section of your config. (If you don't include them, the default will be used).

force:

Single value, type: `boolean` (true/false). Default: `true`

If set to true, the state of shot will be modified even if the shot is disabled. If set to false, this will be ignored unless the shot is currently enabled when the event is posted.

force_show:

Single value, type: `boolean` (true/false). Default: `false`

IF set to true, the show associated with this shot state will be played, even if the shot is already in that state.

Related How To Guides

- *shots:*

shot_groups:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**YES**

You can group shots together via the `shot_groups:` section of your config file.

For example:

```
##! mode: mode1
shot_groups:
  upper_lanes:
    shots: lane_l, lane_a, lane_n, lane_e
    rotate_left_events: sw_left_flipper
    rotate_right_events: sw_right_flipper
    reset_events: upper_lanes_default_lit_complete
    enable_events: ball_started
    disable_events: ball_ending
```

Creating a shot group has several advantages, including:

- You can add "rotation" events which shift the states of all the shots in the group to the left or right, like with flipper-controlled lane change or situations where the slingshots shift which lanes are lit.
- Any time the state of a member shot in a group changes, MPF will check to see what all the other shots' states are. If they are all the same, it will post a "complete" event (in the form of *<shot_group_name>_<active_profile_name>_<profile_state_name>_complete*) which you can use to trigger scores based on complete, light shows, shot group resets, etc.
- Any time a member shot is hit, MPF will post an event (in the form of *<shot_group_name>_<profile_name_of_shot_that_was_hit>_<profile_state_name_of_shot_that_was_hit>_hit*).

You can use this to tie scoring, sounds, or logic blocks to any shot being hit in a group, which can be easier than creating entries for each individual shot.

- Any time a member shot is hit, MPF will post an event (in the form of *<shot_group_name>_<profile_name_of_shot_that_was_hit>_hit*)
- Any time a member shot is hit, MPF will post an event (in the form of *<shot_group_name>_hit*)

At first all these events might seem confusing, but really they all exist to give you the most flexibility when looking to trigger different things based on shots that are part of a shot group being hit. For example, if a shot called *left_lane* is a member of a shot group called *lanes* with a profile called *skill* and a profile state *lit* is hit, the following six(!) events will be posted:

- lanes_skill_lit_hit
- lanes_skill_hit
- lanes_hit
- left_lane_skill_lit_hit
- left_lane_skill_hit
- left_lane_hit

This lets you dial-in on the amount of precision you need when you're tying game logic to shots and shot groups.

Optional settings

The following sections are optional in the `shot_groups:` section of your config. (If you don't include them, the default will be used).

disable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

A list of one or more events that will disable all the shots in this shot group. This can be a simple list of events or a time-delayed list. If you do not specify any disable_events, then MPF will automatically create *disable_events* based on the list in the *config_validator: shot_groups: disable_events:* section of your machine-wide config. (By default that's *ball_ended*.)

disable_rotation_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

A list of one or more events that will disable rotation, meaning the states of the shots in this group will not be rotated if one of the *rotate_left_events*, *rotate_right_events*, or *rotate_events* is posted. This can be a simple list of events or a time-delayed list.

enable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

A list of one or more events that will enable all of the individual shots in this shot group. (The shot group itself has no enabled/disabled state except for rotation.) This can be a simple list of events or a time-delayed list. If a shot in the group is not enabled, then it will not post hit events but it *will* still rotate its profile state when the shot group rotates.

The presence or absence of this value will not affect whether individual shots in the group can be enabled via their own *enable_events* settings. An individual shot can always be enabled/disabled regardless of the group state, although a subsequent group enable/disable events will also affect that individual shot.

enable_rotation_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

A list of one or more events that will allow the states of the shots in this group to be rotated (based on the *rotate_left_events*, *rotate_right_events*, or *rotate_events* as described above). This can be a simple list of events or a time-delayed list. If rotation is not enabled, rotation events being posted will have no effect. (Rotation is enabled by default.)

reset_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

A list of one or more events that will reset all the shots in this shot group. This can be a simple list of events or a time-delayed list. Resetting a shot group means that every shot in the group jumps back to the first state in whatever shot profile is active at that time.

restart_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

A list of one or more events that will restart all the shots in this shot group. A restart is the same as calling reset and enable, so restarting a shot group will jump every shot in the group to the first state of that shot's profile and immediately enable all the shots.

rotate_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Same as `rotate_right_events:`.

rotate_left_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

This list of events that, when posted, will rotate the current state of each shot to the shot to its left. The state of left-most (i.e. first entry) in your shots: list will rotate over to the right-most shot. These states are based on whatever shot profile is active at that time.

rotate_right_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

This list of events that, when posted, will rotate the current lit and unlit shot states to the right. This can be a simple list of events or a time-delayed list. The state of right-most (i.e. last entry) in your *shots:* list will rotate over to the left-most shot.

shots:

List of one (or more) values, each is a type: string name of a *shots* device. Defaults to empty.

The list of shots (from the `shots:` section of your config file) that make up this shot group. Order is important here if you want to implement shot rotation events. Individual shots can belong to more than group at the same time, which is useful in a lot of different situations. For example, you might have three banks of three standup targets each, and you can create shot groups for each bank with events that will be triggered when the individual bank is complete, and then you can create a fourth shot group with all nine targets in it which could post different events when all nine targets have been hit.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to *true* to add lots of logging information about this shot to the debug log. This is helpful when you're trying to troubleshoot problems with this shot. Default is *False*.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

The plain-English name for this device that will show up in operator menus and trouble reports.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

Related How To guides

- *Grouping Shots for lane change, rotation, etc.*
- *Skill Shot*
- *Sequence Shots*

shot_profiles:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `shot_profiles:` section of your config is where you configure the settings for various *shot profiles* that you can then apply to your shots.

Here's an example:

```
##! mode: mode1
shot_profiles:
  my_default_profile:
    states:
      - name: unlit
        show: "off"
      - name: lit
        show: "on"
```

<name>:

This is the name of the shot profile, which is how you'll refer to it elsewhere in your config files when you apply it to shots. The sample shot_profiles: section of the config file above contains a profile named "default" (which is actually included in the system-wide `mpfconfig.yaml` file).

advance_on_hit:

Single value, type: `boolean` (Yes/No or True/False). Default: `True`

This setting controls whether the active shot profile advances to its next state when the shot is hit. The default is true, but you can set this to false if you want to manually advance the shot some other way. (If this is false, you can still advance the shot with *advance_events*, for example.)

block:

Single value, type: `boolean` (Yes/No or True/False). Default: `false`

Lets you control whether hits to this shot are propagated down to lower priority modes. The default value is true if you don't specify this, meaning that blocking is enabled

If you have `block: true` in a shot profile, then hits to that shot when that profile is applied only are registered in the highest mode where that shot is enabled. If you set `block: false`, then when a shot is hit in one mode it will also look down to lower priority modes where that shot is enabled. If that lower priority mode has a different profile applied then it will also register a hit event based on that profile. This will continue until it reaches a level with `block: true` or until it reaches the end of the mode list.

This is better explained with an example.

Imagine you have four lanes at the top of your machine which you use in your base mode in a normal lane-change fashion. (Lanes are unlit by default, hit a lane and they light, complete all four lanes for an award.) Now imagine you also use those lanes for a skillshot where one of the lanes is flashing and you try to hit it while the skillshot is enabled. In this case, you'd have different shot profiles for each mode, perhaps the default profile in your base mode (with unlit->lit states) and a skillshot profile in your skill shot mode (with flashing->complete states).

By default, if the player hits the a lane when the skill shot mode is running, the skillshot profile is the active profile so it's the shot that gets the hit. But then when the skill shot mode ends, the lane the player just hit is not lit, since that shot profile was not active when it was hit. (In other words, the skillshot blocked the hit event.) So if you add `block: false` to your skillshot shot profile, then when the shot is hit when the skill shot mode is running, it will receive the hit and advance the shot from flashing to complete. Then the lower base mode will also get the shot, and it will advance its state from unlit to lit. The lights for the shot will only reflect the skillshot lights since it's the higher priority, however, you will get *yourshot_skillshot_flashing_hit* and *yourshot_default_unlit_hit* events since both the hits registered because you set the skillshot profile not to block the hit.

loop:

Single value, type: `boolean` (Yes/No or True/False). Default: `False`

Controls whether the states of this profile "loop" when they reach the end. If true, then the shot being hit when the profile is in the last state causes the profile to "loop" around back to the first state. This is useful if you want to create a "toggle" shot where you could create a profile with two steps (lit and unlit) and then set loop to be true. (If you have more than two steps in the shot profile, then the looping will go from the last one back to the first one.) The default is false, meaning when the profile reaches its last state, it will just stay there even if it's hit again.

player_variable:

Single value, type: `string`. Default: `None`

This is a profile setting that lets you specify the name of the player variable that will be used to track the status of this shot when this profile is applied. If you don't specify the name of a player variable, it will automatically use *<shot_name>_<profile_name>* as the player variable.

rotation_pattern:

List of one (or more) values, each is a type: `string`. Default: `R`

This setting lets you specify a custom rotation pattern that's used when an event from this profile's rotation_events: section is posted. You enter it as a list of Ls and Rs, for example:

```
##! mode: mode1
shot_profiles:
  my_default_profile:
    states:
      - name: unlit
        show: "off"
      - name: red
        show: led_color
        show_tokens:
          color: red
      - name: blue
        show: "flash"
        show_tokens:
          color: blue
    rotation_pattern: L, L, L, L, R, R, R, R
```

In the above example, the first four times a rotation_event is posted, this shot group will rotate to the left, then the next four to the right, then the next four to the left, etc. The pattern will loop. This is how you could specify a single lit target that "sweeps" back and forth across a group of five targets, for example. This only impacts *rotation_events*, not *rotate_left_events* and *rotate_right_events* since those events imply a direction.

show:

Single value, type: `string`. Default: `None`

The name of the show associated with this shot profile. Note that you can specify a single show which applies to the entire shot profile (here), or you can specify a different show for each step/state (in the `states:` section, covered below.

If you specify a show here, then the show will not auto play, and instead will advance to the next step with each step/state advancement of the shot. This is useful for simple things like turning a light on or off. For more complex scenarios, you can set a full show per step/state below.

show_when_disabled:

Single value, type: `boolean` (Yes/No or True/False). Default: `False`

Controls whether the lights or LEDs for shots which have this profile applied will be active when this shot is disabled. By default this is *true*, so if the shot profile associated with this shot has the light turning on, then when you disable the shot the light will stay on. Set it to *false* if you want the lights or LEDs to turn off when the shot is disabled. (Note that even when this is false, the lights or LEDs can still be controlled by other light scripts, light shows, manual commands, etc.)

state_names_to_not_rotate:

List of one (or more) values, each is a type: `string`. Default: `None`

This works like *state_names_to_rotate*, except it's the opposite where you can enter the names of states to not rotate. You don't need to use both—the options are here just for convenience.

state_names_to_rotate:

List of one (or more) values, each is a type: `string`. Default: `None`

This is a list of state names that will be used to determine which shots in a shot group will be rotated. By default, all states are included. But this can be nice if you only want to rotate a subset of the states. For example, if you have a shot group with a bunch of lights that represent modes, you might have a shot profile with states called *unlit*, *active* (flashing), and *complete* (lit). You'd use these shots (and their lights) to track the game modes you've completed, so at any time, you'd have a bunch of unlit shots representing modes you haven't completed yet, solidly lit shots for modes you've completed, and a single flashing shot representing the mode that will be started next. Then in your game if you wanted to rotate among the incomplete targets, you would set your shot profile so it only rotated those state names, like this

states:

The `states:` section contains the following nested sub-settings

Under each shot profile name, a setting called *states:* lets you specify various properties for the target in different states. You can configure multiple states in the order that you want them to be stepped through. (You use a dash, then a space, then a setting to indicate that items should be a list. The following sections explain the settings for each state:

name:

Single value, type: `string`.

This is the name of the step. In other words, it's what "state" the shot is in when this profile step is active.

loops:

Single value, type: `integer`. Default: `-1`

Loops setting from the show player, controls how many times the show loops (`-1` is unlimited).

manual_advance:

Single value, type: `boolean` (Yes/No or True/False). Default: `False`

If True, the show does not automatically advance to the next step.

priority:

Single value, type: `integer`. Default: `0`

The priority shift of the show that's played.

show:

Single value, type: `string`. Default: `None`

The name of the show that will be played when a shot with this profile applied is in this step (or state).

show_tokens:

One or more sub-entries, each in the format of type: `str`:`str`. Default: `None`

Show tokens for the show.

speed:

Single value, type: `number` (will be converted to floating point). Default: `1`

Playback speed of the show.

start_step:

Single value, type: `integer`. Default: `1`

The step number the show will start on.

sync_ms:

Single value, type: `integer`. Default: `None`

The sync_ms value of the show.

shots:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**YES**

The *shots:* section of your config file is where you define the shots in your machine. A *shot* is a switch, a series of switches that have to be hit in order, or an event or series of events.

Shots are used for things like standup targets, rollover lanes, drop targets, ramps, loops, orbits, etc.

Each shot can have a *shot profile* applied to it which defines what happens when its hit. For example the shot profile might specify that the shot starts unlit, then when it's hit it becomes complete. Or a shot profile might specify that it's flashing slowly, and each hit makes it flash faster and faster until it's been hit enough times, etc.

You can specify different shot profile on a per-mode basis, meaning a shot can have one behavior in the base mode and then take on another behavior when a higher-priority mode is started. The tracking of various states of the shot profiles is maintained on a per-mode basis.

You can group multiple shots together into *shot groups* for group-level functionality like posting events when all the shots in a group in the same state (lit, unlit, complete, etc.) and for rotating the states of shots to the left or right based on certain events happening (slingshot hits, flipper button pushes, etc.). A shot can be a member of multiple groups at the same time.

Here's a sample *shots:* section from a config file:

```
##! mode: mode1
shots:
  lane_l:
    switch: lane_l
    show_tokens:
      light: lane_l
  lane_a:
    switch: lane_a
    show_tokens:
      light: lane_a
  lane_n:
    switch: lane_n
    show_tokens:
      light: lane_n
  lane_e:
    switch: lane_e
    show_tokens:
      light: lane_e
  upper_standup:
    switch: upper_standup
    show_tokens:
      leds: led_17, led_19
```

Create one entry in your `shots:` section for each shot in your machine. Don't worry about grouping shots here. (That's done in the `shot_groups:` section.) The shot name can be whatever you want, and it will be the name for this shot which is used throughout your machine. Remember that everything with at least one switch and a "state" is a shot, so standups, rollovers, inlane/outlines, ramps, loops… You will have lots of shots in your game.

Each shot in your `shots:` section can have the following config options set:

Optional settings

The following sections are optional in the `shots:` section of your config. (If you don't include them, the default will be used).

advance_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Events in this list, when posted, cause this shot to be advanced to its next state in the active shot profile. If the shot is on the last state, then it will roll over if the shot profile is configured to loop, otherwise it will do nothing. *Advance_events* are similar to *hit_events*, except *advance_events* are more "stealthy" in that they only advance the state (and update the lights or LEDs). They do not post hit events and therefore do not trigger scoring or other events related to a shot hit. They are useful if you need to move a shot to a starting state (like selecting a shot to be active for skill shot).

control_events:

List of one (or more) values, each is a type: *shot_control_events*. Defaults to empty.

Control events to change the state of this shot. This supports jumping to state to a value that is specified. This is a `0` index, so the first state is `0`.

For instance in the following example `set_state_one` will set the state to `1`, which is the second state:

```
shots:
  shot_with_control_events:
    control_events:
      - events: set_state_one
        state: 1
```

delay_switch:

One or more sub-entries. Each in the format of string name of a *switches* device : `time string (ms)` (*Instructions for entering time strings*)

A dictionary of switches and times which prevent hits for a certain time. You can use this if you got another lane feeding into your shot and you want to prevent it from hitting this shot. Use this with care as it might cause issues during multiball.

This is an example:

```
##! mode: mode1
shots:
  my_shot:
    switch: s_my_shot
    delay_switch:
      s_other_lane: 2s
```

In this example an activation of `s_other_lane` will prevent the shot from being hit for two seconds.

disable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Events in this list, when posted, disable this shot. If a shot is disabled, then hits to it have no effect. (e.g. The shot will remain in whatever state it's in.)

enable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Events in this list, when posted, enable this shot. If a shot is not enabled, then hits to it have no effect. (e.g. The shot will remain in whatever state it's in.)

hit_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Events in this list, when posted, cause this shot to be "hit". This is effectively the same thing as if the ball activated the switch associated with this shot, (or that the entire switch sequence has been completed), except it comes in via an event instead of from a switch activity.

mark_playfield_active:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Todo: *Help us to write it*

---

persist_enable:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Whether this shot should persist its enable state in a player variable. If set to `True` this will also persist the state into the next ball of the same player.

playfield:

Single value, type: string name of a *playfields* device. Default: `playfield`

On which playfield is this shot? This is only relevant when you have multiple playfields. It is used mostly for ball search.

profile:

Single value, type: string name of a *shot_profiles* device. Default: `default`

The name of the *shot profile* that will be applied to this shot.

- If you're editing a machine-wide config file , then the profile name specified here will be the default profile for that shot any time a mode-specific config doesn't override it. (If you don't specify a profile name, MPF will assign the shot profile called "default".)
- If you're in a mode configuration file , then this profile entry is the name of the shot profile that will be applied only when this mode is active. (i.e. it's applied when the mode starts and it's removed when the mode ends.) Like other mode-specific settings, shot profiles take on the priorities of the modes they're in, so if you have a profile from a mode at priority 200 and another from priority 300, the profile from the priority 300 mode will be applied. If that mode stops, then the shot will get the profile from the priority 200 mode.

Shots can have (and track) multiple profiles at the same time (up to one profile per mode). Only the show from the highest-priority profile will play though.

reset_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Events in this list, when posted, reset this shot. Resetting a shot means that it jumps back to the first state in whatever *shot profile* is active at that time.

restart_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Events in this list, when posted, restart this shot. Restarting a shot is equivalent to resetting and then enabling the shot, done with a single event.

show_tokens:

One or more sub-entries. Each in the format of `string` : template_str

A subsection containing key-value pairs that are passed to the show that's run when this shot is in a certain state.

For example, consider the following shot config:

```
##! mode: mode1
shot_profiles:
  flash:
    states:
      - name: unlit
```

(continues on next page)

(continued from previous page)

```
        show: "off"
      - name: lit
        show: "flash"
shots:
  shot1:
    switch: switch1
    profile: flash
    show_tokens:
      leds: led1
```

The shot above has a show token called *leds* which is set to *led1*. This means that when a show associated with this shot is played, if that show contains placeholder tokens for `(leds)`, they will be dynamically replaced with the value of `led1` when that show is played by this shot.

The purpose of show tokens is so you can create resuable shows that you could apply to any shot.

For example, imagine if you wanted to create a shot to flash an LED between red and off. It might look like this:

```
# show to flash an LED
shows:
  flash_light:
    - time: 0
      lights:
        (leds): red
    - time: 1
      lights:
        (leds): off
```

Assuming the "flash" profile (as defined in the `profile: flash` in the above shot) was configured for the state that show was in, when the shot entered that state, it would replace the `(leds):` section of the show with `led1`.

More information about *show tokens*

start_enabled:

Single value, type: `boolean` (`true`/`false`). Defaults to empty.

Whether the shot starts as enabled (if you set this to `True`) or as disabled (if you set this to `False`). If you do not set this, MPF will check if there are `enable_events`. The shot will start disabled in that case or enabled otherwise.

switch:

List of one (or more) values, each is a type: string name of a *switches* device. Defaults to empty.

The name of the switch (or a list of switches) for this shot. You can use multiple switches if the shot happens to have multiple switches, though this is rare. (Maybe there are two standups on the sides of a ramp that you always want to be the same so you just create them as one logical shot?)

Do *not* enter multiple switches here for different shots, like for a bank of rollover lanes. In that case you would set up each shot as its own shot here and then group them via `shot_groups:`.

Also do *not* enter multiple switches if you want the shot to be complete when all the switches are hit. (That's what the `switch_sequence:` setting is for.) Entering multiple switches here is just in case you have a shot where you want any of the switches being hit to count as that shot being hit.

switches:

List of one (or more) values, each is a type: string name of a *switches* device. Defaults to empty.

This setting is the same as the `switch:` setting above. You can technically enter a single switch or a list of switches in either the `switch:` setting or the `switches:` setting, but we include both since it was confusing to be able to enter multiple switches for a singlular "switch" setting and vice versa.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to *true* to add lots of logging information about this shot to the debug log. This is helpful when you're trying to troubleshoot problems with this shot.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

The plain-English name for this device that will show up in operator menus and trouble reports.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

Related How To guides

- *Shots*

show_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

Note: This section can also be used in a show file in the `shows:` section of a step.

The `show_player:` section of your config is where you start, stop, pause, (etc.) shows.

Here is an example:

```
show_player:
  some_event: your_show_name
  some_other_event: another_show
```

In the example above, when the event *some_event* is posted, the show called `your_show_name` will be played (started). When the event *some_other_event* is posted, the show called `another_show` will be played.

See *Show player* for details.

Optional settings

The following sections are optional in the `show_player:` section of your config. (If you don't include them, the default will be used).

action:

Single value, type: one of the following options: play, stop, pause, resume, advance, step_back, update, queue. Default: `play`

`play` Starts playing the show. This is the default action which will happen if you don't include an `action:` setting.

`stop` Stops the show. Removes and "undoes" anything the show did, and posts the show stop events.

`pause` Pauses the show by holding it at the current step. Posts the show pause events.

`resume` Resumes a previously paused show.

`advance` Manually advances a show to the next step. Posts the show advance events.

`step_back` Manually moves the show back to the previous step. Posts the show step_back events.

`update` Not yet implemented. In the future it will be used to change a setting of a running show, like changing the playback speed.

block_queue:

Single value, type: `boolean` (`true`/`false`). Default: `false`

You can use `block_queue: yes` if you want the show to block a queue event until the show is done. Note that you can only use this if the event that starts the show is a *queue event*.

For example, the mode stopping events are queue events. So take a look at the following config:

```
show_player:
  mode_my_mode_stopping:
    show_1:
      block_queue: true
```

In the example above, when the mode called *my_mode* posts its stopping event, show_1 will start playing. However because this show is set to block the queue event, the mode stopping event will not finish until the show finishes. In other words, the mode will not fully stop, and the *mode_my_mode_stopped* event will not be posted until the show ends.

If you didn't use the block_queue setting, then the show would start and then stop right away since the mode would end and be over (and shows started in modes are stopped when those modes end).

If you used this setting, make sure that you don't have `loops: -1`, or a `duration: -1` as the final step of the show, since those will mean the show will never end, and then the queue event will never be unblocked, and your machine will hang.

events_when_advanced:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

Event(s) that will be posted when this show has been manually advanced to the next step.

events_when_completed:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

Event(s) that will be posted when this show has completed, meaning it ran through to the last step and ended naturally.

Note that if a show loops, these events are *not* posted when the loop happens. (You can use the *events_when_looped* for that.) However if a show is set to loop a specific number of times and then ends, these events will be posted at the end.

Note that if you want an event to post whenever the show stops, even if it didn't make it all the way to the end, you can use *events_when_stopped*.

events_when_looped:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

Event(s) that will be posted when this show has looped (meaning it reached the end and is jumping back to the first step).

events_when_paused:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

Event(s) that will be posted when this show has been paused.

events_when_played:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

Event(s) that will be posted when this show is played (started).

events_when_resumed:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

Event(s) that will be posted when this show is resumed from a pause.

events_when_stepped_back:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

Event(s) that will be posted when this show has been manually stepped back to the previous step.

events_when_stopped:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

Event(s) that will be posted when this show has been stopped. Note that these events are posted anytime the show has been stopped, regardless of whether it made it to the end and stopped on its own, or whether it was stopped randomly where it was.

events_when_updated:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

Event(s) that will be posted when this show has been updated. Note that the show "update" function has not been implemented yet, so this setting is more of a placeholder at the moment.

show_config:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

The `show_config:` section of your config is where you configure a show to play within a device.

See *show_player* for more details about the settings.

Required settings

The following sections are required in the `show_config:` section of your config:

show:

Single value, type: `string`.

The show to play.

Optional settings

The following sections are optional in the `show_config:` section of your config. (If you don't include them, the default will be used).

loops:

Single value, type: `integer`. Default: `-1`

How often should the show loop? `-1` means forever.

manual_advance:

Single value, type: `boolean` (Yes/No or True/False).

Whatever, the show should advance manually only.

priority:

Single value, type: `integer`. Default: `0`

Priority for this show. This is usually added to the mode priority if the device is defined within a mode.

show_tokens:

One or more sub-entries, each in the format of `string` : `string` Dict of show tokens to pass to the show.

speed:

Single value, type: `number` (will be converted to floating point). Default: `1`

Speed multiplier for this show.

start_step:

Single value, type: `integer`. Default: `1`

First step to play.

sync_ms:

Single value, type: `integer`.

See the *Synchronizing multiple shows* documentation for details.

key:

Single value, type: `string`. Defaults to empty.

Used to set a unique identifier you can set when playing a show which can then be used later to identify a show you want to perform an action on.

loops:

Single value, type: int_or_token. Default: `-1`

Controls the looping / repeating of the show. The default if you don't include this setting is `loops: -1` means that the show will repeat indefinitely until it's stopped.

If you just want a show to play once and then stop, use `loops: 0`.

Since this setting is the number of times it loops, the value will be one less than the number of times the show will play. (e.g. `loops: 1` means the show will loop once which means it will play through twice.)

Note that if a show only has one step, *loops* will be set to 0, regardless of the actual loops setting.

manual_advance:

Single value, type: `boolean` (`true`/`false`). Default: `false`

If you set this to yes/true, then the show will not auto-advance based on time. Instead you will have to manually advance the show step-by-step with additional show_player entries with `action: advance` entries.

This can be useful if you want to have some kind of slow progress based on a series of events instead of a show that auto plays.

For example:

```
show_player:
  some_event:
    show_1:
      manual_advance: true
  some_advance_event:
    show_1:
      action: advance
```

In the example above, the event *some_event* will start show_1, but that show will stay on its first step since it's set to manually advance. Then each time the event *some_advance_event* is posted, show_1 will advance to its next step.

priority:

Single value, type: int_or_token. Default: `0`

Adjusts the priority of the show that's played.

By default, shows play at the priority of the mode where the show_player entry is. So this setting merely adjusts the show's priority up or down. For example, if you have a mode running at priority 300, and a show in a show_player with the setting `priority: 10`, then that show will run at priority 310. Priorities can also be negative.

The show's priority affects the priority of everything it does. Sounds, slides, LEDs, etc.

show:

Single value, type: `string`. Defaults to empty.

Todo: *Help us to write it*

show_queue:

Single value, type: string name of a show_queues device. Defaults to empty.

Todo: *Help us to write it*

show_tokens:

One or more sub-entries. Each in the format of `string` : template_str

Allows you to specify show token values that will be used to replace the show tokens in the show when it's played.

Read what show tokens are *here*.

For example:

```
show_player:
  some_event:
    show1:
      show_tokens:
        led: right_inlane
```

In the example above, the show called "show1" will be played, but the show token called "led" in the show will be replaced at runtime with the value "right_inlane".

speed:

Single value, type: float_or_token. Default: `1`

Controls the playback speed of the show. The default value of 1 means the show plays back at 1x speed. (In other words, it plays at the actual speed each step is configured for. In this case you don't actually need to include the setting.)

If you want to play the show at 2x the speed, use `speed: 2`. If you want to play it at half speed, use `speed: .5`. Etc.

start_running:

Single value, type: `boolean` or `template` (`true/false`; *Instructions for entering templates*). Default: `True`

Whether the show starts running immediately when it is played.

By default, calling `play` on a show begins at the starting step and advances through the steps according to the show config. If `start_running` is false, the show will play the starting step and immediately pause. You can begin playing the show by calling show_player with `action: resume`.

start_step:

Single value, type: `integer` or `template` (*Instructions for entering templates*). Default: `1`

Which step the show starts on when it's played.

Note that you can use a *dynamic value* for this setting.

sync_ms:

Single value, type: int_or_token. Defaults to empty.

Sets the sync_ms value of this show which will delay the start to a certain millisecond multiple to ensure that multiple shows started at different times all play in sync with each other.

See the *Synchronizing multiple shows* documentation for details.

Related How To guides

Todo: *Help us to write it*

show_pools:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `show_pools:` section of your config is where you configure a pool of shows. When used one of the shows is selected from the pool based on a configurable pattern called `type`.

This is an example:

```
show_pools:
  group1:
    shows:
      - show1
      - show2
      - show3
    type: random
```

Required settings

The following sections are required in the `show_pools:` section of your config:

shows:

List of one (or more) values, each is a type: string name of a *shows* device. Defaults to empty.

A list of shows which are part of the show pool

Optional settings

The following sections are optional in the `show_pools:` section of your config. (If you don't include them, the default will be used).

type:

Single value, type: one of the following options: random, sequence, random_force_next, random_force_all. Default: `sequence`

How the next show is selected. See *Assets* for details.

Related How To guides

- *Assets*
- *Shows*

shows:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `shows:` section of your config is where you define shows in your config. See *Shows in files versus shows in configs* for details. Furthermore, you can also *define shows in separate files*.

Related How To guides

- *Shows*
- *Tutorial step 16: Create an attract mode display show*

slide_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**
Valid in *shows*	**YES**

The `slide_player:` section of your config is where you configure slides to be shown (or removed) based on events being posted.

This is an example:

```
slide_player:
  event1: slide1
  event2: slide2
  event3: slide3
```

See *Slide player* for details.

Optional settings

The following sections are optional in the `slide_player:` section of your config. (If you don't include them, the default will be used).

action:

Single value, type: one of the following options: play, remove. Default: `play`

`play` Makes the slide active. Note that the actual slide shown on a display will be whichever active slide has the highest priority, so depending on what other slides are active, this action might not technically show the slide.

Also note that if a transition is specified (either in the slide definition or the `transition:` section here, then than transition will be used when showing this slide.

`remove` Removes the slide from the list of active slides. If this slide is the highest priority slide that's currently showing, then the next-highest priority slide will be shown in its place.

If a `transition_out:` setting is used, then that transition will be used here.

For example, to remove *slide1* when the event *remove_slide_1* is posted:

```
slide_player:
  remove_slide_1:          # event name
    slide1:                # slide name
      action: remove
```

You can also specify a transition for the removal, like this:

```
slide_player:
  remove_slide_1:           # event name
    slide1:                 # slide name
      action: remove
      transition: fade
```

background_color:

Single value, type: `color` (*color name*, *hex*, or list of values *0-255*). Default: `000000ff`

Todo: *Help us to write it*

expire:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Defaults to empty.

Specifies that this slide should automatically be removed after the time has passed. When it's removed, whichever slide is the next-highest priority will be shown.

The expiration timer starts immediately, so if the slide you're displaying here doesn't end up being shown because it's not the highest-priority slide, the timer is still running in the background, and the slide will still be removed when the timer expires.

If a `transition_out:` is specified, it will be applied when the slide expires:

```
slides:
  base:
    widgets:
      - type: text
        text: BASE SLIDE
        color: ff0000
        font_size: 100
  expire_slide:
    widgets:
      - type: text
        text: EXPIRE 5s
        color: purple
        y: 66%
    expire: 5s
    transition_out:
      type: wipe
      duration: 5s
slide_player:
  mc_reset_complete.1: expire_slide
  mc_reset_complete.2: base
```

force:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Forces this slide to be shown, even if it's not the highest priority. Note that if you add or remove a slide and the priority list is recalculated, whichever slide is the highest priority will be shown. This `force:` option is sort of a one-time thing. Really you should use priorities to control which slides are shown.

priority:

Single value, type: int_or_token. Defaults to empty.

An adjustment to the priority of the slide that will be shown.

In MPF, all slides have a priority. Only one slide is show on a display at a time, and the slide with the highest priority is automatically shown. If that slide is removed, the next-highest priority slide is shown.

If you have a `slide_player:` section in a mode-based config file, then slides shown will automatically have the priority of the mode. (`slide_player:` sections from your machine-wide config file use priority `0`.) However you can adjust the priority of a slide (up or down) by adding a `priority:` setting with a positive or negative value.

If a slide is being shown as part of a show, the slide will have the priority set to whatever the priority of the show is (which itself is also the priority of the mode unless you adjust it)

show:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Specifies whether this slide should be shown. (It will only be shown if it's the highest priority slide for that display.) If you set `show: false`, then the slide will be created and added to the display's collection of slides, but it won't be shown.

Note that if you add or remove a slide and the priority list is recalculated, whichever slide is the highest priority will be shown. This `show:` option is sort of a one-time thing. Really you should use priorities to control which slides are shown.

slide:

Single value, type: `string`. Defaults to empty.

You can specify the slide name here (instead of as key for the complete player). There are reasons to use this but you won't need it in most cases.

target:

Single value, type: `string`. Defaults to empty.

Specifies the display target this slide will be shown on. If you do not specify a target, then the slide will be shown on the default display.

In MPF, display targets are the names of the displays themselves. However there is also a *slide_frame* widget (literally a widget which you add to a slide which holds other slides, kind of line picture-in-picture). When you add a slide_frame to a slide, you give it a name, and that name is added to the list of valid targets.

So really the `target:` here is either the name of a display, or the name of a slide_frmae where you want this slide to be displayed.

tokens:

One or more sub-entries. Each in the format of `string` : `string`

Todo: *Help us to write it*

transition:

Unknown type. See description below.

Note that you can also configure a transition when the slide is defined in the *slides:* section of your config if you want to use the same transition every time for a slide and don't want to always have to define it here.

If you specify a transition in both places, the transition in the slide_player or show will take precedence.

transition_out:

Unknown type. See description below.

Note that you can add a transition out to the slide player when a slide is shown, and it will be "attached" to the slide and used when that slide is removed (either with the slide player or when a new slide is created with a higher priority than it).

```
slides:
  base:
    widgets:
      - type: text
        text: BASE SLIDE
        color: ff0000
        font_size: 100
  top_slide:
    widgets:
      - type: text
        text: TOP SLIDE
        color: purple
        y: 66%

slide_player:
  mc_reset_complete.1: top_slide
  mc_reset_complete.2: base
  mc_reset_complete.3:
    top_slide:
      action: remove
      transition:
        type: fade
        duration: 3s
```

Or you can specify a transition out when you remove the slide (with `action: remove`).

There can only be one transition between slides, so if an outgoing slide has a transition out set, and an incoming slide has a transition set, then the incoming transition will take precedence.

widgets:

Unknown type. See description below.

Todo: *Help us to write it*

Related How To guides

Todo: *Help us to write it*

slides:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `slides:` section of your config is where you pre-define "named" slides that you can then use later in shows and the slide_player section of a config file. See the *How to Show a Slide on a Display* guide for details on this. You can test slides and widgets interactively using *Interactive MC (iMC)*.

Slide names are universal throughout MPF, so if you create two slides with the same name—even in different modes—one of them will overwrite the other and things will be confusing, so don't do that.

See the *How to create slides* documentation for full details on how to create slides. (You should definitely "learn" about slides there. The settings here are mostly used for reference later.)

There are several different ways you can enter slides. In all cases, you'll have a `slides:` section of your config, and then under that, you'll have sub-entries which are slide names. But what is entered under each slide name varies.

Option 1: Slide with a widget

If you want to define a slide that only has a single widget, you can just add the *widget's properties* under the slide name. In the example below, we're defining two slides, one called *my_slide_1* and the other called *my_slide_2*, and they each only have a single widget.

```
slides:
  my_slide_1:
    type: text
    text: THIS IS MY SLIDE
  my_slide_2:
```

(continues on next page)

(continued from previous page)

```
    type: text
    text: THIS IS ANOTHER SLIDE
    color: lime
    font_size: 25
```

Option 2: List of widgets

Of course many slides you'll define will have more than one widget. To add multiple widgets to a slide, just enter them like you entered a single widget, but use a dash (and a space) to dictate where a new widget starts, like this:

```
slides:
  my_slide_1:
    - type: text
      text: THIS IS MY SLIDE
    - type: image
      image: johnny_5
  my_slide_2:
    - type: text
      text: THIS IS ANOTHER SLIDE
    - type: text
      y: 20%
      text: IT HAS MORE THAN 1 WIDGET
    - type: ellipse
      color: red
      width: 200
      height: 100
```

Option 3: Widgets under "widgets:" section

In addition to widgets, slides have other options (as described below), and sometimes you might want to define a slide that has widgets and slide settings. To do that, you need to move your widgets definition into a sub-section called "widgets:", and then you can add the other slide settings under the slide along with the widgets.

Here's an example. Note that the slide with multiple widgets is using the dash in the widgets: section to separate the individual widgets.

```
slides:
  my_slide_1:
    background_color: red
    widgets:
      type: text
      text: THIS IS MY SLIDE
  my_slide_2:
    widgets:
      - type: text
        text: THIS IS ANOTHER SLIDE
      - type: text
        y: 20%
        text: IT HAS MORE THAN 1 WIDGET
```

(continues on next page)

(continued from previous page)

```
    - type: ellipse
      color: red
      width: 200
      height: 100
  expire: 2s
  transition:
    type: move_in
    direction: right
```

You can mix-and-match the three options for entering widgets as needed within the same slides: section of your config.

Creating a blank slide

If you want to create a blank slide (perhaps an empty canvas that you'll populate via the widget player later?), then you need to tell the slides: section that you have an empty list. In YAML, that's done with a [and] next to each other (which is confusing because it looks like a rectangle, but it's not, like this: `[]`.

You can use this format to create a blank slide with no options:

```
slides:
  my_blank_slide: []
```

Or you can use it to create a blank slide with options, but no widgets, like this:

```
slides:
  my_blank_slide:
    background_color: red
    widgets: []
```

Settings

The following sections provide additional options for your slide which you can use if you move the widgets into their own `widgets:` section. If you just include the widgets as top-level entries (like Options 1 and 2 above), then the default values for each of these settings below will be used.

background_color:

Single value, type: `color` (*color name*, *hex*, or list of values *0*-*255*). Default: `000000ff`

The background color of the slide. Details on how to enter color values are *here*.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set to true/yes if you want to add addition debug information about this slide to the log. (Note this requires a verbose log to see.)

expire:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Defaults to empty.

Sets an expiration time which will automatically remove this slide. If it's showing when it's removed, the next-highest priority active slide will be shown in its place.

Note that you can also configure expiration when the slide is shown (in either a show or via the slide_player), so you don't need to define an expire setting as part of the slide definition unless you want that expire time to be used every time the slide is shown.

If you specify an expire time in both places, the expire time in the slide_player or show will take precedence.

opacity:

Single value, type: `number` (will be converted to floating point). Default: `1.0`

Sets the overall opacity of the slide. A value of 1.0 is fully opaque. A value of .5 means the slide is 50% transparent, and a value of 0 means the slide will be invisible and you'll probably be confused about why it's not showing up.

transition_out:

Unknown type. See description below.

Todo: *Help us to write it*

Note that you can also configure a transition when the slide is shown (in either a show or via the slide_player), so you don't need to define a transition as part of the slide definition unless you want that transition to be used every time the slide is shown.

If you specify a transition in both places, the transition in the slide_player or show will take precedence.

Related How To guides

Todo: *Help us to write it*

spinners:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

Spinner devices provide accruals for switches that are hit in rapid succession, and post events based on timeouts after switch hits.

```
spinners:
  basic_spinner:
    switch: s_orbit_spinner
    active_ms: 500
  dual_spinner:
    switches: s_top_loop_left, s_top_loop_right
    labels: left, right
    active_ms: 1200
    idle_ms: 2400
```

A spinner becomes "active" when a `switch:` or `switches:` is hit, and remains active as long as switch hits continue. The time specified by `active_ms:` determines how long the spinner will wait after the last hit before it is no longer active.

If an `idle_ms:` time is specified, the spinner will move from "active" to "inactive" for that duration, before finally settling on "idle". If a switch is hit while idle, the spinner will become active again. If no `idle_ms:` time is specified, the spinner will go directly from active to idle.

The basic flow:

1. Spinner sits in idle state
2. **A spinner switch is hit**
   i. The spinner becomes "active" and sets a timeout for `active_ms:` duration
   ii. The spinner posts *spinner_<name>_active* event
   iii. The spinner posts *spinner_<name>_hit* event
3. **Additional switch hits occur**
   i. The spinner resets the timeout for another `active_ms:` duration
   ii. The spinner posts a *spinner_<name>_hit* event for each hit
4. **Switch hits stop and the active delay timer expires**
   i. The spinner switches to "inactive" state
   ii. The spinner posts *spinner_<name>_inactive* event
   iii. (Optional) If `idle_ms:` is defined, the spinner sets a timeout for idle_ms duration
5. **(Optional) No switch hits occur and the idle delay timer expires**
   i. The spinner posts *spinner_<name>_idle* event
   ii. The spinner switches to "idle" state

Optional settings

The following sections are optional in the `spinners:` section of your config. (If you don't include them, the default will be used).

active_ms:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `1000ms`

How long the spinner should stay active after the last switch hit. The hit count resets each time the spinner becomes active, so this value determines when one group of spins ends and the next begins.

disable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Events in this list, when posted, disable this spinner. If a spinner is disabled, then hits to it have no effect.

enable_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Events in this list, when posted, enable this spinner. If a spinner is not enabled, then hits to it have no effect.

idle_ms:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Defaults to empty.

How long the spinner should stay inactive before going idle. This time is counted *after* the `active_ms:` has expired, and is useful for displaying slides or widgets for a while after switch hits stop.

labels:

List of one (or more) values, each is a type: `string`. Defaults to empty.

A list of labels to apply to the switches in the spinner. If used, the number of labels should equal the number of switches.

When a spinner switch is hit and `labels:` are defined, additional events will be posted with *spinner_<name>_<label>_active* and *spinner_<name>_<label>_hit*. This allows the game to trigger different behavior based on which spinner switch is hit first or spins more times.

playfield:

Single value, type: string name of a *playfields* device. Default: `playfield`

The name of the playfield that this spinner is on. The default setting is "playfield", so you only have to change this value if you have more than one playfield and you're managing them separately.

reset_when_inactive:

Single value, type: `boolean` (`true`/`false`). Default: `true`

When true, the spinners `hit` count will reset when the spinner goes inactive (after the `active_ms:` expires).

When false, the spinner's `hit` count will reset when the spinner goes idle (after the `idle_ms:` expires)

This value has no effect if `idle_ms:` is not set.

switch:

List of one (or more) values, each is a type: string name of a *switches* device. Defaults to empty.

The name of the switch (or a list of switches) for this spinner. You can use multiple switches if the playfield has a series of spinners that work together (for example at both ends of a horseshoe loop).

switches:

List of one (or more) values, each is a type: string name of a *switches* device. Defaults to empty.

This setting is the same as the `switch:` setting above. You can technically enter a single switch or a list of switches in either the `switch:` setting or the `switches:` setting, but we include both since it was confusing to be able to enter multiple switches for a singlular "switch" setting and vice versa.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

See the *documentation on the debug setting* for details.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Todo: *Help us to write it*

Related How To guides

Todo: *Help us to write it*

smart_virtual:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `smart_virtual:` section of your config is where you configure the *smart virtual platform*.

Optional settings

The following sections are optional in the `smart_virtual:` section of your config. (If you don't include them, the default will be used).

console_log:

Single value, type: one of the following options: none, basic, full. Default: `none`

Log level for the console log for this platform.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

File level for the console log for this platform.

simulate_manual_plunger:

Single value, type: `boolean` (`true`/`false`). Default: `false`

When `simulate_manual_plunger` is set to `True` the smart_virtual platform will automatically plunge balls in devices with mechanical eject after `simulate_manual_plunger_timeout` ms.

simulate_manual_plunger_timeout:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `10s`

When `simulate_manual_plunger` is set to `True` the smart_virtual platform will automatically plunge balls in devices with mechanical eject after `simulate_manual_plunger_timeout` ms.

Related How To guides

- *The "Smart Virtual" Platform*

smartmatrix:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `smartmatrix:` section of your config is where you configure RGB DMD devices.

This is an example:

```
#config_version=5
hardware:
  rgb_dmd: smartmatrix
smartmatrix:
  my_smartmatrix:
    port: com4
    baud: 4000000
displays:
  dmd:
    width: 128
    height: 32
rgb_dmds:
  my_smartmatrix:
    hardware_brightness: .5
```

Required settings

The following sections are required in the `smartmatrix:` section of your config:

baud:

Single value, type: `integer`. Defaults to empty.

Baud rate of your serial port. Depends on the smartmatrix firmware.

port:

Single value, type: `string`. Defaults to empty.

Name of the serial port of your smartmatrix device. This will be *comX* on Windows. On Linux and Mac it depends on the usb-serial chip (usually /dev/ttyUSBX on linux or /dev/tty.usbmodemYYY on Mac).

Optional settings

The following sections are optional in the `smartmatrix:` section of your config. (If you don't include them, the default will be used).

console_log:

Single value, type: one of the following options: none, basic, full. Default: `none`

Todo: *Help us to write it*

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Todo: *Help us to write it*

old_cookie:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set to true to use the old cookie. Will use the new cookie by default.

Related How To guides

- *How to configure a "SmartMatrix" RGB LED DMD*

snux:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `snux:` section of your config is where you configure the *snux platform*.

This is an example:

```
hardware:
  platform: virtual    # use your platform here
  driverboards: wpc
  coils: snux
  switches: snux
system11:
  ac_relay_delay_ms: 75
  ac_relay_driver: c_ac_relay
snux:
  diag_led_driver: c_diag_led_driver
coils:
  c_diag_led_driver:
    number: c24
    default_hold_power: 1.0
  c_flipper_enable_driver:
    number: c23
    default_hold_power: 1.0
  c_ac_relay:
    number: c25
    default_hold_power: 1.0
  c_side_a1:
    number: c11a
  c_side_a2:
    number: c12a
    default_hold_power: 0.5
  c_side_c1:
    number: c11c
  c_side_c2:
    number: c12c
    default_hold_power: 0.5
  c_flipper_left_main:
    number: FLLM
  c_flipper_left_hold:
    number: FLLH
    allow_enable: true

switches:
  s_flipper_left:
    number: sf01
  s_test:
    number: s77

flippers:
  f_test_single:
    main_coil: c_flipper_left_main
    hold_coil: c_flipper_left_hold
    activation_switch: s_flipper_left
```

Required settings

The following sections are required in the `snux:` section of your config:

diag_led_driver:

Single value, type: string name of a *coils* device. Defaults to empty.

The coil to use to drive the diag LED on the snux board. This is usually driver 23 on the Snux board.

Optional settings

The following sections are optional in the `snux:` section of your config. (If you don't include them, the default will be used).

console_log:

Single value, type: one of the following options: none, basic, full. Default: `none`

Log level for the console log for this platform.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this platform.

Related How To guides

WPC Platform to connect to the *SNUX board*.

sound_ducking:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

The `ducking:` setting in your `sounds:` section of your config is where you configure ducking settings for a sound.

Required settings

The following sections are required in the `sound_ducking:` section of your config:

target:

List of one (or more) events.

The list of track names to apply the ducking to when the sound is played. This most commonly contains the name of the track that music is played on.

Optional settings

The following sections are optional in the `sound_ducking:` section of your config. (If you don't include them, the default will be used).

attack:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `10ms`

The duration of the period over which the ducking starts until it reaches its maximum attenuation (attack stage). This value is specified as a *time string*.

attenuation:

Single value, type: `gain setting` (-inf, db, or float between 0.0 and 1.0). Default: `1.0`

The attenuation (gain) to apply to the target track while ducking. `attenuation:` controls how quiet to make the target track while the sound is playing.

delay:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `0`

The duration to delay after the sound starts playing before ducking starts. This value is specified as a *time string*.

release:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `10ms`

The duration of the period over which the ducking goes from its maximum attenuation until the ducking ends (release stage). This value is specified as a *time string*.

release_point:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `0`

The point relative to the end of the sound at which to start the returning the attenuation back to normal (release stage). A value of 0.5 seconds means to begin to release the ducking 0.5 seconds prior to the end of the sound. This value is specified as a *time string*.

Related How To guides

- *Ducking*

sound_loop_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**
Valid in *shows*	**YES**

Note: This section can also be used in a show file in the `sound_loops:` section of a step.

The `sound_loop_player:` section of your config is where you specify actions to perform on *sound loop sets* when MPF events are received.

Examples:

```
sound_loop_player:
  play_basic_beat:
    loops:
      action: play
      sound_loop_set: basic_beat
      timing: loop_end
  add_hi_hats:
    loops:
      action: play_layer
      layer: 1
      timing: loop_end
  stop_hi_hats:
    loops:
      action: stop_looping_layer
      layer: 1
  add_snare:
    loops:
      action: play_layer
      fade_in: 2s
      layer: 2
      timing: now
  add_claps:
    loops:
      action: play_layer
      layer: 3
      timing: loop_end
```

Additional information may be found in the *sound_player* documentation.

Express configuration

The `sound_loop_player` does not support an express configuration.

Required settings

The following sections are required in the `sound_loop_player:` section of your config:

track:

Single value, type: `string`.

This is the name of the track on which to perform the specified action. This must be an existing sound loop track. (You configure tracks and track names in the *sound_system:* section of your machine config files.)

Optional settings

The following sections are optional in the `sound_loop_player:` section of your config. (If you don't include them, the default will be used).

action:

Single value, type: one of the following options: play, stop.

The `action:` setting controls what action will be performed on the specified sound loop set. The other settings for each action vary (additional details may be found below). Options for `action:` are:

- `play` - The specified sound loop set will be played. Additional settings control whether the playback will begin immediately or after the currently playing loop set reaches the end of the master sound. Will cross-fade with the currently playing sound loop set if a `fade_in` setting is used.
- `stop` - The currently playing sound loop set will be stopped. Will fade out before stopping if a `fade_out` setting is used.
- `stop_looping` - Looping will be cancelled for the currently playing sound loop set (the sound loop set will continue to play to the end of the current loop).
- `play_layer` - Plays the sound on the specified layer in the currently playing loop set. Additional settings control whether the layer will begin immediately or will wait until after the currently playing loop set reaches the end of the sound. Will fade in if a `fade_in` setting is used.
- `stop_layer` - Stops the sound on the specified layer in the currently playing loop set. Will fade out before stopping if a `fade_out` setting is used.
- `stop_looping` - Looping will be cancelled for the sound on the specified layer in the currently playing sound loop set (the sound on the layer will continue to play to the end of the current loop).

Settings for *play* action:

Only the `sound_loop_set:` setting is required for the *play* action.

sound_loop_set:

Single value, type: `string`.

This is the name of the `sound_loop_set` asset used to perform the specified action. This must be the name an existing `sound_loop_set` specified in the `sound_loop_sets:` section of your machine config files. This setting is required for the *play* action.

timing:

Single value, type: one of the following options: `now`, `loop_end`, `next_beat_interval`, `next_time_interval`. Default: `loop_end`

The `timing:` setting determines when the specified sound loop set should be played. If the sound loop track is not currently playing any sound, this value is ignored and the sound loop is played immediately. Options for `timing:` are:

- `now` - Play the specified sound loop set immediately, even if another sound loop is currently playing. If the `fade_in:` parameter has a non-zero value, the sound loops will be cross-faded over the `fade_in:` time interval.
- `loop_end` - Play the specified sound loop set as soon as the currently playing sound loop reaches the end of the loop. This will be a gapless switch. The `fade_in:` setting is ignored when `loop_end` is used.
- `next_beat_interval` - Switch to the specified sound loop set on a beat interval of the currently playing sound loop. In order for this to work well the `tempo:` setting must be accurately set in all sound_loop_set assets. This setting works in conjunction with the `interval:` setting to determine the next beat interval to use when switching sound loops. For example, a setting of `1` indicates the switch can occur on any beat while a setting of `4` indicates the sound loops may only be switched every 4 beats (counted from the beginning of the currently playing sound loop set). This is useful to ensure sound loop sets are switched only at musically useful times.
- `next_time_interval` - Switch to the specified sound loop set on a time interval of the currently playing sound loop. This setting works in conjunction with the `interval:` setting to determine the next time interval to use when switching sound loops. For example, a setting of `1` indicates the switch can occur on any second boundary while a setting of `2.5` indicates the sound loops may only be switched every 2.5 seconds (counted from the beginning of the currently playing sound loop set).

interval:

Single value, type: `float`. Default: `1`

Used in conjunction with the `timing: next_beat_interval` and `timing: next_time_interval` setting values, this setting determines the next beat or time interval to use when switching sound loop sets.

synchronize:

Single value, type: `boolean` (Yes/No or True/False). Default: `False`

Indicates whether or not the sound loop will be synchronized in time with the currently playing sound loop. This setting only applies when using the `timing: now` setting value. It most useful to smoothly cross-fade between different variations of the same sound loop.

volume:

Single value, type: `gain setting` (*Instructions for entering gain values*) -inf, db, or float between 0.0 and 1.0. Default: None (Uses the volume setting of the sound_loop_set asset specified in the `sound_loop_set:` setting.

The volume of the specified sound loop master sound (overrides the setting in the sound asset section). This value only controls the master sound and not any layers defined in the sound loop set. As with all volume parameters in MPF, this item can be represented as a number between 0.0 and 1.0 (1.0 is max volume, 0.0 is off, 0.9 is 90%, etc.) It also can be represented as a decibel string from -inf to 0.0 db (ex: `-3.0 db`).

fade_in:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `0`

The number of seconds over which to fade in the sound loop set when it is played (when cross-fading between sound loops).

fade_out:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `0`

The number of seconds over which to fade out the sound loop set when it is stopped. This value is not applied when the sound stops on its own by reaching the end of the sound. It only comes into play when the sound is actively stopped by an event. A fade out sounds much more professional than an abrupt cutoff of a sound.

start_at:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `0`

The position in the sound loop file (in seconds) to start playback of the sound loop when it is played. When the sound loop is looped it will loop back to the beginning of the sound file.

events_when_played:

List of one (or more) values, each is a type: `string`. Default: `use_sound_loop_setting`

A list of one or more names of events that MPF will post when this sound loop set is played. Enter the list in the MPF config list format. These events are posted exactly as they're entered. When set to `use_sound_loop_setting`, the `events_when_played:` setting value specified in the sound loop set will be used.

events_when_stopped:

List of one (or more) values, each is a type: `string`. Default: `use_sound_loop_setting`

A list of one or more names of events that MPF will post when this sound loop set stops playing. Enter the list in the MPF config list format. These events are posted exactly as they're entered. When set to `use_sound_loop_setting`, the `events_when_stopped:` setting value specified in the sound loop set will be used.

events_when_looping:

List of one (or more) values, each is a type: `string`. Default: `use_sound_loop_setting`

A list of one or more names of events that MPF will post when this sound loop set loops back to the beginning while playing. Enter the list in the MPF config list format. These events are posted exactly as they're entered. When set to `use_sound_loop_setting`, the `looping:` setting value specified in the sound loop set will be used.

Settings for *stop* action:

No settings are required for the *stop* action.

fade_out:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `0`

The number of seconds over which to fade out the sound loop set when it is stopped. This value is not applied when the sound stops on its own by reaching the end of the sound. It only comes into play when the sound is actively stopped by an event. A fade out sounds much more professional than an abrupt cutoff of a sound.

Settings for *stop_looping* action:

There are no settings available for the *stop_looping* action.

Settings for *jump_to* action:

The `time:` setting is required for the *jump_to* action.

time:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `0`

The position in the sound loop file (in seconds) to immediately jump to during playback of the current sound loop. When the sound loop reaches the end of the sound, it will loop back to the beginning of the sound file.

Settings for *play_layer* action:

The `layer:` setting is required for the *play_layer* action. This action has no effect if there is no sound loop set currently playing on the specified track.

layer:

Single value, type: `integer`.

An integer value that specifies which layer number of the currently playing sound loop set should be played. Layers are numbered beginning with 1.

timing:

Single value, type: one of the following options: `now`, `loop_end`. Default: `loop_end`

The `timing:` setting determines when the specified layer should be played. Layers are always played in synchronized time with the master sound in the currently playing sound loop set. Options for `timing:` are:

- `now` - Play the specified layer immediately. If the `fade_in:` parameter has a non-zero value, the layer will faded in over the `fade_in:` time interval.
- `loop_end` - Play the specified layer as soon as the currently playing sound loop reaches the end of the loop. If the `fade_in:` parameter has a non-zero value, the layer will faded in over the `fade_in:` time interval.

volume:

Single value, type: `gain setting` (*Instructions for entering gain values*) -inf, db, or float between 0.0 and 1.0. Default: None (uses the volume setting of the sound asset specified in the layer `sound:` setting.

The volume of the specified layer sound (overrides the setting in the sound asset section). As with all volume parameters in MPF, this item can be represented as a number between 0.0 and 1.0 (1.0 is max volume, 0.0 is off, 0.9 is 90%, etc.) It also can be represented as a decibel string from -inf to 0.0 db (ex: `-3.0 db`).

fade_in:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `0`

The number of seconds over which to fade in the sound loop set layer when it is played.

Settings for *stop_layer* action:

The `layer:` setting is required for the *stop_layer* action. This action has no effect if there is no sound loop set currently playing on the specified track or if the specified layer is not currently playing.

layer:

Single value, type: `integer`.

An integer value that specifies which layer number of the currently playing sound loop set should be stopped. Layers are numbered beginning with 1.

fade_out:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `0`

The number of seconds over which to fade out the sound loop set layer when it is stopped.

Settings for *stop_looping_layer* action:

The `layer:` setting is required for the *stop_looping_layer* action. This action has no effect if there is no sound loop set currently playing on the specified track or if the specified layer is not currently playing.

layer:

Single value, type: `integer`.

An integer value that specifies which layer number of the currently playing sound loop set should be stopped when the sound loop set master sound reaches the end. Layers are numbered beginning with 1.

sound_loop_sets:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `sound_loop_sets:` section of your config is where you pre-define "named" sound loop sets for playback in a sound loop audio track using sound_loop_player section of a config file.

Sound loop sets are special groupings of existing sound assets (See the *sounds:* reference page for more details on sound assets.)

Example 1: Simple Sound Loop Set

If you want to define a sound loop set that is made up of only a single sound, you can just add the sound name to the sound loop set. In the example below, we're defining a sound loop set called *basic_beat* that references the sound asset named *kick*. This is the simplest sound loop set definition you can have. The volume of the *kick* sound will be taken from the sound asset definition.

```
sound_loop_sets:
  basic_beat:
    sound: kick
```

Option 2: Sound Loop Set With Multiple Layers

When specifying multiple layers use a dash (and a space) to dictate where a new layer starts, like this:

```
sound_loop_sets:
  basic_beat:
    sound: kick
    volume: 0.5
    tempo: 130.0
    layers:
      - sound: hihat
        volume: 0.7
```

(continues on next page)

(continued from previous page)

```
        initial_state: stop
      - sound: snare
        volume: 0.6
        initial_state: stop
      - sound: clap
        volume: 0.45
        initial_state: stop
    events_when_played: basic_beat_played
    events_when_stopped: basic_beat_stopped
    events_when_looping: basic_beat_looped
    fade_out: 1s
  basic_beat2:
    sound: kick2
    volume: 0.5
    tempo: 130.0
    layers:
      - sound: hihat
        volume: 0.7
      - sound: snare
        volume: 0.6
      - sound: clap
        volume: 0.4
        initial_state: stop
      - sound: bass_synth
        volume: 0.5
        initial_state: play
    fade_out: 1s
```

Required settings

The following sections are required for each named sound loop set in your config:

sound:

Single value, type: `string`.

The name of the sound asset that will be used as the master sound in the sound loop set. This must refer to an existing sound asset or an error will be thrown during initialization. The sound asset also must be stored in memory (and not streaming). Do not include the sound file extension here, only the sound asset name.

Optional settings

The following sections are optional in the `sound_loop_sets:` section of your config. (If you don't include them, the default will be used).

volume:

Single value, type: `gain setting` (*Instructions for entering gain values*) -inf, db, or float between 0.0 and 1.0. Default: Uses the volume setting of the sound asset specified in the `sound:` setting.

The volume of the specified sound (overrides the setting in the sound asset section). This value only controls the master sound and not any layers defined in the sound loop set. As with all volume parameters in MPF, this item can be represented as a number between 0.0 and 1.0 (1.0 is max volume, 0.0 is off, 0.9 is 90%, etc.) It also can be represented as a decibel string from -inf to 0.0 db (ex: `-3.0 db`).

events_when_played:

List of one (or more) values, each is a type: `string`. Default: `None`

A list of one or more names of events that MPF will post when this sound loop set is played. Enter the list in the MPF config list format. These events are posted exactly as they're entered.

events_when_stopped:

List of one (or more) values, each is a type: `string`. Default: `None`

A list of one or more names of events that MPF will post when this sound loop set stops playing. Enter the list in the MPF config list format. These events are posted exactly as they're entered.

events_when_looping:

List of one (or more) values, each is a type: `string`. Default: `None`

A list of one or more names of events that MPF will post when this sound loop set loops back to the beginning while playing. Enter the list in the MPF config list format. These events are posted exactly as they're entered.

fade_in:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `0`

The number of seconds over which to fade in the sound loop set when it is played.

fade_out:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `0`

The number of seconds over which to fade out the sound loop set when it is stopped. This value is not applied when the sound stops on its own by reaching the end of the sound. It only comes into play when the sound is actively stopped by an event. A fade out sounds much more professional than an abrupt cutoff of a sound.

tempo:

Single value, type: `float`. Default: `60.0`

The tempo of the sound loop set, expressed in beats per minute. This setting is used to calculate the timing of beat intervals when switching between sound loops. This setting only needed when using the `timing: next_beat_interval` setting in the (*sound_loop_player*).

layers:

The `layers:` section controls the additional sound layers for the sound loop set. It contains the following nested sub-settings:

Required settings

The following sections are required in the `layers:` section of your config:

sound:

Single value, type: `string`.

The name of the sound asset that will be used in the sound loop set layer. This must refer to an existing sound asset or an error will be thrown during initialization. The sound asset also must be stored in memory (and not streaming). Do not include the sound file extension here, only the sound asset name.

Optional settings

The following sections are optional in the `layers:` section of your config. (If you don't include them, the default will be used).

volume:

Single value, type: `gain setting` (*Instructions for entering gain values*) -inf, db, or float between 0.0 and 1.0. Default: Uses the volume setting of the sound asset specified in the layer `sound:` setting.

The volume of the specified sound in the layer (overrides the setting in the sound asset section). As with all volume parameters in MPF, this item can be represented as a number between 0.0 and 1.0 (1.0 is max volume, 0.0 is off, 0.9 is 90%, etc.) It also can be represented as a decibel string from -inf to 0.0 db (ex: `-3.0 db`).

initial_state:

Single value, type: one of the following options: play, stop. Default: `play`

The `initial_state:` of a sound loop set layer determines the initial play state for the layer when the sound loop set is played. Options for `initial_state:` are:

- `play` - The layer will be played whenever the sound loop set begins playback.
- `stop` - The layer will be stopped whenever the sound loop set begins playback.

sound_marker:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

The `markers:` setting in your `sounds:` section of your config is where you configure markers which trigger events at certain points in playback.

Required settings

The following sections are required in the `sound_marker:` section of your config:

events:

List of one (or more) events.

A list of one or more names of events that MPF will post when this marker is reached during sound playback. Enter the list in the MPF config list format. These events are posted exactly as they're entered.

time:

Single value, type: `time string (secs)` (*Instructions for entering time strings*).

The marker time (in seconds) relative to the beginning of the sound file.

Related How To guides

- *Sounds, Music & Audio*

sound_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

Note: This section can also be used in a show file in the `sounds:` section of a step.

The `sound_player:` section of your config is where you specify actions to perform on sounds when MPF events are received.

This is an example:

```
sound_player:
  mode_attract_started:
    song_01:
      action: play
      loops: -1
  mode_attract_stopped:
    song_01:
      action: stop
  slingshot_hit:
```

(continues on next page)

(continued from previous page)

```
    zap:
      block: true  # "blocks" this event from being passed to sound player sections in lower-priority␣
↪modes
```

Additional information may be found in the *sound_player* documentation.

Express configuration

When referencing sounds in the sound player, there is an alternative syntax to specify a sound when you don't wish to provide any additional settings. This shortcut notation is known as the "express configuration" and for the sound player it is simply the name of the sound asset. It can be used in both configuration files and show steps. In the config file example above, `play_sound_slingshot: slingshot_01` is an example using the express configuration (sound name only).

Sound behavior upon mode (or show) stop

When the mode or show stops that contains a `sound_player`, all sounds started in that mode or show will continue to play and stop automatically when they reach their end. Sounds that are looping will have their looping stopped so the sound will no longer continue to loop and will stop when they reach their end. Sounds that are pending playback and are queued will be canceled (removed from the queue) and will not be played. If you need a sound to be stopped immediately when a mode or show ends, you will need to add an entry in the `sound_player` to trigger a stop action based on the mode or show stop event.

Optional settings

The following sections are optional in the `sound_player:` section of your config. (If you don't include them, the default will be used).

about_to_finish_time:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `-1`

Todo: *Help us to write it*

action:

Single value, type: one of the following options: play, stop, stop_looping, load, unload. Default: `play`

The `action:` setting controls what action will be performed on the specified sound. Options for `action:` are:

- `play` - The specified sound will be played. Any optional parameter values will override the sound's settings.

- `stop` - All currently playing and queued instances of the specified sound will stopped/canceled. Any optional parameter values will be ignored as the stop action takes no parameters. There is currently no way to stop specific instances of a particular sound while leaving others playing, but that is on the list to be implemented in a future version.
- `stop_looping` - Looping will be canceled for all currently playing instances of the specified sound (the sound will continue to play to the end of the current loop). In addition, any queued instances of the sound awaiting playback will be removed/canceled.
- `load` - Loads the specified sound or sound pool from its source file into memory to prepare it to be played. The request is ignored if the sound is already loaded.
- `unload` - Unloads the specified sound or sound pool from memory. All instances of the sound or sound pool will be immediately stopped. The request is ignored if the sound is not currently loaded.

block:

Single value, type: `boolean` (`true`/`false`). Default: `false`

When set to `true`, the triggering event is blocked from being passed to other sound_player sections in lower priority modes. This is useful if you have a switch in a base mode that plays a sound (like a jet bumper), but then in a special mode (like super jets) you want that switch to play a different sound but you don't also want the base mode to play the sound configured there (we don't want two simultaneous sounds for the jet bumper, just one).

```
##! mode: mode1
sound_player:
  sw_jet_bumper_active:
    super_jet_bumper_sound:
      block: true
```

There is also a shorthand way (express config format):

```
##! mode: mode1
sound_player:
  sw_jet_bumper_active: super_jet_bumper_sound|block
```

delay:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Defaults to empty.

When the triggering event occurs, delay for a certain amount of time before playing the sound.

events_when_about_to_finish:

List of one (or more) events. Those will be posted by the device. Default: `use_sound_setting`

Please refer to the *sounds:* documentation for details about this setting as it just overwrites the setting in your sound.

events_when_looping:

List of one (or more) events. Those will be posted by the device. Default: `use_sound_setting`

Please refer to the *sounds:* documentation for details about this setting as it just overwrites the setting in your sound.

events_when_played:

List of one (or more) events. Those will be posted by the device. Default: `use_sound_setting`

Please refer to the *sounds:* documentation for details about this setting as it just overwrites the setting in your sound.

events_when_stopped:

List of one (or more) events. Those will be posted by the device. Default: `use_sound_setting`

Please refer to the *sounds:* documentation for details about this setting as it just overwrites the setting in your sound.

fade_in:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Defaults to empty.

Please refer to the *sounds:* documentation for details about this setting as it just overwrites the setting in your sound.

fade_out:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Defaults to empty.

Please refer to the *sounds:* documentation for details about this setting as it just overwrites the setting in your sound.

key:

Single value, type: `string`. Default: `use_sound_setting`

Used to reference this sound entry when stopping/pausing/resuming it.

loops:

Single value, type: int_or_token. Defaults to empty.

Please refer to the *sounds:* documentation for details about this setting as it just overwrites the setting in your sound.

max_queue_time:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `-1`

Please refer to the *sounds:* documentation for details about this setting as it just overwrites the setting in your sound.

mode_end_action:

Single value, type: one of the following options: stop, stop_looping, use_sound_setting. Default: `use_sound_setting`

Please refer to the *sounds:* documentation for details about this setting as it just overwrites the setting in your sound.

pan:

Single value, type: float_or_token. Defaults to empty.

Please refer to the *sounds:* documentation for details about this setting as it just overwrites the setting in your sound.

priority:

Single value, type: int_or_token. Defaults to empty.

Please refer to the *sounds:* documentation for details about this setting as it just overwrites the setting in your sound.

start_at:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Defaults to empty.

Please refer to the *sounds:* documentation for details about this setting as it just overwrites the setting in your sound.

track:

Single value, type: `string`. Defaults to empty.

Please refer to the *sounds:* documentation for details about this setting as it just overwrites the setting in your sound.

volume:

Single value, type: `gain setting` (-inf, db, or float between 0.0 and 1.0). Defaults to empty.

Please refer to the *sounds:* documentation for details about this setting as it just overwrites the setting in your sound.

Related How To guides

- *Sounds, Music & Audio*

sound_pools:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `sound_pools:` section of your config is where you specify pools (or groupings) of sound assets in your machine.

Creating a sounds pool allows you to reference a group of sound variations as if it were a single sound. A sound pool name may be used anywhere a sound asset name may appear. Pools can be used for random differences in a sound (such as slight variations of a slingshot sound) or for an ordered sequence of sounds that will repeat. Another common use for sound pools is to play a random callout from a defined list when triggered. (Sound pools are part of the MPF media controller and only available if you're using MPF-MC for your media controller.)

Here's an example of a typical sound_pool configuration.

```
sound_pools:
  drain_callout:
    type: random_force_all
    track: voice
    sounds:
      - drain_01
      - drain_02
      - drain_03
      - drain_04
  slingshot:
    load: preload
    type: random
    track: sfx
    sounds:
      - slingshot_01|5
      - slingshot_02|3
      - slingshot_03|2
  target_completion:
    load: on_demand
    type: sequence
    track: sfx
    sounds:
      - target_completion_01
      - target_completion_02
      - target_completion_03
```

To create a sound pool, add a sub entry to the `sound_pools:` section of your config which will be the name of that sound pool. The name must be unique among all sound pools *and* sound assets. In the above example *drain_callout:*, *slingshot:* and *target_completion:* are each a sound pool name. Then create one or more of the following settings for each sound pool:

Required settings

The following sections are required for each named sound pool in your config:

sounds:

The `sounds:` section contains an indented list of existing sound assets (one per line) that will be contained in the sound pool. It is suggested you use block sequence notation for this list (begin each line with a dash followed by a space `- ``). Optionally, a number may be appended to the sound asset name delimited by a pipe (``|`) character. This optional number controls the relative weighting for random item selection, or the number of times to play the sound before moving to the next sound in the pool with a sequence pool. If no weight value is provided, a default value of `1` will be applied. In the example above, the *slingshot:* random sound pool contains relative weighting values. The weights sum to 10 for the three sounds so the *slingshot_01* sound has a probability of being randomly selected of 5 out of 10 (50%), *slingshot_02* 3/10 (30%), and *slingshot_03* 2/10 (20%).

Note: If you want to use a sound that has spaces in its name, the name of the sound must be in quotes:

```
sound_pools:
drain_callout:
   type: random_force_all
   track: voice
   sounds:
      - drain_01
      - drain_02
      - "drain 03" # example of a sound with a space in its name using quotes
      - drain_04
```

track:

Single value, type: `string`. Default: `None`

This is the name of the track this sound pool will play on. (You configure tracks and track names in the *sound_system:* section of your machine config files.)

Optional settings

The following sections are optional for each named sound pool in your config. (If you don't include them, the default will be used).

load:

Single value, type: one of the following options: preload, on_demand. Default: `on_demand`

This controls the timing of when the sound assets in the sound pool will be loaded into memory (see the documentation on (*Managing Assets* for an explanation of what loading is). Options for `load:` are:

- `preload` - The asset is loaded when MPF boots and stays in memory as long as MPF is running.

- `on_demand` - The asset is loaded "on demand" when it's first called for. At this point, assets loaded on demand stay in memory forever, but at some point we'll change that so they can be unloaded on demand too.

type:

Single value, type: one of the following options: sequence, random, random_force_next, random_force_all. Default: `sequence`

The `type:` of sound pool dictates how the next sound in the pool will be selected when the sound pool is referenced for playback. Options for `type:` are:

- `sequence` - Sounds are selected in the order in which they appear in the `sounds:` section. An optional number/weight appended after each sound controls how many times the sound will be played before the next one in the list is selected. The sequence of sounds will repeat once all sounds have been played.
- `random` - Sounds are randomly selected from the list of sounds in the `sounds:` section of the sound pool. The probability of selecting each sound in the list can be controlled by an optional numeric weight value appended after each sound. This weight value is relative to all other sounds in the list.
- `random_force_next` - Sounds are randomly selected from the list of sounds in the `sounds:` section of the sound pool. This sound pool type ensures that the next sound selected will not be the same as the previously selected sound (no back-to-back repeats of a single sound). The probability of selecting each sound in the list can be controlled by an optional numeric weight value appended after each sound. This weight value is relative to all other sounds in the list.
- `random_force_all` - Sounds are randomly selected from the list of sounds in the `sounds:` section of the sound pool. This sound pool type ensures that all sounds in the list will be played once before any sound will be repeated. The probability of selecting each sound in the list can be controlled by an optional numeric weight value appended after each sound. This weight value is relative to all other sounds in the list.

simultaneous_limit:

Single value, type: `integer`. Default: `None`

The numeric value indicating the maximum number of instances of this sound pool that may be played at the same time (up to the limit of the track). Once the maximum number of instances has been reached, the `stealing_method` setting determines the how additional requests to play the sound pool will be managed. This setting is useful for sounds that can be triggered in rapid succession (such as spinners and pop bumpers). Setting a limit will ensure a reasonable number of instances will be played simultaneously and not overwhelm the audio mix. The default value of `None` indicates no limits will be placed on the number of instances of the sound pool that may be played at once up to the limit of the track.

Note: The sounds contained in a sound pool can also have their own `simultaneous_limit` setting which can lead to some unexpected behavior when interacting with the `simultaneous_limit` setting in the sound pool.

stealing_method:

Single value, type: one of the following options: oldest, newest, skip. Default: `oldest`

The `stealing_method:` of a sound pool determines the behavior of additional requests to play the sound pool once the number of simultaneous instances of the sound has reached its `simultaneous_limit` limit. This setting is ignored when `simultaneous_limit` is set to `None`. Options for `stealing_method:` are:

- `oldest` - Steal/stop the oldest playing instance of the sound and replace it with a new instance (essentially restarts the oldest playing instance).
- `newest` - Steal/stop the newest playing instance of the sound and replace it with a new instance (essentially restarts the newest playing instance).
- `skip` - Do not steal/stop any currently running instances of the sound. Simply skip playback of the newly requested instance.

sound_system_tracks:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

The `tracks:` settings in your `sound_system:` section of your config is where you configure which tracks exist in your machine.

Optional settings

The following sections are optional in the `sound_system_tracks:` section of your config. (If you don't include them, the default will be used).

crossfade_time:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `0`

Time to crossfade between to songs on your playlist.

The settings is specific to `playlist` audio tracks. It will ignored in other track types.

ducking:

Single value, type: *sound_ducking*.

Default ducking settings for this track. Those can be overwritten per sound. See *ducking* for details.

events_when_paused:

List of one (or more) events.

A list of one or more names of events that MPF will post when the track is paused. Enter the list in the MPF config list format. These events are posted exactly as they're entered.

events_when_played:

List of one (or more) events.

A list of one or more names of events that MPF will post when the track is played or resumed after being stopped/paused. Enter the list in the MPF config list format. These events are posted exactly as they're entered.

events_when_resumed:

List of one (or more) events.

A list of one or more names of events that MPF will post when the track is resumed. Enter the list in the MPF config list format. These events are posted exactly as they're entered.

events_when_stopped:

List of one (or more) events.

A list of one or more names of events that MPF will post when the track is stopped. Enter the list in the MPF config list format. These events are posted exactly as they're entered.

max_layers:

Single value, type: `integer`. Default: `8`

Maximum number of layers in your loop which can play in parallel.

The settings is specific to `sound_loop` audio tracks. It will ignored in other track types.

simultaneous_sounds:

Single value, type: `integer`. Default: `8`

This sets the maximum number of simultaneous sounds that can be played on this track. The example config file above shows the *music* and *voice* tracks with a max of 1 simultaneous sound playing, since if you have two music clips or voice callouts playing at the same time, it will sound like gibberish. A sound effects track, on the other hand, can probably have a few sounds playing at once. Note that MPF gives you detailed control over what happens if a new sound wants to play when the max simultaneous sounds are already playing on that track. Should the new sound break in and stop an existing sound? Should it wait until the existing sound is done? How long should it wait? You can control all this on a per sound basis (see the :doc:~sounds: </config/sounds>~ documentation for more information).

The settings is specific to `standard` audio tracks. It will ignored in other track types.

type:

Single value, type: one of the following options: standard, sound_loop, playlist. Default: `standard`

The track `type:` setting determines what type of audio track will be used. For more detailed

volume:

Single value, type: `gain setting` (-inf, db, or float between 0.0 and 1.0). Default: `0.5`

This is the volume setting for this track (how loud will it be), as either a value between 0.0 and 1.0 or a decibel value between -inf and 0.0 db. Note that each track's volume will be combined with the overall system volume. So if your MPF master volume is set to 0.8 (80%) and you have a track set to 0.5 (50%), sounds on that track will play at 40% overall volume (50% of 80%).

Related How To guides

- *Sounds, Music & Audio*

sound_system:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `sound_system:` section of your machine config controls the general settings for the machine's sound system. (This section is part of the MPF media controller and only available if you're using MPF-MC for your media controller.)

Here's an example of a typical sound configuration.

```
machine_vars:
  master_volume:
    initial_value: 0.8

sound_system:
  buffer: 1024
  channels: 1
  enabled: true
  frequency: 44100
  tracks:
    music:
      type: standard
      simultaneous_sounds: 1
      volume: 0.5
    voice:
      type: standard
      simultaneous_sounds: 1
      volume: 0.7
    sfx:
```

(continues on next page)

(continued from previous page)

```
    type: standard
    simultaneous_sounds: 8
    volume: 0.4
```

Required settings

The following sections are required in the `sound_system:` section of your config:

tracks:

One or more sub-entries. Each in the format of `string` : *sound_system_tracks*

Every sound that's played in MPF is played on a track. If you are familiar with an audio mixer a track can be thought of as a mixer channel. Each track can have it's own settings, and you can set volume on a per-track basis. You can have up to 8 audio tracks in your MPF machine. The example above shows three tracks, called *music*, *voice*, and *sfx*. The idea (in case it isn't obvious) is that you play all your music clips on the music track, voice callouts on the voice track, and the sound effects on the sfx track. To create a track, add a sub entry to the *tracks:* section which will be the name of that track. (So again, *music:*, *voice:* and *sfx:* in the example.)

Optional settings

The following sections are optional in the `sound_system:` section of your config. (If you don't include them, the default will be used).

buffer:

Single value, type: `integer`. Default: `2048`

This is the size of your sound buffer. It must be a power of 2. The exact value you should use may take some trial-and-error. A bigger buffer means that there's less chance of skipping and dropout (lower CPU usage), but it also means that sounds can take longer to play since the buffer has to fill first. Some limited power platform have to run with a buffer of 4096 or 8192 or 16384, others at 512 or 256. So just play with it and see what works for you.

channels:

Single value, type: `integer`. Default: `1`

The number of channels the sound system will support. 1 for mono, 2 for stereo. You're probably thinking, "aww man, I need stereo sound!" But almost no pinball machines do this since the speakers in the backbox are 2 feet apart and they're 4 feet away from the player's ears. (Maybe if you're going to use headphones or put tweeters in the front of the machine?) Again, if you have a resource-constrained system, then go for mono and make sure all your sound files are mono. If not, meh, go ahead and use stereo.

enabled:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Indicates whether or not the sound system will be enabled in your machine.

frequency:

Single value, type: `integer`. Default: `44100`

How many sound samples per second you want. 44100 is so-called "CD quality" audio, though with the sound systems in most pinball machines, if you cut it in half (to 22050) it still sounds virtually the same. If you're running on a resource-constrained host computer, you should make sure all your sound files are encoded at the same rate so MPF doesn't waste time re-encoding them on the fly. Smaller values mean smaller sound files, less memory consumption, and less CPU processing. So if you're on a resource constrained host computer, think about 22050 instead of 44100. (But be sure to resample all your sound files to match.)

master_volume:

Unknown type. See description below.

DEPRECATED! Will removed in future MPF versions.

Master volume has been moved to the machine variable `master_volume`. You can use the following snippet:

```
machine_vars:
  master_volume:
    initial_value: 0.8
```

Note that this only controls the volume of the MPF app, not the host OS'es system volume. So you still need to make sure that the host OS is not on mute and that the volume is turned up.

Related How To guides

- *Sounds, Music & Audio*

sounds:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `sounds:` section of your config is where you configure non-default parameter values for any sound assets you want to use in your game. Note: You do *not* have to have an entry for every single sound you want to use, rather, you only need to add individual assets to your config file that have settings which different from other assets in that asset's folder. (This section is part of the MPF media controller and only available if you're using MPF-MC for your media controller.)

MPF-MC currently supports 16-bit Wave (.wav), Ogg Vorbis (.ogg), and FLAC (.flac) files.

Here's an example:

```
sounds:
  extra_ball:
    file: extra_ball_12753.wav
    events_when_stopped: extra_ball_callout_finished
    streaming: false
    track: voice
    volume: 0.5
    priority: 50
    max_queue_time: None
    ducking:
      target: music
      delay: 0
      attack: 0.3 sec
      attenuation: -18db
      release_point: 2.0 sec
      release: 1.0 sec
  slingshot_01:
    volume: 0.5
    max_queue_time: 0
```

Optional settings

The following sections are optional in the `sounds:` section of your config. (If you don't include them, the default will be used).

about_to_finish_time:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Defaults to empty.

The point relative to the end of the sound at which to post the `events_when_about_to_finish` event(s). A value of 0.5 seconds means to post the event(s) prior to the end of the sound. When set to `None`, no events will be posted. If the value of this setting is greater than the duration of the sound, the event(s) will be posted as soon as the sound begins playback. This value is specified as a *time string*.

ducking:

Single value, type: *sound_ducking*. Defaults to empty.

The `ducking:` section controls *ducking* for the sound.

events_when_about_to_finish:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A list of one or more names of events that MPF will post when this sound is about to finish playing. The exact timing of this event is determined by the `about_to_finish_time` setting for this sound. Enter the list in the MPF config list format. These events are posted exactly as they're entered.

events_when_looping:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A list of one or more names of events that MPF will post when this sound loops back to the beginning while playing. Enter the list in the MPF config list format. These events are posted exactly as they're entered.

events_when_played:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A list of one or more names of events that MPF will post when this sound is played. Enter the list in the MPF config list format. These events are posted exactly as they're entered.

events_when_stopped:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A list of one or more names of events that MPF will post when this sound stops playing. Enter the list in the MPF config list format. These events are posted exactly as they're entered. These events can be useful to trigger some action when a callout has finished playing.

fade_in:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `0`

The number of seconds over which to fade in the sound when it is played.

fade_out:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `0`

The number of seconds over which to fade out the sound when it is stopped. This value is not applied when the sound stops on its own by reaching the end of the sound (will likely be added in a future version). At the moment it only comes into play when the sound is actively stopped by an event.

file:

Single value, type: `string`. Defaults to empty.

Sometimes you might want to name a file one thing on disk but refer to it as another thing in your game and config files. In this case, you can create an `file:` setting in an asset entry. (Note the file: *extra_ball_12753.wav* setting in the example above, and note that it includes the file extension.) In this example, you would refer to that image asset as *extra_ball* even though the file is *extra_ball_12753*. You might be wondering why this exists? Why not just change the file name to be whatever you want and/or who cares what the name is? The reason this function exists is because it allows for the separation of the actual file on disk from the way it's called in the game. For example, you could use this to create two sets of assets—one for a traditional DMD and one for a color DMD—and then you could refer to the asset by its generic name throughout your configs. (In other words, you could swap out assets for different physical machine types without having to update your display code.) That said, we expect that 99% of people won't use this `file:` setting, which is fine.

key:

Single value, type: `string`. Defaults to empty.

Todo: *Help us to write it*

loop_end_at:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Defaults to empty.

The position in the sound file (in seconds) at which to start looping and return to the start of the loop as determined by the `loop_start_at:` setting. By default (None) the sound will loop when it reaches the end of the sound. This setting only applies to sounds loaded in memory and played on a standard audio track (not to any streaming sound or sound played on any other track type).

loop_start_at:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `0`

The position in the sound file (in seconds) to start playback of the sound after it is looped. By default when the sound is looped it will loop back to the beginning of the sound file. Setting this value to something other than zero is particularly useful when you have a music sound that has an introduction section and want it to loop back to a verse and not the intro. This setting works in correlation with `loop_end_at:` and only applies to sounds loaded in memory and played on a standard audio track (not to any streaming sound or sound played on any other track type). Be sure to use many decimal places in your times as precision is important when it comes to loop points. If you hear pops and clicks at the loop points, you may need to slightly adjust your start and end loop times to alleviate them.

loops:

Single value, type: `integer`. Default: `0`

An integer value that controls the looping behavior of this sound. A value of 0 indicates the sound will not loop when reaching the end (also known as a "one-shot"). A value of -1 indicates the sound should loop infinitely until it is stopped. A value greater than 0 specifies the number of times the sound should loop back to the beginning while playing. Note that this value is not the total number of times the sound is played, but the number of times it should play again after the first time through.

markers:

List of one (or more) values, each is a type: *sound_marker*. Defaults to empty.

The `markers:` section establishes a list of markers and their associated events at specific times in the sound. When a marker is reached during playback, the associated events will be posted. Markers are useful for synchronizing various actions with specific points in a sound. A typical use might be to send an 'almost_finished_playing' event a short time before a sound finishes playback or establish various checkpoints in a sound that could be used to restart a sound at that point on the user's next turn (using mode code).

Here's a simple example utilizing markers:

```
sounds:
  long_sound_1:
    volume: 0.8
    markers:
      - time: 2.534 sec
        events: send_this_event, also_this_event
      - time: 6.712 sec
        events: almost_finished_playing
```

max_queue_time:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Defaults to empty.

Specifies the maximum time this sound can be queued before it's played. If the time between when this sound is requested and when MPF can actually play it is longer than this queue time, then the request is discarded and the sound doesn't play. This only comes into play if this sound is requested but the track it's playing on is at its `simultaneous_sounds` limit. Then if this sound doesn't have a high enough priority to kill any of the existing sounds, it will be queued to play later. Some sounds (like voice callouts) might be ok to queue, but other sounds (like sound effects for when you hit a pop bumper or slingshot) might only make sense if they're played right away, so in those cases you might want to use a short (or no) queue time. The default setting is "None" which means this sound will have no queue limit and will always play eventually.

mode_end_action:

Single value, type: one of the following options: stop, stop_looping. Default: `stop_looping`

The `mode_end_action:` setting determines what action to take when the mode that initiates the playback of the sound ends. Options for `mode_end_action:` are:

- `stop` - All currently playing and queued instances of the specified sound started by the mode will be stopped/canceled. If the `fade_out` parameter has a non-zero value, the sound will fade out over the specified number of seconds.
- `stop_looping` - Looping will be canceled for all currently playing instances of the specified sound started by the mode (the sound will continue to play to the end of the current loop). In addition, any queued instances of the sound awaiting playback will be removed/canceled.

pan:

Single value, type: `number` (will be converted to floating point). Default: `0`

Pan the audio to the left or right channel. Currently, broken due to a bug. Let us know if you need this.

priority:

Single value, type: `integer`. Default: `0`

The numeric value indicating the priority or importance of this sound. Sounds with higher priority values will preempt other sounds with lower priorities that are playing when a track has reached the maximum number of simultaneous sounds it is configured to play. If the track is busy and the priorities of all sounds currently playing greater than or equal to this sound, the sound will be queued for playback and will have to wait to be played.

simultaneous_limit:

Single value, type: `integer`. Defaults to empty.

The numeric value indicating the maximum number of instances of this sound that may be played at the same time (up to the limit of the track). Once the maximum number of instances has been reached, the `stealing_method` setting determines the how additional requests to play the sound will be managed. This setting is useful for sounds that can be triggered in rapid succession (such as spinners and pop bumpers). Setting a limit will ensure a reasonable number of instances will be played simultaneously and not overwhelm the audio mix. The default value of `None` indicates no limits will be placed on the number of instances of the sound that may be played at once up to the limit of the track. The value of this setting is ignored when the `streaming` setting has a value of `False`.

start_at:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `0`

The position in the sound file (in seconds) to start playback of the sound when it is played. When the sound is looped it will loop back to the beginning of the sound file.

stealing_method:

Single value, type: one of the following options: skip, oldest, newest. Default: `oldest`

The `stealing_method:` of a sound determines the behavior of additional requests to play the sound once the number of simultaneous instances of the sound has reached its `simultaneous_limit` limit. This setting is ignored when `simultaneous_limit` is set to `None`. Options for `stealing_method:` are:

- `oldest` - Steal/stop the oldest playing instance of the sound and replace it with a new instance (essentially restarts the oldest playing instance).
- `newest` - Steal/stop the newest playing instance of the sound and replace it with a new instance (essentially restarts the newest playing instance).
- `skip` - Do not steal/stop any currently running instances of the sound. Simply skip playback of the newly requested instance.

streaming:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Indicates whether or not the sound sound will be streamed (rather than stored in memory). Streaming sounds are limited to a single instance of the sound playing at a time. Multiple different streaming sounds may be played simultaneously, just not more than a single instance of a particular sound. When `streaming` is set to `True`, the `simultaneous_limit` setting is ignored and a value of 1 is used.

track:

Single value, type: `string`. Defaults to empty.

This is the name of the track this sound will play on. (You configure tracks and track names in the *sound_system:* section of your machine config files.)

volume:

Single value, type: `gain setting` (-inf, db, or float between 0.0 and 1.0). Default: `0.5`

The volume of this sound. This value is factored into the track and overall MPF volumes. It's used to "balance" your sounds if you have one particular sound that's too loud or too quiet. As with all volume parameters in MPF, this item can be represented as a number between 0.0 and 1.0 (1.0 is max volume, 0.0 is off, 0.9 is 90%, etc.) It also can be represented as a decibel string from -inf to 0.0 db (ex: `-3.0 db`).

Related How To guides

- *Sounds, Music & Audio*

spi_bit_bang:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `spi_bit_bang:` section of your config is where you configure the *How to use SPI Bit Bang in MPF* platform.

Required settings

The following sections are required in the `spi_bit_bang:` section of your config:

clock_pin:

Single value, type: string name of a *digital_outputs* device. Defaults to empty.

This output is used to clock the SPI chip.

cs_pin:

Single value, type: string name of a *digital_outputs* device. Defaults to empty.

This output is used to chip select the SPI chip. It usually also triggers the parallel read of the chip.

miso_pin:

Single value, type: string name of a *switches* device. Defaults to empty.

This input is read serially to determine the state of your inputs.

Optional settings

The following sections are optional in the `spi_bit_bang:` section of your config. (If you don't include them, the default will be used).

bit_time:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `50ms`

How long should the platform wait until reading the `miso_pin`. Depending on your platform it might need a while to settle. Especially if your platform is connected via USB. If your inputs are local (i.e. on a RPi) this might be very short compared.

clock_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `1ms`

How long should the clock pulse be? 1ms is the lower limit for most platforms and more than long enough for any chip so this should be good.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set to true to get more debug output.

inputs:

Single value, type: `integer`. Default: `8`

How many inputs should the platform read? Reading less inputs will result in faster updates.

Related How To guides

- *How to use SPI Bit Bang in MPF*
- *Using the Stern Spike Trough*

spike:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `spike:` section of your machine-wide config is where you configure hardware options that are specific to the SPIKE interface when you're using MPF with a Stern SPIKE machine. Note that we have a how to guide which includes *all the SPIKE-specific settings* throughout your entire config file, so be sure to read that if you have a SPIKE machine.

```
hardware:
  platform: spike
spike:
  port: /dev/ttyUSB0
  baud: 115200
  runtime_baud: 3000000
  flow_control: true
  debug: false
  nodes: 0, 1, 8, 9, 10, 11
```

Required settings

The following sections are required in the `spike:` section of your config:

baud:

Single value, type: `integer`. Defaults to empty.

This needs to match the value from Step 3 in the MPF SPIKE bridge instructions.

nodes:

List of one (or more) values, each is a type: `integer`. Defaults to empty.

Configure the nodes from your manual. Note that there should always be a node 0 and 1.

port:

Single value, type: `string`. Defaults to empty.

on the RPi.

Optional settings

The following sections are optional in the `spike:` section of your config. (If you don't include them, the default will be used).

bridge_debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set to True if you want to debug your MPF Spike bridge.

bridge_debug_log:

Single value, type: `string`. Default: `/mnt/spike.log`

Path on your Spike system where the bridge logs to if `bridge_debug` is `True`. Needs to be writable and sufficiently large. A USB stick mounted to `/mnt/` will work fine.

bridge_path:

Single value, type: `string`. Default: `/bin/bridge`

Path of your bridge.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `none`

Log level to console.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set to true for troubleshooting to print more details in the log.

default_debounce_close:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `4`

Default debounce close time.

default_debounce_open:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `4`

Default debounce open time.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level to file.

flow_control:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set to `True` to enable serial RTS/CTS flow control between MPF and the Spike bridge. May help improve responsiveness and reduce latency when streaming display data to the DMD. Default is `False`.

max_led_batch_size:

Single value, type: `integer`. Default: `6`

Maximum number of leds to batch. This seems to differ between machines. `3` seems to be safe everywhere.

node_config:

One or more sub-entries. Each in the format of `integer` : *spike_node*

A list of your nodes with their config each. This is entirely optional but may improve performance.

oc_time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `100`

Some time related to over current. We believe this is the time over which spike averages the value.

periodically_query_nodes:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Whether to periodically query nodes. The spike game does this but we do not use the values so it is probably save to disable this. Related to over current detection.

poll_hz:

Single value, type: `integer`. Default: `1000`

Numeric value of how many times per second MPF will poll the SPIKE system to check for switch changes. Default is `1000`.

response_time:

Single value, type: `integer`. Default: `837`

A parameter send to the spike bus driver. We believe this is some kind of bus timeout. No need to change it.

runtime_baud:

Single value, type: `integer`. Default: `921600`

Baud rate to use during runtime.

spike_version:

Single value, type: one of the following options: 1, 2. Default: `1`

The spike version you are using.

use_send_key:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Send some magic commands like Spike does. Not needed as far as we know.

verify_checksums_on_readback:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Whether to verify checksums on readback from commands. This should be always on unless you are debugging something.

wait_times:

One or more sub-entries. Each in the format of `integer` : `integer`

A list of commands and their corresponding wait times on the bus. Ususally, you do not have to change this.

Related How To guides

- *How to use MPF with Stern SPIKE / SPIKE 2 machines*

spike_node:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

The `node_config:` section of your config is where you configure your node boards in your `spike:` section.

Optional settings

The following sections are optional in the `spike_node:` section of your config. (If you don't include them, the default will be used).

coil_priorities:

List of one (or more) values, each is a type: `integer`.

A list of coils ordered by priority. This list is send to the hardware to priorize coils when multiple hardware rules active. The exact logic is unknown.

num_inputs:

Single value, type: `integer`.

Number of inputs on that node board.

num_leds:

Single value, type: `integer`.

Number of LEDs on that node board.

Related How To guides

- *How to use MPF with Stern SPIKE / SPIKE 2 machines*

state_machines:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `state_machines:` section of your config is where you configure generic *state machines*.

Settings in Machine Config Files

If the `state_machines:` section is placed in a config file, it will retain its state across games. When the game is started, the value is initialized, and it will retain in its state until the game is turned off. So to reset this, a transition would need to happen upon game end.

Settings in Mode Config Files

If the `state_machines:` section is placed in a mode file, it will retain its state across balls, but will be reset to its base mode for each game. It is player specific, and will retain the correct value fo each player in a given game.

Required settings

The following sections are required in the state_machines: section of your config:

states:

One or more sub-entries. Each in the format of string : *state_machine_states*

List all of your states here, with their applicable settings. Go to *state_machine_states* to see a full list of all settings under states:. For example:

```
##! mode: my_mode
state_machines:
  my_state:
    states:
      start:
        label: Start state
      step1:
        label:
        show_when_active:
          show: on
          show_tokens: None
        events_when_started: step1_start
        events_when_stopped: step1_stop
      step2:
        label: Step 2
    transitions:
```

The first state must be start: or MPF will throw errors when trying to initialize this value (you can change this using starting_state setting). All other states can be any string as defined by the user.

transitions:

List of one (or more) values, each is a type: *state_machine_transitions*. Defaults to empty.

These move from any state to another state, including backward or back to the first step, when a given event is posted.

List all your transitions here (we start with the same steps as above):

```
##! mode: my_mode
state_machines:
  my_state:
    states:
      start:
        label: Start state
      step1:
        label:
        show_when_active:
          show: on
          show_tokens: None
        events_when_started: step1_start
        events_when_stopped: step1_stop
```

(continues on next page)

(continued from previous page)

```
    step2:
      label: Step2
  transitions:
    - source: start
      target: step1
      events: state_machine_proceed
    - source: step1
      target: step2
      events: state_machine_proceed2
      events_when_transitioning: going_to_step2
    - source: step2
      target: start
      events: state_machine_proceed3
    - source: step1, step2
      target: start
      events: state_machine_reset
```

Optional settings

The following sections are optional in the `state_machines:` section of your config. (If you don't include them, the default will be used).

persist_state:

Single value, type: `boolean` (`true`/`false`). Default: `false`

If set to true MPF will restore the state of a logic_block on mode restart.

starting_state:

Single value, type: `string`. Default: `start`

The start state of your state machine.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Not used.

state_machine_transitions:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

The `state_machine_transitions:` section of your config is where you configure the transitions of your *state machine*.

Transitions will only be available if the state machine is in one of the states listed in `source`. In that case the machine will transition to the state listed in `target`. See *state machines* for details.

Required settings

The following sections are required in the `state_machine_transitions:` section of your config:

events:

List of one (or more) events.

If the state machine is in one of the states listed in `source` this event will transition the machine to the state listed in `target`.

source:

List of one (or more) values, each is a type: `string`.

Transitions will only be available if the state machine is in one of the states listed in `source`.

target:

Single value, type: `string`.

The machine will transition to this state if it is in a state listed in `source` and one of the `events` is posted.

Optional settings

The following sections are optional in the `state_machine_transitions:` section of your config. (If you don't include them, the default will be used).

events_when_transitioning:

List of one (or more) events.

This event will be posted when the transition is triggered.

Related How To guides

- *State Machine Logic Block*

state_machine_states:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

The `state_machine_states:` section of your config is where you configure the states of your *state machine*.

See *state machines* for details.

Optional settings

The following sections are optional in the `state_machine_states:` section of your config. (If you don't include them, the default will be used).

events_when_started:

List of one (or more) events.

The event will be posted when the state machine enters this state. This is the entry action for this state in your finite state machine.

events_when_stopped:

List of one (or more) events.

The event will be posted when the state machine leaves this state. This is the exit action for this state in your finite state machine.

label:

Single value, type: `string`.

The full name/description of this state.

show_when_active:

Single value, type: *show_config*.

A show which is played when the state machine is in this state. This is kind of an entry action as you could use `events_when_started` and a *show_player:* to achieve the same. It is meant as a helper because it is common to play one show per step.

Related How To guides

- *State Machine Logic Block*

Related How To guides

- *State Machine Logic Block*
- *Skill Shot*

steppers:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `steppers:` section of your config is where you configure steppers.

This is an example:

```
# main config
p_roc:
  use_separate_thread: true
  pd_led_boards:
    6:
      use_stepper_0: true
      stepper_speed: 1352400000  # Determine empiricall. Increasing slows pulsesrate
```

(continues on next page)

(continued from previous page)

```
switches:
  s_stepper_home:
    number: 4/0/5
steppers:
  ramp_diverter:
    number: 6-0
    homing_mode: switch
    homing_switch: s_stepper_home
    homing_direction: clockwise
    pos_min: 0 # Default. (Neg values are behind home)
    pos_max: 100 # Default
    reset_events: machine_reset_phase_3, ball_starting, ball_will_end
    reset_position: 0 # Default
    debug: true
    named_positions:
      2: move_to_2
      25: move_to_25
      45: move_to_45
##! mode: base
# base mode
timers:
  test_diverter:
    start_value: 0
    end_value: 6
    start_running: true
    restart_on_complete: true
event_player:
  timer_test_diverter_tick{device.timers.test_diverter.ticks==1}: move_to_2
  timer_test_diverter_tick{device.timers.test_diverter.ticks==3}: move_to_25
  timer_test_diverter_tick{device.timers.test_diverter.ticks==5}: move_to_45
```

Required settings

The following sections are required in the `steppers:` section of your config:

number:

Single value, type: `string`. Defaults to empty.

This is the number of the stepper which specifies which stepper the it is physically connected to. The exact format used here will depend on which control system you're using and how the stepper is connected.

See the *How to configure "number:" settings* guide for details.

Optional settings

The following sections are optional in the `steppers:` section of your config. (If you don't include them, the default will be used).

ball_search_max:

Single value, type: `integer`. Default: `1`

The maximum position to use during ball search for this stepper. During ball search the stepper will move between `ball_search_min` and `ball_search_max`.

ball_search_min:

Single value, type: `integer`. Default: `0`

The minimum position to use during ball search for this stepper. During ball search the stepper will move between `ball_search_min` and `ball_search_max`.

ball_search_wait:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `5s`

How long should the stepper wait after moving to `ball_search_min` before moving to `ball_search_max`.

homing_direction:

Single value, type: one of the following options: clockwise, counterclockwise. Default: `clockwise`

In which direction should the stepper move to reach the home position?

homing_mode:

Single value, type: one of the following options: hardware, switch. Default: `hardware`

Some controllers support `hardware` homing which should be preferred. However, you can also define a `homing_switch` which will be used to determine whether the stepper is at the home position.

homing_switch:

Single value, type: string name of a *switches* device. Defaults to empty.

Switch to check if the stepper is at the home position when `homing_mode` is set to `switch`.

include_in_ball_search:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Set to true to enable ball search on this stepper.

named_positions:

One or more sub-entries. Each in the format of number (will be converted to floating point) : string

This is a sub-section mapping of stepper positions to MPF event names. For example:

```
named_positions:
  0: move_home
  999: move_to_999
  -500: move_to_-500 # Negative positions are behind home
```

The values in this named_positions: list represent MPF events that, when posted, tell this stepper to move to a certain position. So in the example above, when the *move_to_999* event is posted, this stepper will move to position 999.

platform:

Single value, type: string. Defaults to empty.

Name of the platform this stepper is connected to. The default value of None means the default hardware platform will be used. You only need to change this if you have multiple different hardware platforms in use and this stepper is not connected to the default platform.

See the *Mixing-and-Matching hardware platforms* guide for details.

platform_settings:

Single value, type: dict. Defaults to empty.

Platform specific stepper settings for this stepper. Check the *documentation of your platform* for details.

pos_max:

Single value, type: integer. Default: 1000

Maximum possible position.

pos_min:

Single value, type: integer. Default: 0

Minimum possible position. Negative values are left of the home position.

reset_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: machine_reset_phase_3, ball_starting, ball_will_end, service_mode_entered

Events to reset the position of the stepper.

reset_position:

Single value, type: `integer`. Default: `0`

Reset position for this stepper. Usually this is the home position.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Not used currently.

Related How To guides

- *Stepper Motors*

step_stick_stepper_settings:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

The `step_stick_stepper_settings:` section of your config is where you configure the *Stepstick hardware platform*.

Optional settings

The following sections are optional in the `step_stick_stepper_settings:` section of your config. (If you don't include them, the default will be used).

high_time:

Single value, type: `time string (secs)` (*Instructions for entering time strings*) . Default: `20ms`

How long should the digital output be held to high during a step pulse? This time depends on the latency/jitter of your output and the speed your stepper can be moved. Usually the jitter of your output is the limiting factor.

low_time:

Single value, type: `time string (secs)` (*Instructions for entering time strings*) . Default: `20ms`

How long should the digital output be held to low after a step pulse? This time depends on the latency/jitter of your output and the speed your stepper can be moved. Usually the jitter of your output is the limiting factor.

switch_overwrites:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

Some devices offer a `switch_overwrites:` setting where you can overwrite settings of a switch used in that devices. This is commonly used in *flippers:* and *autofire_coils:*.

Optional settings

The following sections are optional in the `switch_overwrites:` section of your config. (If you don't include them, the default will be used).

debounce:

Single value, type: one of the following options: quick, normal, None. Default: `None`

Overwrite the `debounce` setting on a coil. See `debounce` in *switches:* for details.

switch_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `switch_player:` section of your config is where you can replay a series of switches for testing purposes. Also have a look at the *MPF monitor* for interactive testing purposes.

This is an example:

```
#config_version=5
switches:
  s_test1:
    number:
    x: 0.4
    y: 0.5
    z: 0
  s_test2:
    number:
    x: 0.6
    y: 0.7
  s_test3:
    number:
plugins: switch_player
switch_player:
  start_event: test_start
  steps:
    - time: 100ms
      switch: s_test1
      action: activate
    - time: 600ms
      switch: s_test3
      action: hit
    - time: 100ms
      switch: s_test1
      action: deactivate
    - time: 1s
      switch: s_test2
      action: activate
    - time: 1s
      switch: s_test3
      action: hit
    - time: 100ms
      switch: s_test2
      action: deactivate
    - time: 1s
      switch: s_test3
      action: hit
```

Optional settings

The following sections are optional in the `switch_player:` section of your config. (If you don't include them, the default will be used).

start_event:

Single event. The device will add an handler for this event. Default: `machine_reset_phase_3`

Event to trigger the start of the switch player.

steps:

Unknown type. See description below.

The steps of the switch_player. See the example above.

Related How To guides

- *plugins:*

switches:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The *switches:* section of the config files is used to map switch names to controller board inputs. You can map both direct and matrix switches. Here's an example section:

```
switches:
  flipper_lwr_eos:
    number: SF1
  flipper_lwr:
    number: SF6
  fire_r:
    number: S12
    tags: plunger
  start:
    number: S13
    tags: start
  plumbbob:
    number: S14
    tags: tilt
  outlane_l:
    number: S16
    tags: playfield_active
    debounce: normal
  inlane_l:
    number: S17
    tags: playfield_active
    debounce: quick
  trough1:
    number: S81
    type: 'NC'
  shooter_lane:
    number: S82
    events_when_activated: ball_in
    events_when_deactivated: ball_out
```

Each subsection of `switches:` is a switch name, which is how you refer to the switch in your game code. A fully working example for the Cobra board can be found in *OPP Switches*, that example might be as well helpful when using other hardware to understand what events are being fired when using a switch.

When configuring switches, then there are several parameters for each switch:

Required settings

The following sections are required in the `switches:` section of your config:

number:

Single value, type: `string`. Defaults to empty.

This is the number of the switch which specifies which switch input the switch is physically connected to. The exact format used here will depend on which control system you're using and how the switch is connected.

Note: In a virtual environment with *keyboard:* section you don't have to fill in a switch number. With a keyboard section the switch is activated by a defined keyboards key.

See the *How to configure "number:" settings* guide for details.

Optional settings

The following sections are optional in the `switches:` section of your config. (If you don't include them, the default will be used).

debounce:

Single value, type: one of the following options: auto, quick, normal. Default: `auto`

The debounce setting to use in hardware. `quick` means very low to no debounce (could also be named "off"). `normal` implies debounce "on" and should be used in most cases. The exact timings of those settings depend on your hardware platform. (`quick` usually is 0-1ms, `normal` is 1-4ms).

The main purpose of this is to reduce the number of events/amount of communication from the hardware. For targets and swiches in debounce `normal` should be good in almost all cases.

However, in some cases, you want to disable debounce (e.g. use `quick`) when using *hardware rules* such as pop bumpers or sling shots. `auto` will use `normal` if no hardware rules are configured or `quick` when rules are configured. Therefore, you usually can leave this at `auto`.

Switch debouncing is somewhat different from debouncing in other domains since the switch has to be active for the whole period of debouncing (at least during sampling). It could also be referred as "minimum activation time" (as one discipline of debouncing). If you want to make sure that the switch does not activate again within a certain period have a look at `ignore_window_ms` (another discipline of debouncing). If you want to control the fire rate of your *coil* have a look at the `recycle` setting (configurable in some platforms).

See *Debouncing in Pinball Machines* for details.

events_when_activated:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A list of one or more names of events that MPF will post when this switch goes active. These events are posted exactly as they're entered, in addition to the events that are posted based on the switch's tags. See as well the `tags` section below. In addition, an event will be posted based on the switch name, `<switch name>_active`.

The events will only be visible in the `mpf monitor` if they are consumed by something, e.g. `light_player` or if `debug:true` is defined for them. They will be posted regardless of the `debug` setting, it is only a question of visibility in the `mpf monitor`.

events_when_deactivated:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A list of one or more names of events that MPF will post when this switch goes inactive. These events are posted exactly as they're entered, in addition to the events that are posted based on the switch's tags. See as well the `tags` section below. In addition, an event will be posted based on the switch name, `<switch name>_inactive`.

The events will only be visible in the `mpf monitor` if they are consumed by something, e.g. `light_player` or if `debug:true` is defined for them. They will be posted regardless of the `debug` setting, it is only a question of visibility in the `mpf monitor`.

ignore_window_ms:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `0`

Specifies a duration of time during which additional switch activations will be ignored.

For example, if you set `ignore_window_ms: 100`, then a switch is activated once, then again 50ms later, the second activation will be ignored. The timer is set based on the last switch hit that *activated* the switch, so if another switch hit came in 105ms after the first (which would be 55ms after the second), it will also count.

platform:

Single value, type: `string`. Defaults to empty.

Name of the platform this switch is connected to. The default value of `None` means the default hardware platform will be used. You only need to change this if you have multiple different hardware platforms in use and this switch is not connected to the default platform.

See the *Mixing-and-Matching hardware platforms* guide for details.

platform_settings:

Single value, type: dict. Defaults to empty.

Dict of platform specific settings. See your *platform documentation* about this.

type:

Single value, type: one of the following options: NC, NO. Default: `NO`

You can add `NC` as a type (like `type: NC`) to indicate that this switch is a normally closed switch, i.e. it's closed when it's inactive and open when it's active. This is mostly used for optos.

Switches which are type NC are automatically inverted by the Switch Controller. In other words an NC switch is still "active" when it's being activated, but the Switch Controller knows that activation actually occurs when the switch opens, rather than closes. Setting the type to NC here means that you never have to worry about this inversion anywhere else in your game code.

x:

Single value, type: `number` (will be converted to floating point). Defaults to empty.

X Position of this switch on the playfield. Currently unused.

y:

Single value, type: `number` (will be converted to floating point). Defaults to empty.

Y Position of this switch on the playfield. Currently unused.

z:

Single value, type: `number` (will be converted to floating point). Defaults to empty.

Z Position of this switch on the playfield. Currently unused.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to get additional debug output. You need to set this flag to see event you have defined for this switch in mpf monitor.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this switch in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

You can add tags to switches to logically group them in your game code to make it easier to do things. (Like "if all the switches tagged with `droptarget_bank1` are active, then do something.") Tags are also used to create MPF events which are automatically posted with an `sw_` prefix, by tag, when a switch is activated. For example, if you have a switch tagged with "hello", then every time that switch is activated, it will post the event `sw_hello`. If you have a switch tagged with "hello" and "yo", then every time that switch is activated it will post the events `sw_hello` and `sw_yo`. MPF also makes use of several tags on its own.

In addition, events will be posted based on the switch name, `<switch name>_active` and `<switch name>_inactive`.

The events will only be visible in the `mpf monitor` if they are consumed by something, e.g. `light_player` or if `debug:true` is defined for them. They will be posted regardless of the `debug` setting, it is only a question of visibility in the `mpf monitor`.

Special-purpose tags for switches include:

- `playfield_active` - This tag should be used for all switches on the playfield that indicate a ball is loose on the playfield. This tag is used by the playfield to know that balls are on it. Note that if you have more than one playfield, the tag name is (playfield_name)_active, so if you have a playfield called "upper playfield", you'd tag the switches on that playfield with "upper_playfield_active".
- `start` - Let's MPF know that this switch is used to start a game. (Note that in MPF, the game start process is kicked off when this switch is released, not pressed, which allows the "time held down" to be sent to MPF to perform alternate game start actions.)

Related How To guides

- *Switches*
- *How to configure opto switches*
- *Mechanical Switches*

system11:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `system11:` section of your config is where your system11 machine. This is usually used together with the *snux platform* or *apc platform*.

Required settings

The following sections are required in the `system11:` section of your config:

ac_relay_driver:

Single value, type: string name of a *coils* device. Defaults to empty.

The driver to use to drive the AC relay which switches between A and C side drivers.

Optional settings

The following sections are optional in the `system11:` section of your config. (If you don't include them, the default will be used).

ac_relay_debounce_ms:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `0`

Todo: *Help us to write it*

ac_relay_delay_ms:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `75ms`

Delay when switching between A and C side.

ac_relay_switch:

Single value, type: string name of a *switches* device. Defaults to empty.

Todo: *Help us to write it*

console_log:

Single value, type: one of the following options: none, basic, full. Default: `none`

Log level for the console log for this platform.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Todo: *Help us to write it*

---

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this platform.

platform:

Single value, type: `string`. Defaults to empty.

Upstream platform for hardware. System 11 is a virtual platform which drives coils on another underlying platform which can be configured here.

prefer_a_side_event:

Single event. The device will add an handler for this event. Default: `game_ended`

Event to trigger A-side preference. This is triggered at game end by default to reduce stress on the AC-relay during attract.

prefer_c_side_event:

Single event. The device will add an handler for this event. Default: `game_will_start`

Event to trigger C-side preference. This is triggered at game start by default to increase response times.

queue_c_side_while_preferred:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Todo: *Help us to write it*

Related How To guides

- *Snux System 11 Driver Board*
- *Arduino Pinball Controller*

text_strings:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `text_strings:` section of your config is where you define text strings which can be used in slides or widgets.

This is an example:

```
text_strings:
  greeting: HELLO PLAYER. THIS IS YOUR BALL (ball)
slides:
  slides_with_text:
    - type: text
      text: $greeting
```

Related How To guides

- *Text Widget*

text_ui:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `text_ui:` section of your config is where you configure the Text UI that appears in the console while MPF is running.

The Text UI displays information about the machine and game: switch states, active modes, variable values, and game information. By default, it displays all machine variables and player variables.

Depending on the complexity of your game and the mode you're working on, you may not want the Text UI to display *every* variable. In that case, you can use the `text_ui:` section to specify which player and machine variables you want to see.

Optional settings

The following sections are optional in the `text_ui:` section of your config. (If you don't include them, the default will be used).

machine_vars:

List of one (or more) values, each is a type: `string`. Defaults to empty.

A list of all of the machine variables to display and update in the Text UI. If the list is empty, no machine variables will be displayed.

If the `machine_vars:` setting is not included in your config, *all* machine variables will be displayed.

player_vars:

List of one (or more) values, each is a type: `string`. Defaults to empty.

A list of all of the player variables to display and update in the Text UI.

While a game is active, MPF will always show three player variables: player number, ball number, and player score. If the `player_vars:` setting is provided, the variable names listed will also be shown in the Text UI.

If the `player_vars:` setting is not included in your config, *all* player variables will be displayed.

Related How To guides

Todo: *Help us to write it*

tic_stepper_settings:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

If you use the *Pololu Tic Stepper Controller* you can use the following settings in `platform_settings` of your steppers.

Optional settings

The following sections are optional in the `tic_stepper_settings:` section of your config. (If you don't include them, the default will be used).

current_limit:

Single value, type: `integer`. Default: `192`

Todo: *Help us to write it*

max_acceleration:

Single value, type: `integer`. Default: `40000`

Todo: *Help us to write it*

max_deceleration:

Single value, type: `integer`. Default: `40000`

Todo: *Help us to write it*

max_speed:

Single value, type: `integer`. Default: `2000000`

Todo: *Help us to write it*

poll_ms:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Default: `100ms`

How often should MPF poll the state of your steppers? This is used to check for completion of movements. There should be no need to modify this.

starting_speed:

Single value, type: `integer`. Default: `0`

Todo: *Help us to write it*

step_mode:

Single value, type: `integer`. Default: `1`

Todo: *Help us to write it*

Related How To guides

- *How to use Pololu Tic in MPF*

tilt:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**YES**

The `tilt:` section of your config is where you configure a tilt mode.

Optional settings

The following sections are optional in the `tilt:` section of your config. (If you don't include them, the default will be used).

multiple_hit_window:

Single value, type: `time string (ms) or template` (*Instructions for entering time strings* and *Instructions for entering templates*). Default: `300ms`

Window in which hits are ignored after a tilt hit.

reset_warnings_events:

List of one (or more) device control events (*Instructions for entering device control events*). Default: `ball_will_end`

Default: `ball_will_end`

These events, when posted, will cause the `warnings_to_tilt:` to be reset to zero.

settle_time:

Single value, type: `time string (ms) or template` (*Instructions for entering time strings* and *Instructions for entering templates*). Default: `5s`

Time to wait after the machine is tilted before a new ball can be started. This prevents that a player can tilt his ball and the first ball of the next player.

slam_tilt_switch_tag:

Single value, type: `string`. Default: `slam_tilt`

Switch tags which will cause a slam tilt.

tilt_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Events in this list, when posted, cause a *tilt* to occur which will end the current ball in progress with no end of ball bonus. You usually want to use tilt_warning_events because this one will instantly tilt the machine on the first event.

tilt_slam_tilt_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Events in this list, when posted, cause a *slam_tilt* event to be posted. The slam tilt typically ends the current game and also clears all credits from the machine.

tilt_switch_tag:

Single value, type: `string`. Default: `tilt`

Switch tag for switches which cause the machine to tilt (without prior warnigns). You want to use the tag configured in `tilt_warning_switch_tag` in most cases.

tilt_warning_events:

List of one (or more) device control events (*Instructions for entering device control events*). Defaults to empty.

Default: `None`

Events in this list, when posted, cause a tilt warning to occur. They will post the *tilt_warning* event, and if the `warnings_to_tilt:` limit is hit, will also cause the *tilt* event.

tilt_warning_switch_tag:

Single value, type: `string`. Default: `tilt_warning`

Switch tags for switches which cause a tilt warning.

tilt_warnings_player_var:

Single value, type: `string`. Default: `tilt_warnings`

Player var to use to store tilt warnings.

warnings_to_tilt:

Single value, type: `integer` or `template` (*Instructions for entering templates*). Default: `3`

Number of warnings until the machine tilts.

Also note that you can use *dynamic values* here if you want to do math or use settings to make this configurable.

Related How To guides

- *Tilt*

timed_switches:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

Specifies *timed switches* which are used to post events when a switch is active for a continuous amount of time.

Here's an example. This example is actually built-in to MPF via the MPF default config file, so if you want to use these flipper cradle events, you don't have to enter them yourself as they're already there.

```
timed_switches:
  flipper_cradle:
    switch_tags: left_flipper, right_flipper
    time: 3s
    events_when_active: flipper_cradle
    events_when_released: flipper_cradle_release
```

Like other devices in MPF, the format is:

```
timed_switches:
   name_of_your_timed_switch:
      <settings>
   some_other_timed_switch:
      <settings>
```

Required settings

The following sections are required in the `timed_switches:` section of your config:

time:

Single value, type: `time string (ms)` (*Instructions for entering time strings*). Defaults to empty.

How long a switch must be continuously active before the `events_when_active` are posted.

Optional settings

The following sections are optional in the `timed_switches:` section of your config. (If you don't include them, the default will be used).

events_when_active:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

If you don't enter any events here, an event will automatically be posted in the format *<name_of_this_timed_switch>_active*. In other words, in the example at the top of this page, the timed switch entry is called "flipper_cradle", so the automatically-created event would be called *flipper_cradle_active*, but since that config has an `events_when_active: flipper_cradle` entry, then the event will just be *flipper_cradle*.

events_when_released:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

If you've defined multiple switches and two switches go active, the release event will not be posted until all the switches are released.

state:

Single value, type: one of the following options: active, inactive. Default: `active`

Controls whether the `events_when_active:` are posted when the switch is active for the `time:` amount, or whether it's flipped and the events are posted when the switch is `inactive` for the time amount.

switch_tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

A list of switch tags (or a single tag) that will be used to set which switches are used with these timed switch settings. Each switch with these tags will be added.

switches:

List of one (or more) values, each is a type: string name of a *switches* device. Defaults to empty.

A list of switches (or a single switch) that will be used for these timed switch settings. Note that you can use `switch_tags:` instead of `switches:`.

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Set this to true to see additional debug output. This might impact the performance of MPF.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Not used.

Related How To guides

- *Timed Switches*

timers:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**YES**

The `timers:` section of your config is where configure *timers* that can "tick" up or down. Timers post events with each tick which you can use to update slides, etc. You can set the start and stop values of the timers, as well as how fast they tick, how much they change per tick, and other settings.

The settings structure of timers is like this:

```
timers:
   timer_name:
      <settings>
   some_other_timer_with_a_different_name:
      <settings>
   a_third_timer:
      <settings>
```

Here's an example timers: section from the "Money Bags" mode in *Brooks 'n Dunn* which contains two timers:

```
##! mode: mode1
timers:
  mb_intro_timer:
    start_value: 3
    end_value: 0
    direction: down
    control_events:
      - action: start
        event: mode_money_bags_started
  money_bags_timer:
    start_value: 15
    end_value: 0
    direction: down
    tick_interval: 1.25s
    control_events:
      - action: start
        event: timer_mb_intro_timer_complete
      - action: add
        event: money_bags_advertise_flashing_hit
        value: 5
      - action: stop
        event: logicblock_money_bags_counter_complete
```

In the example above, an intro timer which runs for 3 seconds is started by the event *mode_money_bags_started* (which means this timer starts when the mode starts). A second timer (the "money_bags_timer") starts when the intro timer is complete. It starts with a value of 15 and counts down to 0 (but at a count interval of 1.25 seconds so it's a bit slower than real time. It will also get reset back to 15 each time a flashing shot is hit.

Here's another example of timers from *Demo Man's* skillshot mode:

```
##! mode: mode1
timers:
  mode_timer:
    start_value: 3
    end_value: 0
    direction: down
    tick_interval: 1s
    control_events:
      - event: balldevice_playfield_ball_enter
        action: start
    start_running: false
  target_rotator:
    start_running: true
    tick_interval: 1s
```

The skillshot mode starts when the ball is waiting to be plunged. The timer called "mode_timer" in the example above starts when the ball enters the playfield and runs for 3 seconds. If it runs all the way down, the skill shot mode will stop (meaning the player missed the skillshot).

A second timer doesn't have any count values associated with it, rather it just "ticks" once a second. That tick event is used to rotate the lit skillshot.

See *timer_control_events:* for more details about all the actions available in a timer.

Optional settings

The following sections are optional in the `timers:` section of your config. (If you don't include them, the default will be used).

bcp:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Controls whether the various timer events (count, start, stop, complete, etc.) are sent to the MPF-MC via BCP.

control_events:

List of one (or more) values, each is a type: *timer_control_events*. Defaults to empty.

Timer control events is where you specify what happens to this timer when other events are posted. See *timer_control_events:* for more details.

direction:

Single value, type: one of the following options: up, down. Default: `up`

Controls which direction this timer runs in. Options are `up` or `down`.

end_value:

Single value, type: `integer` or `template` (*Instructions for entering templates*). Defaults to empty.

Specifies what the final value for this timer will be. When the timer value equals or exceeds this (for timers counting up), or when it equals or is lower than this (for timers counting down), the *timer_<name>_complete* event is posted and the timer is stopped. (If the `restart_on_complete:` setting is true, then the timer is also reset back to its `start_value:` and started again.)

Note that you can use a *dynamic value* for this setting.

max_value:

Single value, type: `integer`. Defaults to empty.

The maximum value this timer can be. If you try to add value above this, the timer's value will be reset to this value.

restart_on_complete:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Controls what should happen when this timer completes. If you have `restart_on_complete: true`, then this timer will reset back to the start_value and start again after it completes.

start_running:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Controls whether this timer starts running ("started"), or whether it needs to be started with one of the start control events.

start_value:

Single value, type: `integer` or `template` (*Instructions for entering templates*). Default: `0`

The initial value of the timer. When a timer is restarted, this is the value it will always start from. If you ever need to change the value, you can use a jump control action to set it to whatever value you want. See *timer_control_events:* for more details.

Note that you can use a *dynamic value* for this setting.

tick_interval:

Single value, type: `time string (secs)` or `template` (*Instructions for entering time strings* and *Instructions for entering templates*). Default: `1s`

A time value for how fast each tick is. The default is 1 second, but quite often "pinball time" is slower than real world time, and a countdown timer will actually tick a speed that's slower than 1 second per tick. (So in that case, you might set `tick_interval: 1.25s` or something like that. You can also set this really short if you want a hurry up, maybe every 100ms removed 77,000 worth of points or something.

Also note that you can use *dynamic values* here if you want to do math or use settings to make this configurable.

timer_control_events:

Config file section

Valid in *machine config files*	**NO**
Valid in *mode config files*	**NO**

The `timer_control_events:` section of your config is where you configure control events for your *timer*.

They're entered as a list (with dashes) under the `control_events:` section. All control events have an `event:` and `action:` setting. (When the "event" is posted, the "action" is taken. Some actions require an additional `value:` setting. For example, for the "add" action which adds time, you need to to specify how much time you want to add. But other actions, like "start" or "stop" don't need values.

Here's an example of control events in action:

```
##! mode: mode1
timers:
  my_timer:
    direction: down
    start_value: 10
    tick_interval: 125s
```

(continues on next page)

(continued from previous page)

```
  control_events:
    - event: start_my_timer
      action: start
    - event: reset_my_timer
      action: reset
    - event: add_5_secs
      action: add
      value: 5
```

In the example above, when the event *start_my_timer* is posted, the timer called "my_timer" will start running. When the event *add_5_secs* is posted, 5 seconds will be added to whatever the current value of "my_timer" is, etc.

Required settings

The following sections are required in the `timer_control_events:` section of your config:

action:

Single value, type: one of the following options: add, subtract, jump, start, stop, reset, restart, pause, set_tick_interval, change_tick_interval, reset_tick_interval.

Take a look at the various types of actions you can perform on timers with control events:

add Adds the time (specified in the `value:` setting) to the timer. If the value would be higher than the timer's `max_value:` setting, then the value is set to the max value. Posts the *timer_<name>_time_added* event.

This action does not change the timer's running state.

The timer is checked for done after the value has been added. (So, for example, if you have a timer that's set to count up, and the timer finishes at 10, and the timer is currently at 6, and you add value of 5, then the timer will be complete.

subtract Subtracts time (specified in the `value:` setting) from the timer. Posts the *timer_<name>_time_subtracted* event and checks to see if the timer is complete.

jump "Jumps" the timer to a specific new value (specified in the `value:` setting) and checks to see if the timer is complete.

start Starts the timer if it's not running. Does nothing if the timer is already running. Posts the *timer_<name>_started* event.

stop Stops the timer and posts the *timer_<name>_stopped* event. Removes any outstanding "pause" delays.

reset Changes the timers current value back to the `start_value:`. Nothing else is touched, so if the timer is running, it stays running, etc.

restart Acts as a combination of reset, then start.

pause Pauses the timer for a given `value:` time (in seconds). Note that the timer pause value is real world seconds and does not take the timers tick interval into consideration. If the pause value is 0, the timer is paused indefinitely. Posts the *timer_<name>_paused* event.

set_tick_interval Sets the tick interval to a new value (specified in the `value:` setting).

change_tick_interval Changes the tick interval by multiplying the current tick interval by the new one specified in the `value:` setting. In other words, if you want to make the tick interval 10% faster, than set this to `value: 1.1`. If you want to make it 50% slower, set this to `value: 0.5`, etc.

reset_tick_interval (added in MPF 0.33)

Resets the timer's tick interval back to the original from the `tick_interval:` setting.

event:

Single value, type: `string`.

The event which will trigger this value.

Optional settings

The following sections are optional in the `timer_control_events:` section of your config. (If you don't include them, the default will be used).

value:

Single value, type: `number` or `template` (will be converted to floating point; *Instructions for entering templates*).

The value for this `action`. Not all actions require a value (i.e. `start` and `stop` do not). You can use placeholders here to calculate it during runtime.

Related How To guides

- *Timers*

console_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the console log for this device.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

If true/yes, adds additional logging information to the verbose log for this timer.

file_log:

Single value, type: one of the following options: none, basic, full. Default: `basic`

Log level for the file log for this device.

label:

Single value, type: `string`. Default: `%`

Name of this device in service mode.

tags:

List of one (or more) values, each is a type: `string`. Defaults to empty.

Not used.

Related How To guides

- *Timers*

track_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

Note: This section can also be used in a show file in the `tracks:` section of a step.

The `track_player:` section of your config is where you specify actions to perform on audio tracks when MPF events are received. Tracks can be stopped, paused, or played with an optional fade time. The volume of a track can also be changed with an optional fade time. Finally, all sounds currently playing on a track can be stopped (again with an optional fade out time). (This player is part of the MPF media controller and only available if you're using MPF-MC for your media controller.)

This is an example:

```
track_player:
  pause_music_track:
    music:
      action: pause
      fade: 1 sec
  resume_music_track:
    music:
      action: play
  stop_sounds_on_all_tracks:
    __all__:
      action: stop_all_sounds
      fade: 0.5 sec
```

See the *config player* for more information on config players.

Express configuration

There is no express (one line) configuration for the track player. You must specify the `action` setting every time.

Required settings

The following sections are required in the `track_player:` section of your config:

action:

Single value, type: one of the following options: play, stop, pause, set_volume, stop_all_sounds. Defaults to empty.

The `action:` setting controls what action will be performed on the specified track. Options for `action:` are:

- `play` - The specified track will be played after it has been stopped or paused.
- `stop` - The track is stopped (with an optional fade out time). All sound processing on the track is stopped and the track is cleared. All playing and queued sounds are canceled. All sound events on the track are ignored/discarded while the track is stopped.
- `pause` - The track is paused (with an optional fade out time). All sound processing on the track is paused. The track will pick-up where it left off when played/resumed. All sound events on the track are ignored/discarded while the track is paused.
- `set_volume` - Set a new volume level for the track (with an optional timed fade from the current volume level).
- `stop_all_sounds` - Stops all sounds currently playing on the track (with optional fade out time) and cancels any pending sounds in the track sound queue. The `fade_out` setting for any playing sounds will be ignored. The track will continue to process new sound events.

Optional settings

The following sections are optional in the `track_player:` section of your config. (If you don't include them, the default will be used).

fade:

Single value, type: `time string (secs)` (*Instructions for entering time strings*). Default: `0.1sec`

The number of seconds over which to fade the specified track action. Applies to all track player actions.

volume:

Single value, type: `gain setting` (-inf, db, or float between 0.0 and 1.0). Defaults to empty.

The new volume setting for the track. As with all volume parameters in MPF, this item can be represented as a number between 0.0 and 1.0 (1.0 is max volume, 0.0 is off, 0.9 is 90%, etc.) It also can be represented as a decibel string from -inf to 0.0 db (ex: `-3.0 db`). This setting only applies to the `set_volume` action and will be ignored for all others.

Related How To guides

- *Track player*
- *Sound & Audio Tips & Tricks*

trinamics_steprocker:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `trinamics_steprocker:` section of your config is where you configure the *trinamics steprocker platform*.

Required settings

The following sections are required in the `trinamics_steprocker:` section of your config:

port:

Single value, type: `string`. Defaults to empty.

Serial port to use to connect to the steprocker.

Related How To guides

- *Trinamic's StepRocker*

twitch_client:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `twitch_client:` section of your config is where you configure the built-in Twitch chat monitor.

Before using this plugin you must install the irc library with `pip3 install irc`

```
twitch_client:
  user: TwitchBotAccount
  password: oauth:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  channel: ChatChannel
```

```
machine_vars:
  twitch_user:
    initial_value: 'TwitchBotAccount'
    value_type: str
  twitch_password:
    initial_value: 'oauth:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
    value_type: str
  twitch_channel:
    initial_value: 'ChatChannel'
    value_type: str

twitch_client:
  user_var: twitch_user
  password_var: twitch_password
  channel_var: twitch_channel
```

DO NOT CHECK YOUR PASSWORD INTO SOURCE CONTROL. If you use a service like Github you should not check in your password. If this is stored publicly then someone could log in as you on Twitch.

Optional settings

The following sections are optional in the `twitch_client:` section of your config. (If you don't include them, the default will be used).

channel:

Single value, type: `string`. Defaults to empty.

The channel on Twitch which will be monitored for messages.

channel_var:

Single value, type: `string`. Defaults to empty.

his is the machine variable name that contains the channel on Twitch which will be monitored for messages.

password:

Single value, type: `string`. Defaults to empty.

This is a Twitch OAuth token, not the actual password of the user. You can generate this token at https://twitchapps.com/tmi/

password_var:

Single value, type: `string`. Defaults to empty.

This is the machine variable name that contains the password to use when connecting to Twitch, This is a Twitch OAuth token, not the actual password of the user. You can generate this token at https://twitchapps.com/tmi/

When you run `mpf -b` it may show your token in the machine variables portion of the window. Be careful what you share on stream.

user:

Single value, type: `string`. Defaults to empty.

This is the user that will connect to Twitch. You may want to create a separate bot account on Twitch to use for this purpose.

user_var:

Single value, type: `string`. Defaults to empty.

This is the machine variable name that contains the user that will connect to Twitch. You may want to create a separate bot account on Twitch to use for this purpose.

Related How To guides

Todo: *Help us to write it*

variable_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `variable_player:` section of your mode config lets you add, subtract, or replace player variables based on events that are posted.

At the most basic level, you can use this to add to a player's score (which is technically adding value to the player variable called *score*), but in reality you can affect any player or machine variable.

Here's an example:

```
##! mode: mode1
variable_player:
  target_1_hit:
    score: 1000     # adds 1000 to the player's "score" variable
  ramp_1_hit:
    score: 10000    # adds 10,000 to the player's "score" variable
    ramps: 1        # adds 1 to the player's "ramps" variable
  ramp_1_timeout:
    ramps:
```

(continues on next page)

(continued from previous page)

```
      int: 0          # sets the player's "ramps" variable to 0.
      action: set     # means that this event will "set" (or reset) the variable to the value, rather␣
↪than add to it
  ramp_2_hit:
    score:
      int: 25000 * current_player.ramps     # multiplies the value of the current player's "ramps"␣
↪variable by 25,000 and adds the result to the player's "score" variable
      block: true      # "blocks" this event from being passed to variable player sections from lower-
↪priority modes
  counter_treasure_value_complete:
    treasure_name:
      string: RUBY     # Sets the player's "treasure_name" variable to a string called "RUBY"
```

See *Variable player* for details.

Settings

Like many sections of MPF configs, the `variable_player:` section format is generically setup like this:

```
variable_player:
   some_event:
      <settings>
   some_other_event:
      <settings>
   another_event:
      <settings>
```

The following settings can be used with each event section listed in your variable_player section:

Example

You can include any player variable under an event to add numeric value to that variable. (If the variable doesn't exist, it will set the player variable to that.) For example:

```
##! mode: mode1
variable_player:
  some_event:
    score: 1000
    aliens: 1
    bonus: 10
```

The above config will add 1000 to the "score" player variable, 1 to the "aliens" player variable, and 20 to the "bonus" player variable when the event called *some_event* is posted. Note that you don't even need to include a "score" if you just want to add to other player vars.

Note that you can use a *dynamic value* for this setting too, which means you can pull in values from other player variables, device states, etc. and do math on them.

Optional settings

The following sections are optional in the `variable_player:` section of your config. (If you don't include them, the default will be used).

action:

Single value, type: one of the following options: add, set, add_machine, set_machine. Default: add

By default, the variable player entries will be added to the existing value of a player variable. If you want to replace or reset the value of the player var, you can add `action: set` to the entry. However to do this, you have to indent that setting under the player var name, and then specify the value in the "int:" section. For example, if you want the example from the above section to reset the aliens player variable to 1 instead of adding 1 to the current value, it would look like this:

```
##! mode: mode1
variable_player:
  some_event:
    score: 1000
    aliens:          # the player var you want to reset
      int: 1         # the integer value you're resetting this player var to
      action: set    # means you're resetting it, rather than adding to it
    bonus: 10
```

Starting in MPF 0.33, you can also add and set machine variables, by specifying `action: add_machine` or `action: set_machine`. In these cases the machine variable is specified just like the player variable in the "set" example above.

block:

Single value, type: `boolean` (`true`/`false`). Default: `false`

This is useful if you have a shot in a base mode that scores 500 points, but then in some timed mode you want that shot to be 5,000 points but you don't also want the base mode to score the 500 points on top of the 5,000 from the higher mode.

Note that when you use block, you also have to include the `int:`, `float:`, or `string:` setting indented. For example:

```
##! mode: mode1
variable_player:
  ramp_1_hit:
    score:
      int: 5000
      block: true
```

There is also a shorthand way:

```
##! mode: mode1
variable_player:
  ramp_1_hit:
    score: 5000|block
```

float:

Single value, type: `number` or `template` (will be converted to floating point; *Instructions for entering templates*). Defaults to empty.

Adds or sets a player or machine variable to the specified float value. The `int:` setting takes priority over the `float:` setting so if both are present only the `int:` will be used. You can use *placeholders* which evalute to float as well.

int:

Single value, type: `integer` or `template` (*Instructions for entering templates*). Defaults to empty.

Adds or sets a player or machine variable to the specified integer value (this is the most common use of the variable_player). The `int:` setting takes priority over the `float:` setting so if both are present only the `int:` will be used. You can use *placeholders* which evalute to int as well.

player:

Single value, type: `integer`. Defaults to empty.

```
##! mode: mode1
variable_player:
  add_score_to_player_2:
    score:
      int: 1000
      player: 2
```

If the `player:` setting is not used, then this variable_player entry will default to the current player.

string:

Single value, type: template_str. Defaults to empty.

Here's an example from *Brooks 'n Dunn* where there is a player variable (set via a counter) which tracks the player's current album value. We use the variable_player section tied to the events posted when the player variable changes and conditional events to set the current name of the album value, like this:

```
##! mode: mode1
variable_player:
  player_album_value{value==1}:
    album_name:
      string: SILVER
  player_album_value{value==2}:
    album_name:
      string: GOLD
  player_album_value{value==3}:
    album_name:
      string: PLATINUM
  player_album_value{value==4}:
    album_name:
      string: DOUBLE PLATINUM
  player_album_value{value==5}:
    album_name:
      string: QUINTUPLE PLATINUM
  player_album_value{value>5}:
```

(continues on next page)

(continued from previous page)

```
  album_name:
    string: OFF THE CHARTS!
```

The above config lets us always have a player var called "album_name" we can use in slides and widgets which matches the value of the album, and it's automatically updated whenever the player var "album_value" changes.

Related How To guides

- *Variable player*
- *Scoring*
- *Tutorial step 15: Add scoring*
- *Persisting the State of a Logic Block in a Player Variable*

video_pools:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `video_pools:` section of your config is where you. . .

Todo: *Help us to write it*

videos:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `videos:` section of your config is where you configure non-default parameter values for any video assets you want to use in your game. Note: You do *not* have to have an entry for every single video you want to use, rather, you only need to add individual assets to your config file that have settings which different from other assets in that asset's folder. (This section is part of the MPF media controller and only available if you're using MPF-MC for your media controller.)

More information on working with assets is in the *Assets* section of the documentation.

Each sub-entry in your `videos:` section is the name that MPF will use to refer to that asset. (In other words it's how you specify that asset in other areas of your config files.) The asset manager works by first scanning the file system to build up a list of asset files it finds. Then it looks at the config to see if there are any additional settings specified for each asset.

For example:

```
videos:
  intro_video:
    width: 100
    height: 70
    file: mpf_video_small.mpg
```

So in the example above, if the asset manager found a file called `mpf_video_small.mpg` on disk, then it will also see the `intro_video` entry in the config file and know that those two match. (The "match" is just based on the part of the file name without the extension, so the settings entry for `intro_video:` would match `mpf_video_small.mpg` and `mpf_video_small.m4v`. In other words, don't name two files with the same name if you want to keep them straight.)

Optional settings

The following sections are optional in the `videos:` section of your config. (If you don't include them, the default will be used).

events_when_played:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A list of one or more names of events that MPF will post when this video is played. Enter the list in the MPF config list format. These events are posted exactly as they're entered.

events_when_stopped:

List of one (or more) events. Those will be posted by the device. Defaults to empty.

A list of one or more names of events that MPF will post when this video stops playing. Enter the list in the MPF config list format. These events are posted exactly as they're entered. These events can be useful to trigger some action when a video has finished playing (like remove a slide).

file:

Single value, type: `string`. Defaults to empty.

Sometimes you might want to name a file one thing on disk but refer to it as another thing in your game and config files. In this case, you can create an `file:` setting in an asset entry. (Note the file: `hello_face_300.jpg` setting in the example above, and note that it includes the file extension.) In this example, you would refer to that image asset as `hello_face` even though the file is `hello_face_300`.

You might be wondering why this exists? Why not just change the file name to be whatever you want and/or who cares what the name is? The reason this function exists is because it allows for the separation of the actual file on disk from the way it's called in the game. For example, you could use this to create two sets of assets—one for a traditional DMD and one for a color DMD—and then you could refer to the asset by its generic name throughout your configs. (In other words, you could swap out assets for different physical machine types without having to update your display code.) That said, we expect that 99% of people won't use this `file:` setting, which is fine.

height:

Single value, type: `number` (can be integer or floating point). Defaults to empty.

The height of this video, in pixels.

load:

Single value, type: `string`. Default: `preload`

Videos are always streamed from disk (rather than preloaded into memory), so this setting has no effect with video assets.

priority:

Single value, type: `integer`. Default: `0`

Loading priority of this asset.

width:

Single value, type: `number` (can be integer or floating point). Defaults to empty.

The width of this video, in pixels.

Related How To guides

Todo: *Help us to write it*

virtual_platform_start_active_switches:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `virtual_platform_start_active_switches:` section of your config is where you define all switches which should start as active when running your machine with the virtual or smart_virtual platform (e.g. when running `mpf -X`).

This is an example:

```
switches:
  s_ball_switch1:
    number:
  s_ball_switch2:
    number:
```

(continues on next page)

(continued from previous page)

```
  s_ball_switch3:
    number:
# Start with two (virtual) balls
virtual_platform_start_active_switches:
  - s_ball_switch1
  - s_ball_switch2
```

Related How To guides

- *Using MPF without physical hardware*
- *Troughs / Ball Drains*

virtual_segment_display_connector:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `virtual_segment_display_connector:` section of your config is where you configure the connector that establishes the link between segment displays and the virtual segment display emulator widgets in the MPF-MC.

```
virtual_segment_display_connector:
  segment_displays: display1
```

Optional settings

The following sections are optional in the `virtual_segment_display_connector:` section of your config. (If you don't include them, the default will be used).

bcp_connection:

Single value, type: `string`. Default: `local_display`

The name of the BCP connection the MPF-MC is connected to. Normally this does not need to be modified as the default value should be correct.

debug:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Todo: *Help us to write it*

segment_displays:

List of one (or more) values, each is a type: string name of a *segment_displays* device. Defaults to empty.

A list of one or more segment display names which is used to specify which segment displays should be activated in the connector to send the appropriate information to the MPF-MC.

Related How To guides

Todo: *Help us to write it*

widget_player:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

Note: This section can also be used in a show file in the `widgets:` section of a step.

The `widget_player:` section of your config is where you configure widgets to be added to, removed from, or updated on slides (or parent frames) based on based on events being posted.

This is an example:

```
widget_player:
  some_event:
    widget_1:
      slide: slide_2
```

It will add widget_1 to slide_2.

See *Widget player* for details.

Settings

The following sections can be added under the the a particular widget's settings `widget_player:` section of your config. (If you don't include any of them, the default will be used).

So again, the format in a config file would be:

```
#config_version=5

widget_player:
   some_event:
      name_of_your_widget:
         <list of settings below go here>
```

(continues on next page)

(continued from previous page)

```
  some_other_event:
    name_of_a_different_widget:
      <list of settings below go here>
```

And the format in a show file would be:

```
#show_version=5

- duration: 1s
  widgets:
    name_of_your_widget:
      <list of settings below go here>
    name_of_a_different_widget:
      <list of settings below go here>
```

Here are the settings you can use:

Optional settings

The following sections are optional in the `widget_player:` section of your config. (If you don't include them, the default will be used).

action:

Single value, type: one of the following options: add, remove, update. Default: `add`

`add` The widget or widget group is added to the slide or display target.

`remove` The widget or widget group is removed from the slide or display target.

`update` One or more of the widget or widget group's properties is updated.

key:

Single value, type: `string`. Defaults to empty.

Used to uniquely identify a widget. With "add" actions, this sets the key name, and with "remove" or "update" actions, the key is used to identify which widget should be removed or updated.

Note that more than one widget (across displays and across slides) can have the same key, and if you remove a widget based on a key, it will remove all the widgets with that key. (In fact this is how MPF works internally to remove all widgets that were created by a mode when that mode ends.)

See the *Widget Keys* guide for details.

slide:

Single value, type: `string`. Defaults to empty.

The name of the slide you want to add this widget to. If this is not specified, then the widget will be added to whichever slide is currently active on the default display.

target:

Single value, type: `string`. Defaults to empty.

The name of the display or slide frame this widget will be added to. When this setting is used, the widget is not added to a slide, rather, it's added "on top" of the slide (to the parent display or slide frame). See the *Widget layers, z-order, & parent frames* guide for details.

Note that the `target:` and `slide:` setting are fundamentally not compatible with each other. If you used both, the `target:` setting will be used and the `slide:` value will be ignored.

widget_settings:

Unknown type. See description below.

```
widget_player:
  trigger_event:
    my_widget:
      widget_settings:
        z: 1
```

Related How To guides

- *Widget player*

widget_styles:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `widget_styles:` section of your config is where you configure styles for your widgets.

Default styles for widget types

You can define defaults for certain *widget types*. A widget will use the style `(name)_default` if no other style is specified. For instance, a default style for all *text widgets* would look like:

```
widget_styles:
  text_default:
    font_size: 21
    color: red
```

Specifying widget styles

You can also specify re-usable styles and apply them to widgets. In the following example, the text "HELLO" will render at font size 100:

```
widget_styles:
  big_style:
    font_size: 100
slides:
  slide1:
    - type: text
      text: HELLO
      style: big_style
```

You can supply multiple styles to a single widget, and they will be applied in the order given.

```
widget_styles:
  warning_text:
    font_size: 12
    color: yellow
  bottom_left:
    anchor_x: left
    anchor_y: bottom
    x: 5
    y: 5
  hurryup:
    color: red
widgets:
  timer_runout:
    - type: text
      text: Hurry!
      style: warning_text, bottom_left, hurryup
```

In the above example, the text "Hurry!" will be anchored in the lower-left of the display and rendered at size 12 and color red. Notice that the color from the *hurryup* style overwrites the color from *warning_text* style, because of the order the styles are listed in the widget.

The config reference below is incomplete. You can use all settings of your widget.

Optional settings

The following sections are optional in the `widget_styles:` section of your config. (If you don't include them, the default will be used).

color:

Single value, type: `color` (*color name*, *hex*, or list of values *0-255*). Default: `ffffffff`

The color of the widget.

Related How To guides

- *Working with Fonts*

widgets:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**YES**

The `widgets:` section of your config is where you pre-define "named" widgets that you can then use later in shows and the widget_player section of a config file. See the *Widgets* guide for details. You can test slides and widgets interactively using *Interactive MC (iMC)*.

Since there are many different types of widgets in MPF, it doesn't make sense to list all the widget type and all the options here. Instead check out the *Widgets* documentation which has all the details, including config file reference for different types of widgets.

window:

Config file section

Valid in *machine config files*	**YES**
Valid in *mode config files*	**NO**

The `window:` section of your config is where you configure the properties of the main on-screen window which is created by MPF-MC.

```
window:
  width: 800
  height: 600
  title: Mission Pinball Framework
  resizable: true
  borderless: true
  fullscreen: false
  exit_on_escape: true
  source_display: window
  effects:
    - type: dmd
```

Note: If you do not add a `window:` section to your machine config, MPF will create a window at the default size of 800x600.

Optional settings

The following sections are optional in the `window:` section of your config. (If you don't include them, the default will be used).

borderless:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Controls whether the pop-up window has a border (the "frame") around it.

effects:

Unknown type. See description below.

An optional list of effects to apply to the window contents. These effects perform image processing to the source image and can be used to get an old school "DMD look" or "color DMD look" to your window as well as other special effects. For more information on effects, please review the *effects* documentation.

exit_on_escape:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Controls whether the MPF MC shuts down when the `Esc` key is pressed.

fullscreen:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Controls whether the pop-up window should be a full screen window (if the value is "true") or whether it should be a regular window.

height:

Single value, type: `integer`. Default: `600`

The initial height of the popup window, specified in pixels.

icon:

Single value, type: `string`. Defaults to empty.

The icon for the window which will be shown in the title bar.

left:

Single value, type: `integer`. Defaults to empty.

Used to position a non-fullscreen window in a precise location on the screen. (This is useful if you're using an LCD display in your machine and your backbox has a smaller opening than the size of the screen. In that case you need to make sure the pop-up window always shows up in the proper location.)

The `left:` value specifies how many pixels the left edge of the window will be offset from the left edge of the screen. (See the `top:` setting to control the vertical placement.)

maxfps:

Single value, type: `integer`. Default: `60`

Sets the maximum frames-per-second that the window is updated. Setting a lower value can potential save CPU / GPU usage.

minimum_height:

Single value, type: `integer`. Default: `0`

If you have a resizable window, this specifies the minimum height the window can be resized to.

minimum_width:

Single value, type: `integer`. Default: `0`

If you have a resizable window, this specifies the minimum width the window can be resized to.

no_window:

Single value, type: `boolean` (`true`/`false`). Default: `false`

Controls whether the pop up window is used.

resizable:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Specifies whether the pop-up window can be resized (by dragging an edge with the mouse). If your window is full screen, then this setting will have no effect.

show_cursor:

Single value, type: `boolean` (`true`/`false`). Default: `true`

Specifies whether the mouse cursor should be drawn when the pointer is moved over the window. If you set this to False/No, then when you drag the pointer over the window, the pointer will disappear.

source_display:

Single value, type: `string`. Default: `window`

The name of the MPF display that will be used for the source content for the pop-up window.

title:

Single value, type: `string`. Default: `Mission Pinball Framework`

The text that's shown in the window title bar (assuming your window is not full screen and not borderless).

top:

Single value, type: `integer`. Defaults to empty.

Used to position the pop up window in a fixed position when MPF MC starts.

See the setting `left:` for details.

width:

Single value, type: `integer`. Default: `800`

The initial width of the popup window, specified in pixels.

Related How To guides

- *Using an LCD for a display*
- *Using multiple screens*
- *How to give your on-screen window the DMD "dot look"*

CHAPTER 26

Events

The concept of *events* is one of the most important concepts in MPF. MPF is an event-driven framework, and just about everything is either posting and event or responding to an event that was posted.

There are several important concepts about events in MPF that you should understand:

Events Overview

Video about events in MPF:

https://youtu.be/G3UbVP8gFU0

It's easiest to understand the concept of events by going through some examples.

For example, you might have a `variable_player:` entry in your config which watches for an event called *target1_hit*, and when it sees it, it adds 1000 points to the player's score, like this:

```
##! mode: base
variable_player:
  target1_hit:
    score: 1000
```

What's really happening behind the scenes here is MPF's variable_player system tells the event system, "Hey, if you see an event called *target1_hit*, let me know about it." (This is called "registering a handler", because the variable_player system is registering with the event since that it can handle that event.)

Then later on, the switch for target 1 gets activated, and the shot controller posts the event called *target1_hit*. The Event Manager says, "Hey, I remember the variable_player system wanted to know about that", so it tells the variable_player system that *target1_hit* was just posted and the variable_player system can wake up and deal with it (adding the points, in this case).

So really there are two parts to the events system:

- Things that generate (post) events.
- Things that take action on (handle) events.

Let's look at each of these.

Things that generate (post) events

There are hundreds of different things that post events in MPF (for all sorts of reasons). Just to pick some random examples of things that post events:

- A switch is hit
- A player variable changes
- A timer expires
- A mode stops or starts
- A new slide is shown on the display
- A ball drains
- A ball enters a ball device
- A new player's turn starts
- etc.

We actually have a giant list of all the events that are posted by everything in MPF. This is called the *event_reference*. (It's also linked from the "Reference" section in the menu on the left of every page in the docs website since it's so important.)

As you read through the rest of the documentation for various aspects of MPF, you'll see settings for things like `events_when_XX:` with the "XX" being some state.

For example, logic blocks have a setting called `events_when_hit:` where you can enter the name of an event. (In that case the name can be whatever you want, like `events_when_hit: mpf_is_awesome`, and then when that logic block is hit, it will post the event *mpf_is_awesome*, and any other components that are registered for that event will see it and take their respective action.

This means that while the event reference is useful because it shows all the *built-in* events, your machine will have lots of other events not on that list that you define.

Things that take action on (handle) events

The flip side of things that post events is things that taken action on (or "handle") events. These are the things that watch for certain event names, and then when they see them, they take action.

Some random examples:

- The game mode will look for ball_drain events which it will handle by ending the current player's ball.
- The variable_player system might look for a shot hit event to add points to the player's score.
- A jackpot mode might look for a ramp made event to play a show which will flash some lights and display a jackpot slide.

- A mode might look for the event which comes from shooting a ball into a ball lock to start a multiball mode.
- etc.

As you'll see as you read through the MPF documentation, there are two main ways (plus a lot of little ways) to make things happen when certain events are posted:

In the various *config players* (*slide_player*, *light_player*, *show_player*, etc.), you create entries based on event names.

For example, in a config file:

```
slide_player:
  mpf_is_awesome: my_slide
```

The above config will show the slide called "my_slide" on the display when the event *mpf_is_awesome* is posted. Of course this could be any event, including one from the Events Reference list or a custom event like we discussed above.

Also, a lot of things in MPF have `XX_events:` settings, (the "XX" will be some word) which is where you can event event names that cause that action to happen. For example, you may have a drop target configured like this:

```
drop_targets:
  my_drop_target:
    switch: s_drop_target_1
    reset_coil: c_drop_target_reset
    reset_events: mpf_is_awesome
```

In this case, when the event *mpf_is_awesome* is posted, that will cause that drop target to reset. Again, this is just one random example of the literally hundreds of things that can take action on events, and these events could be from the master events list or your own custom events.

The Event Manager

One of MPF's internal core components is called the *Event Manager*. The event manager keeps track of the hundreds of handlers that have registered for different events, and it's what other components contact when they want to post and event.

When an event is posted, the event manager contacts the handlers to let them know that they need to take action on their event.

Luckily the complexity of the event manager is hidden from you—all you have to know is that events are posted and handlers can act on them.

Finally, here are a few more random thoughts about events in MPF:

- There are lots and lots of events in MPF. Sometimes they come really fast—a dozen or more in a few milliseconds.
- Not every event will have a handler registered. If something posts an event and nothing is registered to handle it, so be it!
- Multiple handlers can be registered for the same event. In this case the event manager just notifies the handlers one-by-one.

- Event handlers are constantly added and removed throughout the lifecycle of a game. (For example, when a mode starts, all sorts of handlers are registered to watch for things that mode needs, and when the mode ends, those handlers are removed.)
- Event names are *not* case sensitive. (They're technically all converted to lowercase internally.)

Conditional Events

So far we've talked about how events are just strings of text, for example:

- ball_started
- game_ending
- shot1_hit
- mode_jackpot_starting
- etc.

However, it's possible for events to have key/value parameters attached to them.

For example, when the "ball_started" event is posted, it has two parameters attached to it: "ball" (which is the number of the ball that's started), and "player" which is the number of the player whose ball just started.

This means that the "ball_started" event isn't just MPF saying, "Hey, a ball just started", rather, it's more like MPF saying, "Hey, a ball just started for player 2, ball 3.". You can also see that in your mpf log:

```
INFO : EventManager : Event: ======'ball_started'====== Args={'player': 2, 'ball': 3}
```

By the way, in case you're wondering how we know that the `ball_started` event has those parameters (or even that `ball_started` is an event), they're all in the *event reference guide*, and the entry for *ball_started* lists the parameters it has along with an explanation of what those mean.

In addition to parameters in your event you can also reference most devices in your machine. For instance, you might want to start a mode after a counter reached a certain value. You can reference any *placeholder variable* in a condtion and also apply arbitrary logic/conditions to them.

Using keyword arguments in your config files

What's *really cool* about event parameters is that you can use them in your config files when you enter things that take action on events.

For example, here's a section of a config file that would show a slide called "lets_go" when the *ball_started* event was posted:

```
slide_player:
  ball_started: lets_go
  ball_ended:
    lets_go:
      action: remove
```

The example above will show that slide any time that the *ball_started* event was posted, regardless of what the values of the parameters are.

However, you can enter the event name in your config file a bit differently so that the action only takes place if that event is posted AND if the parameters have certain values.

For example:

```
slide_player:
  ball_started{ball==1}: first_ball_intro
  ball_ended:
    first_ball_intro:
      action: remove
```

In the above example, the slide "first_ball_intro" will only be posted when the *ball_started* AND when the value of ball is 1. (Since this entry doesn't mention "player", then this action would happen when ball 1 is started for any player.)

Of course you can use multiple entries with different values, like this:

```
slide_player:
  ball_started{ball==1}: first_ball_intro
  ball_started{ball>1}: lets_go
  ball_ended:
    first_ball_intro:
      action: remove
    lets_go:
      action: remove
```

In this case, when the *ball_started* event is posted for Ball 1, the "first_ball_intro" slide will be shown. And if it's posted with a ball after Ball 1, the "lets_go" slide will be posted.

You can also combine things here using `and` or `or`. For example:

```
slide_player:
  ball_started{ball==1 or ball==3}: special_slide
```

Now the "special_slide" will be shown for either ball 1 *or* ball 3.

You can also combine with "and", for example:

```
slide_player:
  ball_started{ball==3 and player==1}: special_slide
```

Now the "special_slide" will only show when the *ball_started* event is posted for player 1, ball 3 (but not player 2, ball 3, etc.).

Feeling crazy yet?

In addition to keyword arguments from events), you can also use `current_player.` to access player variables, `players[x]` to access player variables from any player (x is the player index), `machine.` to access machine variables, `game.` game attributes, and `settings.` to access operator settings.

```
slide_player:
  ball_started{current_player.score > 1000000}: you_rule
  ball_started{current_player.score < 10000 and ball == 3}: you_stink
```

The above config will show the slide "you_rule" any time the *ball_started* event is posted and the player's score is more than 1 million. It will also show the slide "you_stink" if ball 3 is starting and the player has less than 10,000 points.

But wait, there's more!

You can also use standard math operators (`+`, `-`, `*`, `/`, `//`, etc.) to evaluate whether the action should take place:

```
slide_player:
  ball_started{ball > 1 and current_player.score < ((ball - 1) * 10000)}: uh_oh
```

This will post the slide "uh_oh" if the player is starting a ball after Ball 1 and their score is less than an average of 10k points per ball. (Notice that you can also use parentheses to control the order of operation stuff you learned in school.)

Most likely you wouldn't get that complex, but it's nice to know that you can if you want. :)

You can also reference devices in your machine. The syntax for that is `device.DEVICE_TYPE.DEVICE_NAME.PLACEHOLDER`. For instance, to reference the `value` of a `counter` called `your_mode_counter` you would use `device.counters.your_mode_counter.value`. In the following example we show a slide when the value of the counter is above `5` in ball `3`

```
slide_player:
  ball_started{ball == 3 and device.counters.your_mode_counter.value > 5}: nearly_did_all_modes
```

You can use all *placeholder variables*.

Subscribed config players

Sometimes you want to play a show, display a slide or enable a light when certain condition hold true and remove/disable it when the condition no longer holds. This would usually require two config player entries with two different events to add and remove the show (or light). However, MPF supports subscriptions in config players for certain (not all) variables.

This is an example:

```
light_player:
  "{machine.test_machine_var == 23}":
    led4: red
  "{current_player.test_player_var == 42}":
    led5: red
```

If will turn `led4` to `red` once the machine variable `test_machine_var` becomes `23` and turns `led4` back to `off` once `test_machine_var` becomes something else. Same for `led5` and player variable `test_player_var`.

Comparisons

- == equal
- != not equal
- > greater than
- >= greater than or equal to
- < less than
- <= less than or equal to

Operators

- `+` add
- `-` subtract (or negative if there's no space after it)
- `*` multiply
- `/` divide
- `^` power (exponent)
- `%` modulus
- `^=` bit xor
- `not`
- `and`
- `or`

Handler Priorities

When you have some code you want to register to be a handler for an event, you can optionally specify a priority. (Priority is just an integer value.) The default priority for events is 1. If you want a guarantee that a certain event handler will fire last, then register that handler with a priority that's lower than any other handler for that event. And if you want to guarantee that a handler fires first, register it with a higher priority. (In this case, "higher" and "lower" are literal. A handler with a priority of 500 will be called before a handler of 100.)

The actual integer values of the priorities are arbitrary. They're called one-by-one, one after the other, in order from highest to lowest. Whether your priorities are 3, 2, and 1, or 1000, 100 and 0, or 1000, 999, 998, and 1 makes no difference.

MPF automatically registers event handlers from modes with the priority of that mode, meaning high-priority modes get access to an event before lower-priority modes. (This is useful since it gives higher-priority modes a chance to "block" events from lower-priority modes.)

See *Device Control Events* on how to use event handlers in devices.

Types of events

There are several different *types* of events in MPF, including:

- Basic
- Queue

You can find the details of how to use each of these events by reading through the API documentation for the event manager, but here's a quick overview.

Video about events in MPF:

https://youtu.be/G3UbVP8gFU0

Basic Events

The basic event is a simple event with a name (and possibly keyword argument pairs) that is posted.

The event manager will call the registered handlers one-by-one in the order of their priority (from when they registered).

Queue Events

Queue events are similar to basic events, except that the event won't actually finish until all the handlers say it's ok to do so.

The *game_ending* event is an example of a queue event. When the game is over, *game_ending* is posted, and when that's done, *game_ended* is posted and the attract mode starts again. However there are several modes that might want to "block" the completion of game_ending until they can do whatever they need to do. For example, if match is enabled, it will want to block game_ending until it can run the match animation. If a player has achieved a high score, the high score mode will want to block game ending, etc.

You can create your own queue events with the *queue_event_player:* and *queue_relay_player:* config file sections.

Note for Programmers

If you're a programmer and familiar with Python, you'll notice in the source code that there are more types of events than just basic and queue events. The basic and queue events are the only ones that are exposed via config files, but you'll notice there are boolean and relay events, and that there are asynchronous versions of all events too. See the API reference for details.

Event Reference

Here's a list of all the "built in" events that are included in MPF and the MPF MC. Of course your own machine could include custom events that aren't on the list here.

Every event in MPF is just a string of text. You'll see that in many cases, the actual event that's posted has a slight variation of the event text, typically incorporating something about which mechanism or logic device posted the event.

For example, the switch event called *(name)_active* will replace the "(name)" part of the event text with the actual switch name. So the when a switch called `s_left_slingshot` is activated, it will posted an event called *switch_s_left_slingshot_active*.

achievement_(name)_changed_state

MPF Event

Achievement (name) changed state.

Valid states are: disabled, enabled, started, completed, stopped

This is only posted once per state. Its also posted on restart on the next ball to restore state.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

restore true if this is reposted to restore state

selected Whatever this achievement is selected currently

state Current state

Event is posted by *achievements:*

achievement_(name)_state_(state)

MPF Event

Achievement (name) changed to state (state).

Valid states are: disabled, enabled, started, completed, stopped

This is only posted once per state. Its also posted on restart on the next ball to restore state and when selection changes.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

restore true if this is reposted to restore state

selected Whatever this achievement is selected currently

state Current state

Event is posted by *achievements:*

asset_loading_complete

MPF Event

Posted when the asset manager has loaded all the assets in its queue.

Note that this event does *NOT* necessarily mean that all asset loading is complete. Rather is just means that the asset manager has loaded everything in its queue.

For example, when the MPF-MC boots, it will load the assets it is configured to load on start. However, if the MPF MC is started but MPF is not, then the MPF MC will load its assets and then post this *asset_loading_complete* event when it's done. Then when MPF is started and connects, MPF will need to load its own assets, which means the MPF MC will post more *loading_assets* and then a final *asset_loading_complete* event a second time for the MPF-based assets.

This event does not have any keyword arguments

ball_drain

MPF Event

A ball (or balls) has just drained. (More specifically, ball(s) have entered a ball device tagged with "drain".)

This is a relay event.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

balls The number of balls that have just drained. Any balls remaining after the relay will be processed as newly-drained balls.

device The ball device object that received the ball(s)

ball_ended

MPF Event

The ball has ended.

Note that this does not necessarily mean that the next player's turn will start, as this player may have an extra ball which means they'll shoot again.

This event does not have any keyword arguments

ball_ending

MPF Event

The ball is ending. This is a queue event and the ball won't actually end until the queue is cleared.

This event is posted just after *ball_will_end*

This event does not have any keyword arguments

ball_hold_(name)_balls_released

MPF Event

The ball hold device (name) has just released a ball(s).

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

balls_released The number of balls that were just released.

Event is posted by *ball_holds:*

ball_hold_(name)_full

MPF Event

The ball hold device (name) is now full.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

balls The number of balls currently held in this device.

Event is posted by *ball_holds:*

ball_hold_(name)_held_ball

MPF Event

The ball hold device (name) has just held additional ball(s).

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

balls_held The number of new balls just held.

total_balls_held The current total number of balls this device has held.

Event is posted by *ball_holds:*

ball_save_(name)_disabled

MPF Event

The ball save called (name) has just been disabled.

This event does not have any keyword arguments

Event is posted by *ball_saves:*

ball_save_(name)_enabled

MPF Event

The ball save called (name) has just been enabled.

This event does not have any keyword arguments

Event is posted by *ball_saves:*

ball_save_(name)_grace_period

MPF Event

The ball save called (name) has just entered its grace period time.

This event does not have any keyword arguments

Event is posted by *ball_saves:*

ball_save_(name)_hurry_up

MPF Event

The ball save called (name) has just entered its hurry up mode.

This event does not have any keyword arguments

Event is posted by *ball_saves:*

ball_save_(name)_saving_ball

MPF Event

The ball save called (name) has just saved one (or more) balls.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

balls The number of balls this ball saver is saving.

early_save True if this is an early ball save.

Event is posted by *ball_saves:*

ball_save_(name)_timer_start

MPF Event

The ball save called (name) has just start its countdown timer.

This event does not have any keyword arguments

Event is posted by *ball_saves:*

ball_save_(name)_add_a_ball_timer_start

MPF Event

The multiball add a ball ball save called (name) has just start its countdown timer.

This event does not have any keyword arguments

Event is posted by *multiballs:*

ball_save_(name)_timer_start

MPF Event

The multiball ball save called (name) has just start its countdown timer.

This event does not have any keyword arguments

Event is posted by *multiballs:*

ball_search_failed

MPF Event

The ball search process has failed to locate a missing or stuck ball and has given up. This event will be posted immediately after the *ball_search_stopped* event.

This event does not have any keyword arguments

ball_search_phase_(num)

MPF Event

The ball search phase (num) has started.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

`iteration` Current iteration of phase (num)

ball_search_prevents_game_start

MPF Event

A game start has been requested, but the ball search process is running and thus the game start has been blocked. This is a good event to use for a slide player to inform the player that the machine is looking for a missing ball.

This event does not have any keyword arguments

ball_search_started

MPF Event

The ball search process has been begun.

This event does not have any keyword arguments

ball_search_stopped

MPF Event

The ball search process has been disabled. This event is posted any time ball search stops, regardless of whether it found a ball or gave up. (If the ball search failed to find the ball, it will also post the *ball_search_failed* event.)

This event does not have any keyword arguments

ball_start_target

MPF Event

Posted when a new ball starts and is ready to be physically ejected to the playfield.

This is a relay event.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

target The name of the ball_device target where the ball will be ejected to. Can be modified by a relay event handler to change the target before the ball is ejected.

ball_started

MPF Event

A new ball has started.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

ball The ball number

balls_remaining The number of balls left in the game (not including this one)

is_extra_ball True if this ball is an extra ball (default False)

player The player number

ball_starting

MPF Event

A ball is starting. This is a queue event, so the ball won't actually start until the queue is cleared.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

ball The ball number

balls_remaining The number of balls left in the game (not including this one)

is_extra_ball True if this ball is an extra ball (default False)

player The player number

ball_will_end

MPF Event

The ball is about to end. This event is posted just before *ball_ending*.

This event does not have any keyword arguments

ball_will_start

MPF Event

The ball is about to start. This event is posted just before *ball_starting*.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

ball The ball number

balls_remaining The number of balls left in the game (not including this one)

is_extra_ball True if this ball is an extra ball (default False)

player The player number

balldevice_(name)_ball_count_changed

MPF Event

The ball count for device (name) just changed.

This event may also be called without a change in some circumstances.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

balls The number of new balls in this device.

Event is posted by *ball_devices:*

balldevice_(name)_ball_eject_attempt

MPF Event

The ball device called "name" is attempting to eject a ball (or balls). This is a queue event. The eject will not actually be attempted until the queue is cleared.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

balls The number of balls that are to be ejected.

mechanical_eject Boolean as to whether this is a mechanical eject.

num_attempts How many eject attempts have been tried so far.

source The source device that will be ejecting the balls.

target The target ball device that will receive these balls.

Event is posted by *ball_devices:*

balldevice_(name)_ball_eject_failed

MPF Event

A ball (or balls) has failed to eject from the device (name).

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

balls The number of balls that failed to eject.

num_attempts How many attemps have been made to eject this ball (or balls).

retry Boolean as to whether this eject will be retried.

target The target device that was supposed to receive the ejected balls.

Event is posted by *ball_devices:*

balldevice_(name)_ball_eject_success

MPF Event

One or more balls has successfully ejected from the device (name).

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

balls The number of balls that have successfully ejected.

target The target device that has received (or will be receiving) the ejected ball(s).

Event is posted by *ball_devices:*

balldevice_(name)_ball_enter

MPF Event

A ball (or balls) have just entered the ball device called "name".

Note that this is a relay event based on the "unclaimed_balls" arg. Any unclaimed balls in the relay will be processed as new balls entering this device.

Please be aware that we did not add those balls to balls or available_balls of the device during this event.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

device A reference to the ball device object that is posting this event.

unclaimed_balls The number of balls that have not yet been claimed.

Event is posted by *ball_devices:*

balldevice_(name)_ball_entered

MPF Event

A ball (or balls) have just entered the ball device called "name".

The ball was also added to balls and available_balls of the device.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

device A reference to the ball device object that is posting this event.

new_balls The number of new balls that have not been claimed (by locks or similar).

Event is posted by *ball_devices:*

balldevice_(name)_ball_missing

MPF Event

The device (name) is missing a ball. Note this event is posted in addition to the generic *balldevice_ball_missing* event.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

balls The number of balls that are missing

Event is posted by *ball_devices:*

balldevice_(name)_broken

MPF Event

The ball device called "name" is broken and will no longer operate.

This event does not have any keyword arguments

Event is posted by *ball_devices:*

balldevice_(name)_ejecting_ball

MPF Event

The ball device called "name" is ejecting a ball right now.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

balls The number of balls that are to be ejected.

mechanical_eject Boolean as to whether this is a mechanical eject.

num_attempts How many eject attempts have been tried so far.

source The source device that will be ejecting the balls.

target The target ball device that will receive these balls.

Event is posted by *ball_devices:*

balldevice_ball_missing

MPF Event

A ball is missing from a device.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

balls The number of balls that are missing

name Name of device which lost the ball

Event is posted by *ball_devices:*

balldevice_balls_available

MPF Event

A device has balls available to be ejected.

This event does not have any keyword arguments

Event is posted by *ball_devices:*

balldevice_captured_from_(captures_from)

MPF Event

A ball device has just captured a ball from the device called (captures_from)

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

balls The number of balls that were captured.

Event is posted by *ball_devices:*

balls_in_play

MPF Event

The number of balls in play has just changed, and there is at least 1 ball in play.

Note that the number of balls in play is not necessarily the same as the number of balls loose on the playfield. For example, if the player shoots a lock and is watching a cut scene, there is still one ball in play even though there are no balls on the playfield.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

balls The number of ball(s) in play.

bcp_clients_connected

MPF Event

All BCP outgoing BCP connections have been made.

This event does not have any keyword arguments

bcp_connection_attempt

MPF Event

MPF is attempting to make a BCP connection.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

host The host name MPF is attempting to connect to.

name The name of the connection.

port The TCP port MPF is attempting to connect to

bonus_multiplier

MPF Event

Posted after "bonus_subtotal" and used to trigger the bonus multiplier screen. If the bonus multiplier is 1, then this event is skipped.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

multiplier The numeric value of the bonus multiplier.

bonus_start

MPF Event

The end-of-ball bonus is starting. You can use this event in your slide player to trigger the bonus intro slide. If the game has tilted, this event will not be posted.

This event does not have any keyword arguments

bonus_subtotal

MPF Event

Posted by the bonus mode after all the individual bonus entries have been posted and processed.

This event is typically posted just before the bonus multiplier screen, so if the bonus multiplier is 1, then this event will be skipped.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

score The score of the bonus (so far)

cancel_ball_search

MPF Event

This event will cancel all running ball searches and mark the balls as lost. This is only a handler so all you have to do is to post the event.

This event does not have any keyword arguments

clear

MPF Event

Posted to cause config players to clear whatever they're running based on the key passed. Typically posted when a show or mode ends.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

key string name of the configs to clear

client_connected

MPF-MC Event

Posted on the MPF-MC only when a BCP client has connected.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

address The IP address of the client that connected.

port The port the client connected on.

client_disconnected

MPF-MC Event

Posted on the MPF-MC only (e.g. not in MPF) when the BCP client disconnects. This event is also posted when the MPF-MC starts before a client is connected.

This is useful for triggering a slide notifying of the disconnect.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

host The hostname or IP address that the socket is listening on.

port The port that the socket is listening on.

collecting_balls

MPF Event

Posted by the ball controller when it starts the collecting balls process.

This event does not have any keyword arguments

collecting_balls_complete

MPF Event

Posted by the ball controller when it has finished the collecting balls process.

This event does not have any keyword arguments

(name)_both

MPF Event

Combo switch (name) changed to state both.

A switch from group 1 and group 2 are both active at the same time, having been pressed within the `max_offset_time:` and being active for at least the `hold_time:`.

This event does not have any keyword arguments

Event is posted by *combo_switches:*

The event name can be changed by using the "events_when_both:" attribute.

(name)_inactive

MPF Event

Combo switch (name) changed to state inactive.

Both switches are inactive.

This event does not have any keyword arguments

Event is posted by *combo_switches:*

The event name can be changed by using the "events_when_inactive:" attribute.

(name)_one

MPF Event

Combo switch (name) changed to state one.

Either switch 1 or switch 2 has been released for at least the `release_time:` but the other switch is still active.

This event does not have any keyword arguments

Event is posted by *combo_switches:*

The event name can be changed by using the "events_when_one:" attribute.

(name)_switches_1

MPF Event

Combo switch (name) changed to state switches_1.

Only switches_1 is active. max_offset_time has passed and this hit cannot become both later on. Only emited when `max_offset_time:` is defined.

This event does not have any keyword arguments

Event is posted by *combo_switches:*

The event name can be changed by using the "events_when_switches_1:" attribute.

(name)_switches_2

MPF Event

Combo switch (name) changed to state switches_2.

Only switches_2 is active. max_offset_time has passed and this hit cannot become both later on. Only emited when `max_offset_time:` is defined.

This event does not have any keyword arguments

Event is posted by *combo_switches:*

The event name can be changed by using the "events_when_switches_2:" attribute.

credits_added

MPF Event

Credits (or partial credits) have just been added to the machine.

This event does not have any keyword arguments

display_(name)_initialized

MPF-MC Event

The display called (name) has been initialized. This event is generated in the MC, so it won't be sent to MPF if the MC is started up and ready first.

This event is part of the MPF-MC boot process and is not particularly useful for game developers. If you want to show a "boot" slide as early as possible, use the *mc_ready* event.

This event does not have any keyword arguments

Event is posted by *displays:*

display_(name)_ready

MPF-MC Event

The display target called (name) is now ready and available to show slides. This event is useful with display widgets where you want to add a display to an existing slide which shows some content, but you need to make sure the display exists before showing a slide. So if you have a display called "overlay", then you can add it to a slide however you want, and when it's added, the event "display_overlay_ready" will be posted, and then you can use that event in your slide_player to trigger the first slide you want to show. Note that this event is posted by MPF-MC and will not exist on the MPF side. So you can use this event for slide_player, widget_player, etc., but not to start shows or other things controlled by MPF.

This event does not have any keyword arguments

Event is posted by *displays:*

displays_initialized

MPF-MC Event

Posted as soon as MPF MC displays have been initialized.

Note that this event is used as part of the internal MPF-MC startup process. In some cases it will be posted *before* the slide_player is ready, meaning that you *CANNOT* use this event to post slides or play sounds.

Instead, use the *mc_ready* event, which is posted as early as possible once the slide player and sound players are setup.

Note that this event is generated by the media controller and does not exist on the MPF side of things.

Also note that if you're using a media controller other than the MPF-MC (such as the Unity 3D backbox controller), then this event won't exist.

This event does not have any keyword arguments

diverter_(name)_activating

MPF Event

The diverter called (name) is activating itself, which means it's physically pulsing or holding the coil to move.

This event does not have any keyword arguments

Event is posted by *diverters:*

diverter_(name)_deactivating

MPF Event

The diverter called (name) is deativating itself.

This event does not have any keyword arguments

Event is posted by *diverters:*

diverter_(name)_disabling

MPF Event

The diverter called (name) is disabling itself. Note that if this diverter has `activation_switches:` configured, it will not physically deactivate now, instead deactivating based on switch hits and timing. Otherwise this diverter will deactivate immediately.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

`auto` Boolean which indicates whether this diverter disabled itself automatically for the purpose of routing balls to their proper location(s).

Event is posted by *diverters:*

diverter_(name)_enabling

MPF Event

The diverter called (name) is enabling itself. Note that if this diverter has `activation_switches:` configured, it will not physically activate until one of those switches is hit. Otherwise this diverter will activate immediately.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

`auto` Boolean which indicates whether this diverter enabled itself automatically for the purpose of routing balls to their proper location(s).

Event is posted by *diverters:*

drop_target_bank_(name)_down

MPF Event

Every drop target in the drop target bank called (name) is now in the "down" state. This event is only posted once, when all the drop targets are down.

This event does not have any keyword arguments

Event is posted by *drop_target_banks:*

drop_target_bank_(name)_mixed

MPF Event

The drop targets in the drop target bank (name) are in a "mixed" state, meaning that they're not all down or not all up. This event is posted every time a member drop target changes but the overall bank is not not complete.

This event does not have any keyword arguments

Event is posted by *drop_target_banks:*

drop_target_bank_(name)_up

MPF Event

Every drop target in the drop target bank called (name) is now in the "up" state. This event is only posted once, when all the drop targets are up.

This event does not have any keyword arguments

Event is posted by *drop_target_banks:*

drop_target_(name)_down

MPF Event

The drop target with the (name) has just changed to the "down" state.

This event does not have any keyword arguments

Event is posted by *drop_targets:*

drop_target_(name)_up

MPF Event

The drop target (name) has just changed to the "up" state.

This event does not have any keyword arguments

Event is posted by *drop_targets:*

enabling_credit_play

MPF Event

The game is no longer on free play. Credits are required to start a game. This event is also posted on MPF boot if the credits mode is enabled and the game is not set to free play.

This event does not have any keyword arguments

enabling_free_play

MPF Event

Credits are no longer required to start a game. This event is also posted on MPF boot if the credits mode is enabled and the game is set to free play.

This event does not have any keyword arguments

extra_ball_award_disabled

MPF Event

The award for an extra ball has just been disabled.

This event does not have any keyword arguments

Event is posted by *extra_balls:*

extra_ball_awarded

MPF Event

An extra ball has just been awarded.

This event does not have any keyword arguments

Event is posted by *extra_balls:*

extra_ball_(name)_award_disabled

MPF Event

The award for the extra ball called (name) has just been disabled.

This event does not have any keyword arguments

Event is posted by *extra_balls:*

extra_ball_(name)_awarded

MPF Event

The extra ball called (name) has just been awarded.

This event does not have any keyword arguments

Event is posted by *extra_balls:*

extra_ball_(name)_lit

MPF Event

The extra ball called (name) has just been lit.

This event does not have any keyword arguments

Event is posted by *extra_balls:*

extra_ball_group_(name)_award_disabled

MPF Event

Posted when you have the global extra ball settings set to not enable extra balls but where an extra ball would have been awarded. This is a good alternative event to use to score points or whatever else you want to give the player when extra balls are disabled.

This event does not have any keyword arguments

Event is posted by *extra_ball_groups:*

extra_ball_group_(name)_awarded

MPF Event

An extra ball from this group was just awarded. This is a good event to use to trigger award shows, sounds, etc.

This event does not have any keyword arguments

Event is posted by *extra_ball_groups:*

extra_ball_group_(name)_lit

MPF Event

An extra ball was just lit. This is a good event to use to start your extra ball lit mode, to turn on an extra ball light, to play the "get that extra ball" sound, etc.

Note that this event is posted if an extra ball is lit during play and also when a player's turn starts if they have a lit extra ball.

See also the *extra_ball_(name)_lit* for a similar event that is only posted when an extra ball is lit during play, and not if the player starts their turn with the extra ball lit.

This event does not have any keyword arguments

Event is posted by *extra_ball_groups:*

extra_ball_group_(name)_lit_awarded

MPF Event

This even is posted when an extra ball is lit during play. It is NOT posted when a player's turn starts if they have a lit extra ball from their previous turn. Therefore this event is a good event to use for your award slides and shows when a player lights the extra ball, because you don't want to use *extra_ball_group_(name)_lit* because that is also posted when the player's turn starts and you don't want the award show to play again when they're starting their turn.

This event does not have any keyword arguments

Event is posted by *extra_ball_groups:*

extra_ball_group_(name)_unlit

MPF Event

No more lit extra balls are available for this extra ball group. This is a good event to use as a stop event for your extra ball lit mode or whatever you're using to indicate to the player that an extra ball is available.

This event does not have any keyword arguments

Event is posted by *extra_ball_groups:*

flipper_cancel

MPF Event

Posted when both flipper buttons are hit at the same time, useful as a "cancel" event for shows, the bonus mode, etc.

Note that in order for this event to work, you have to add `left_flipper` as a tag to the switch for your left flipper, and `right_flipper` to your right flipper.

See *combo_switches:* for details.

This event does not have any keyword arguments

flipper_cradle

MPF Event

Posted when one of the flipper buttons has been active for 3 seconds.

Note that in order for this event to work, you have to add `left_flipper` as a tag to the switch for your left flipper, and `right_flipper` to your right flipper.

See *timed_switches:* for details.

This event does not have any keyword arguments

flipper_cradle_release

MPF Event

Posted when one of the flipper buttons that has previously been active for more than 3 seconds has been released.

If the player pushes in one flipper button for more than 3 seconds, and then the second one and holds it in for more than 3 seconds, this event won't be posted until both buttons have been released.

Note that in order for this event to work, you have to add `left_flipper` as a tag to the switch for your left flipper, and `right_flipper` to your right flipper.

See *timed_switches:* for details.

This event does not have any keyword arguments

game_ended

MPF Event

The game has ended.

This event does not have any keyword arguments

game_ending

MPF Event

The game is in the process of ending. This is a queue event, and the game won't actually end until the queue is cleared.

This event does not have any keyword arguments

game_start

MPF Event

Starts game while bypassing the many systems which have to "approve" the start. (Are the balls in the right places, are there enough credits, etc.) Use of this method is not recommended but may be useful in testing code. Instead, use the *request_to_start_game* event.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

`buttons` A list of switches tagged with *player* that were held in when the start button was released. This is used for "alternate" game starts (e.g. hold the right flipper and press start for tournament mode, etc.)

`hold_time` The time, in seconds, that the start button was held in to start the game. This can be used to start alternate games via a "long press" of the start button.

game_started

MPF Event

A new game has started.

This event does not have any keyword arguments

game_starting

MPF Event

A game is in the process of starting. This is a queue event, and the game won't actually start until the queue is cleared.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

game A reference to the game mode object.

game_will_end

MPF Event

The game is about to end. This event is posted just before *game_ending*.

This event does not have any keyword arguments

game_will_start

MPF Event

The game is about to start. This event is posted just before *game_starting*.

This event does not have any keyword arguments

init_done

MPF Event

Posted when the initial (one-time / boot) init phase is done. In other words, once this is posted, MPF is booted and ready to go.

This event does not have any keyword arguments

init_phase_1

MPF Event

Posted during the initial boot up of MPF.

This event does not have any keyword arguments

init_phase_2

MPF Event

Posted during the initial boot up of MPF.

This event does not have any keyword arguments

init_phase_3

MPF Event

Posted during the initial boot up of MPF.

This event does not have any keyword arguments

init_phase_4

MPF Event

Posted during the initial boot up of MPF.

This event does not have any keyword arguments

init_phase_5

MPF Event

Posted during the initial boot up of MPF.

This event does not have any keyword arguments

kickback_(name)_fired

MPF Event

Kickback fired a ball.

This event does not have any keyword arguments

Event is posted by *kickbacks:*

loading_assets

MPF Event

Posted when the number of assets waiting to be loaded changes.

Note that once all the assets are loaded, all the values below are reset to zero.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

loaded The number of assets that have been loaded so far.

percent The numerical percent completion of the assets loaded, express in the range of 0 to 100.

remaining The number of assets that are remaining to be loaded.

total The total number of assets that need to be loaded. This is equal to the sum of the *loaded* and *remaining* values below. It also includes assets that MPF is loading itself as well as any assets that have been reported from remotely connected BCP hosts (e.g. the media controller).

logicblock_(name)_complete

MPF Event

The logic block called "name" has just been completed.

Note that this is the default completion event for logic blocks, but this can be changed in a logic block's "events_when_complete:" setting, so this might not be the actual event that's posted for all logic blocks in your machine.

This event does not have any keyword arguments

Event is posted by *counters:*

Event is posted by *accruals:*

Event is posted by *sequences:*

logicblock_(name)_hit

MPF Event

The logic block "name" was just hit.

Note that this is the default hit event for logic blocks, but this can be changed in a logic block's "events_when_hit:" setting, so this might not be the actual event that's posted for all logic blocks in your machine.

This event does not have any keyword arguments

Event is posted by *counters:*

Event is posted by *accruals:*

Event is posted by *sequences:*

logicblock_(name)_updated

MPF Event

The logic block called "name" has changed.

This might happen when the block advanced, it was resetted or restored.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

enabled Whatever this block is enabled or not.

value The current value of this block.

Event is posted by *counters:*

Event is posted by *accruals:*

Event is posted by *sequences:*

machine_reset_phase_1

MPF Event

The first phase of resetting the machine.

These events are posted when MPF boots (after the init_phase events are posted), and they're also posted subsequently when the machine is reset (after existing the service mode, for example).

This is a queue event. The machine reset phase 1 will not be complete until the queue is cleared.

This event does not have any keyword arguments

machine_reset_phase_2

MPF Event

The second phase of resetting the machine.

These events are posted when MPF boots (after the init_phase events are posted), and they're also posted subsequently when the machine is reset (after existing the service mode, for example).

This is a queue event. The machine reset phase 2 will not be complete until the queue is cleared.

This event does not have any keyword arguments

machine_reset_phase_3

MPF Event

The third phase of resetting the machine.

These events are posted when MPF boots (after the init_phase events are posted), and they're also posted subsequently when the machine is reset (after exiting the service mode, for example).

This is a queue event. The machine reset phase 3 will not be complete until the queue is cleared.

This event does not have any keyword arguments

machine_var_(name)

MPF Event

Posted when a machine variable is added or changes value. (Machine variables are like player variables, except they're maintained machine-wide instead of per-player or per-game.)

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

change If the machine variable just changed, this will be the amount of the change. If it's not possible to determine a numeric change (for example, if this machine variable is a list), then this *change* value will be set to the boolean *True*.

prev_value The previous value of this machine variable, e.g. what it was before the current value.

value The new value of this machine variable.

Event is posted by *machine_vars:*

magnet_(name)_flinged_ball

MPF Event

The magnet called (name) has just flinged a ball.

This event does not have any keyword arguments

Event is posted by *magnets:*

magnet_(name)_flinging_ball

MPF Event

The magnet called (name) is flinging a ball by disabling and enabling the magnet again for a short time.

This event does not have any keyword arguments

Event is posted by *magnets:*

magnet_(name)_grabbed_ball

MPF Event

The magnet called (name) has completed grabbing the ball. Note that the magnet doesn't actually "know" whether it successfully grabbed a ball or not, so this even is saying that it things it did. to).

This event does not have any keyword arguments

Event is posted by *magnets:*

magnet_(name)_grabbing_ball

MPF Event

The magnet called (name) is attempting to grab a ball.

This event does not have any keyword arguments

Event is posted by *magnets:*

magnet_(name)_released_ball

MPF Event

The magnet called (name) has just released a ball.

This event does not have any keyword arguments

Event is posted by *magnets:*

magnet_(name)_releasing_ball

MPF Event

The magnet called (name) is in the process of releasing a ball.

This event does not have any keyword arguments

Event is posted by *magnets:*

master_volume_decrease

MPF Event

Decrease the master volume of the audio system.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

volume New volume as float between 0.0 an 1.0

master_volume_increase

MPF Event

Increase the master volume of the audio system.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

volume New volume as float between 0.0 an 1.0

match_has_match

MPF Event

At least one player has a match.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

match_number0 Match number for player 0

match_number1 Match number for player 1

match_numberX Match number for player X (up to max players)

winner_number Winner number

winners Number of winners (always more than 0 here)

match_no_match

MPF Event

All players missed the match number.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

match_number0 Match number for player 0

match_number1 Match number for player 1

match_numberX Match number for player X (up to max players)

winner_number Winner number

winners Number of winners (always 0 here)

max_credits_reached

MPF Event

Credits have just been added to the machine, but the configured maximum number of credits has been reached.

This event does not have any keyword arguments

mc_ready

MPF-MC Event

Posted when the MPF-MC is available to start showing slides and playing sounds.

Note that this event does not mean the MC is done loading. Instead it's posted at the earliest possible moment that the core MC components are available, meaning you can trigger "boot" slides from this event (which could in turn be used to show asset loading status, boot progress, etc.)

If you want to show slides that require images or video loaded from disk, use the event "init_done" instead which is posted once all the assets set to "preload" have been loaded.

This event does not have any keyword arguments

mc_reset_complete

MPF-MC Event

Posted on the MPF-MC only (e.g. not in MPF). This event is posted when the MPF-MC reset process is complete.

This event does not have any keyword arguments

mc_reset_phase_1

MPF-MC Event

Posted on the MPF-MC only (e.g. not in MPF). This event is used internally as part of the MPF-MC reset process.

This event does not have any keyword arguments

mc_reset_phase_2

MPF-MC Event

Posted on the MPF-MC only (e.g. not in MPF). This event is used internally as part of the MPF-MC reset process.

This event does not have any keyword arguments

mc_reset_phase_3

MPF-MC Event

Posted on the MPF-MC only (e.g. not in MPF). This event is used internally as part of the MPF-MC reset process.

This event does not have any keyword arguments

mode_(name)_started

MPF Event

Posted when a mode has started. The "name" part is replaced with the actual name of the mode, so the actual event posted is something like *mode_attract_started*, *mode_base_started*, etc.

This is posted after the "mode_(name)_starting" event.

This event does not have any keyword arguments

mode_(name)_starting

MPF Event

The mode called "name" is starting.

This is a queue event. The mode will not fully start until the queue is cleared.

This event does not have any keyword arguments

mode_(name)_stopped

MPF Event

Posted when a mode has stopped. The "name" part is replaced with the actual name of the mode, so the actual event posted is something like *mode_attract_stopped*, *mode_base_stopped*, etc.

This event does not have any keyword arguments

mode_(name)_stopping

MPF Event

The mode called "name" is stopping. This is a queue event. The mode won't actually stop until the queue is cleared.

This event does not have any keyword arguments

mode_(name)_will_start

MPF Event

Posted when a mode is about to start. The "name" part is replaced with the actual name of the mode, so the actual event posted is something like *mode_attract_will_start*, *mode_base_will_start*, etc.

This is posted before the "mode_(name)_starting" event.

This event does not have any keyword arguments

mode_(name)_will_stop

MPF Event

Posted when a mode is about to stop. The "name" part is replaced with the actual name of the mode, so the actual event posted is something like *mode_attract_will_stop*, *mode_base_will_stop*, etc.

This is posted immediately before the "mode_(name)_stopping" event.

This event does not have any keyword arguments

motor_(name)_reached_(position)

MPF Event

A motor device called (name) reached position (position) (device)

This event does not have any keyword arguments

Event is posted by *motors:*

multi_player_ball_started

MPF Event

A new ball has started, and this is a multiplayer game.

This event does not have any keyword arguments

multiball_lock_(name)_full

MPF Event

The multiball lock device (name) is now full.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

balls The number of balls currently locked in this device.

Event is posted by *multiball_locks:*

multiball_lock_(name)_locked_ball

MPF Event

The multiball lock device (name) has just locked one additional ball.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

total_balls_locked The current total number of balls this device has locked.

Event is posted by *multiball_locks:*

multiball_(name)_ended

MPF Event

The multiball called (name) has just ended.

This event does not have any keyword arguments

Event is posted by *multiballs:*

multiball_(name)_grace_period

MPF Event

The multiball ball save called (name) has just entered its grace period time.

This event does not have any keyword arguments

Event is posted by *multiballs:*

multiball_(name)_hurry_up

MPF Event

The multiball ball save called (name) has just entered its hurry up mode.

This event does not have any keyword arguments

Event is posted by *multiballs:*

multiball_(name)_lost_ball

MPF Event

The multiball called (name) has lost a ball after ball save expired.

This event does not have any keyword arguments

Event is posted by *multiballs:*

multiball_(name)_shoot_again

MPF Event

A ball has drained during the multiball called (name) while the ball save timer for that multiball was running, so a ball (or balls) will be saved and re-added into play.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

balls The number of balls that are being saved.

Event is posted by *multiballs:*

multiball_(name)_shoot_again_ended

MPF Event

Shoot again for multiball (name) has ended.

This event does not have any keyword arguments

Event is posted by *multiballs:*

multiball_(name)_started

MPF Event

The multiball called (name) has just started.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

balls The number of balls in this multiball

Event is posted by *multiballs:*

multiplayer_game

MPF Event

A second player has just been added to this game, meaning this is now a multiplayer game.

This event is typically used to switch the score display from the single player layout to the multiplayer layout.

This event does not have any keyword arguments

(name)_timeout

MPF Event

The logic block called "name" has just timeouted.

Timeouts are disabled by default but you can set logic_block_timeout to enable them. They will run from start of your logic block until it is stopped.

This event does not have any keyword arguments

Event is posted by *counters:*

Event is posted by *accruals:*

Event is posted by *sequences:*

not_enough_credits

MPF Event

A player has pushed the start button, but the game is not set to free play and there are not enough credits to start a game or add a player.

This event does not have any keyword arguments

player_add_request

MPF Event

Posted to request that an additional player be added to this game. Any registered handler can deny the player add request by returning *False* to this event.

This event does not have any keyword arguments

player_added

MPF Event

A new player was just added to this game

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

num The number of the player that was just added. (e.g. Player 1 will have *num=1*, Player 4 will have *num=4*, etc.)

player A reference to the instance of the Player() object.

player_adding

MPF Event

A new player is in the process of being added to this game. This is a queue event, and the player won't actually be finished adding until the queue is cleared.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

number The player number

player The player object for the player being added

player_(name)

MPF Event

Posted when simpler types of player variables are added or change value.

The actual event has (var_name) replaced with the name of the player variable that changed. Some examples:

- player_score
- player_shot_upper_lit_hit

Lots of things are stored in player variables, so there's no way to build a complete list of what all the options are here. Elsewhere in the documentation, if you see something that says it's stored in a player variable, that means you'll get this event when that player variable is created or is changed.

Note that this event is only posted for simpler types of player variables, including player variables that are integers, floating point numbers, or strings. More complex player variables (lists, dicts, etc.) do not get this event posted.

This event is posted for a single player variable changing, meaning if multiple player variables change at the same time, multiple events will be posted, one for each change.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

change If the player variable just changed, this will be the amount of the change. If it's not possible to determine a numeric change (for example, if this player variable is a string), then this *change* value will be set to the boolean *True*.

kwargs Additional keyword arguments to include in the event args.

player_num The player number this variable just changed for, starting with 1. (e.g. Player 1 will have *player_num=1*, Player 4 will have *player_num=4*, etc.)

prev_value The previous value of this player variable, e.g. what it was before the current value.

value The new value of this player variable.

Event is posted by *player_vars:*

player_turn_ended

MPF Event

The current player's turn has ended. This event is only posted when this player's turn is totally over. If the player gets an extra ball and shoots again, this event is not posted until after all their extra balls and it's no longer their turn.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

number The player number

player The player object whose turn is ending.

player_turn_ending

MPF Event

The current player's turn is ending. This is a queue event, and the player's turn won't actually end until the queue is cleared.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

number The player number

player The player object whose turn is ending.

player_turn_started

MPF Event

A new player's turn started. This event is only posted after the start of a new player's turn. If that player gets an extra ball and shoots again, this event is not posted a second time.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

number The player number

player The player object whose turn is starting.

player_turn_starting

MPF Event

The player's turn is in the process of starting. This is a queue event, and the player's turn won't actually start until the queue is cleared.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

number The player number

player The player object whose turn is starting.

player_turn_will_end

MPF Event

The player's turn is about to end. This event is only posted when this player's turn is totally over. If the player gets an extra ball and shoots again, this event is not posted until after all their extra balls and it's no longer their turn.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

number The player number

player The player object whose turn is over.

player_turn_will_start

MPF Event

A new player's turn will start. This event is only posted before the start of a new player's turn. If that player gets an extra ball and shoots again, this event is not posted a second time.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

number The player number

player The player object whose turn is starting.

player_will_add

MPF Event

A new player will be added to this game. This event is sent immediately prior to the player_adding event.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

number The new player number that will be added

(name)_active

MPF Event

The playfield called (name) is now active, meaning there's at least one loose ball on it.

This event does not have any keyword arguments

Event is posted by *playfields:*

(name)_ball_count_change

MPF Event

The playfield with the name (name) has changed the number of balls that are live.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

balls The current number of balls on the playfield.

change The change in balls from the last count.

Event is posted by *playfields:*

playfield_transfer_(playfield_transfer)_ball_transferred

MPF Event

The playfield_transfer called (playfield_transfer) transferred a ball from playfield (source) to playfield (target).

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

source The source playfield.

target The target playfield.

Event is posted by *playfield_transfers:*

reel_(name)_advanced

MPF Event

The reel (name) advanced to the next position.

This event does not have any keyword arguments

Event is posted by *score_reels:*

request_to_start_game

MPF Event

This event is posted when to start a game. This is a boolean event. Any handler can return *False* and the game will not be started. Otherwise when this event is done, a new game is started.

Posting this event is the only way to start a game in MPF, since many systems have to "approve" the start. (Are the balls in the right places, are there enough credits, etc.)

This event does not have any keyword arguments

reset_complete

MPF Event

The machine reset process is complete

This event does not have any keyword arguments

(name)_hit

MPF Event

The sequence_shot called (name) was just completed.

This event does not have any keyword arguments

Event is posted by *sequence_shots:*

(name)_complete

MPF Event

All the member shots in the shot group called (name) are in the same state.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

`state` name of the common state of all shots.

Event is posted by *shot_groups:*

(name)_hit

MPF Event

A member shots in the shot group called (name) has been hit.

This event does not have any keyword arguments

Event is posted by *shot_groups:*

(name)_(state)_complete

MPF Event

All the member shots in the shot group called (name) are in the same state named (state).

This event does not have any keyword arguments

Event is posted by *shot_groups:*

(name)_(state)_hit

MPF Event

A member shot with state (state) in the shot group (name) has been hit.

This event does not have any keyword arguments

Event is posted by *shot_groups:*

(name)_hit

MPF Event

The shot called (name) was just hit.

Note that there are four events posted when a shot is hit, each with variants of the shot name, profile, and current state, allowing you to key in on the specific granularity you need.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

`profile` The name of the profile that was active when hit.

`state` The name of the state the profile was in when it was hit

Event is posted by *shots:*

(name)_(profile)_hit

MPF Event

The shot called (name) was just hit with the profile (profile) active.

Note that there are four events posted when a shot is hit, each with variants of the shot name, profile, and current state, allowing you to key in on the specific granularity you need.

Also remember that shots can have more than one active profile at a time (typically each associated with a mode), so a single hit to this shot might result in this event being posted multiple times with different (profile) values.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

`profile` The name of the profile that was active when hit.

`state` The name of the state the profile was in when it was hit

Event is posted by *shots:*

(name)_(profile)_(state)_hit

MPF Event

The shot called (name) was just hit with the profile (profile) active in the state (state).

Note that there are four events posted when a shot is hit, each with variants of the shot name, profile, and current state, allowing you to key in on the specific granularity you need.

Also remember that shots can have more than one active profile at a time (typically each associated with a mode), so a single hit to this shot might result in this event being posted multiple times with different (profile) and (state) values.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

`profile` The name of the profile that was active when hit.

`state` The name of the state the profile was in when it was hit

Event is posted by *shots:*

(name)_(state)_hit

MPF Event

The shot called (name) was just hit while in the profile (state).

Note that there are four events posted when a shot is hit, each with variants of the shot name, profile, and current state, allowing you to key in on the specific granularity you need.

Also remember that shots can have more than one active profile at a time (typically each associated with a mode), so a single hit to this shot might result in this event being posted multiple times with different (profile) and (state) values.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

`profile` The name of the profile that was active when hit.

`state` The name of the state the profile was in when it was hit

Event is posted by *shots:*

shutdown

MPF Event

Posted when the machine is shutting down to give all modules a chance to shut down gracefully.

This event does not have any keyword arguments

single_player_ball_started

MPF Event

A new ball has started, and this is a single player game.

This event does not have any keyword arguments

slam_tilt

MPF Event

A slam tilt has just occurred.

This event does not have any keyword arguments

slide_(name)_active

MPF-MC Event

A slide called (name) has just become active, meaning that it’s now showing as the current slide. This is useful for things like the widget_player where you want to target a widget for a specific slide, but you can only do so if that slide exists. Slide names do not take into account what display they’re playing on, so be sure to create machine-wide unique names when you’re naming your slides.

This event does not have any keyword arguments

Event is posted by *slides:*

slide_(name)_created

MPF-MC Event

A slide called (name) has just been created.

This means that this slide now exists, but it's not necessarily the active (showing) slide, depending on the priorities of the other slides and/or what else is going on.

This is useful for things like the widget_player where you want to target a widget for a specific slide, but you can only do so if that slide exists.

Slide names do not take into account what display or slide frame they're playing on, so be sure to create machine-wide unique names when you're naming your slides.

This event does not have any keyword arguments

Event is posted by *slides:*

slide_(name)_removed

MPF-MC Event

A slide called (name) has just been removed.

This event is posted whenever a slide is removed, regardless of whether or not that slide was active (showing).

Note that even though this event is called "removed", it's actually posted as part of the removal process. (e.g. there are still some clean-up things that happen afterwards.)

Slide names do not take into account what display or slide frame they're playing on, so be sure to create machine-wide unique names when you're naming your slides.

This event does not have any keyword arguments

Event is posted by *slides:*

spinner_(name)_active

MPF Event

The idle spinner (name) was just hit and became active.

This event will post whenever a spinner switch is hit and the spinner is not already active.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

`label` The label of the switch that triggered the activation

Event is posted by *spinners:*

spinner_(name)_hit

MPF Event

The spinner (name) was just hit.

This event will post whenever a spinner switch is hit.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

hits The number of switch hits the spinner has had since it became active

label The label of the switch that was hit

Event is posted by *spinners:*

spinner_(name)_idle

MPF Event

The spinner (name) is now idle

This event will post whenever a spinner has not received hits and its idle_ms has timed out. If no idle_ms is defined, this event will not post.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

hits The number of switch hits the spinner had while it was active

Event is posted by *spinners:*

spinner_(name)_inactive

MPF Event

The spinner (name) is no longer receiving hits

This event will post whenever a spinner has not received hits and its active_ms has timed out.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

hits The number of switch hits the spinner had while it was active

Event is posted by *spinners:*

spinner_(name)_(label)_active

MPF Event

The idle spinner (name) was just hit and became active.

This event will post whenever a spinner switch is hit and the spinner is not already active, but only if labels are defined for the spinner.

This event does not have any keyword arguments

Event is posted by *spinners:*

spinner_(name)_(label)_hit

MPF Event

The spinner (name) was just hit on the switch labelled (label).

This event will post whenever a spinner switch is hit and labels are defined for the spinner

This event does not have any keyword arguments

Event is posted by *spinners:*

sw_(name)_active

MPF Event

The playfield called (name) was active, though a ball was just removed from it.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

balls The number of balls that were just removed from this playfield.

Event is posted by *playfields:*

sw_(tag)

MPF Event

Posted when a switch with this tag becomes active. Note that this will only be posted if there is an event handler for it or if debug is set to True on this switch for performance reasons.

This event does not have any keyword arguments

Event is posted by *switches:*

sw_(tag)_active

MPF Event

Posted when a switch with this tag becomes active. Note that this will only be posted if there is an event handler for it or if debug is set to True on this switch for performance reasons.

This event does not have any keyword arguments

Event is posted by *switches:*

sw_(tag)_inactive

MPF Event

Posted when a switch with this tag becomes inactive. Note that this will only be posted if there is an event handler for it or if debug is set to True on this switch for performance reasons.

This event does not have any keyword arguments

Event is posted by *switches:*

(name)_active

MPF Event

Posted when this switch becomes active. Note that this will only be posted if there is an event handler for it or if debug is set to True on this switch for performance reasons.

This event does not have any keyword arguments

Event is posted by *switches:*

(name)_inactive

MPF Event

Posted when this switch becomes inactive. Note that this will only be posted if there is an event handler for it or if debug is set to True on this switch for performance reasons.

This event does not have any keyword arguments

Event is posted by *switches:*

switch_(name)_active

MPF-MC Event

Posted on MPF-MC only (e.g. not in MPF) when the MC receives a BCP "switch" active command. Useful for video modes and graphical menu navigation. Note that this is not posted for every switch all the time, rather, only for switches that have been configured to send events to BCP.

This event does not have any keyword arguments

Event is posted by *switches:*

switch_(name)_inactive

MPF-MC Event

Posted on MPF-MC only (e.g. not in MPF) when the MC receives a BCP "switch" inactive command. Useful for video modes and graphical menu navigation. Note that this is not posted for every switch all the time, rather, only for switches that have been configured to send events to BCP.

This event does not have any keyword arguments

Event is posted by *switches:*

text_input_(key)_abort

MPF-MC Event

This event is posted by a *text_input* display widget when the entering process was aborted.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

text A string of the characters that were entered so far.

text_input_(key)_complete

MPF-MC Event

This event is posted by a *text_input* display widget when the entered text is finalized.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

text A string of the final characters that were entered.

tilt

MPF Event

The player has tilted.

This event does not have any keyword arguments

tilt_clear

MPF Event

Posted after a tilt, when the settling time has passed after the last tilt switch hit. This is used to hold the next ball start until the plumb bob has settled to prevent tilt throughs.

This event does not have any keyword arguments

tilt_warning

MPF Event

A tilt warning just happened.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

warnings The total number of warnings so far.

warnings_remaining The remaining number of warnings until a tilt.

tilt_warning_(number)

MPF Event

A tilt warning just happened. The number of this tilt warning is in the event name in the (number).

This event does not have any keyword arguments

(name)_active

MPF Event

Posted when one of the switches buttons has been active for `time`.

This event does not have any keyword arguments

Event is posted by *timed_switches:*

The event name can be changed by using the "events_when_active:" attribute.

(name)_released

MPF Event

Posted when one of the switches that has previously been active for more than `time` has been released.

This event does not have any keyword arguments

Event is posted by *timed_switches:*

The event name can be changed by using the "events_when_released:" attribute.

timer_(name)_complete

MPF Event

The timer named (name) has completed.

Note that this timer may reset and start again after this event is posted, depending on its settings.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

ticks The current tick number this timer is at.

ticks_remaining The number of ticks in this timer remaining.

Event is posted by *timers:*

timer_(name)_paused

MPF Event

The timer named (name) has paused.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

ticks The current tick number this timer is at.

ticks_remaining The number of ticks in this timer remaining.

Event is posted by *timers:*

timer_(name)_started

MPF Event

The timer named (name) has just started.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

ticks The current tick number this timer is at.

ticks_remaining The number of ticks in this timer remaining.

Event is posted by *timers:*

timer_(name)_stopped

MPF Event

The timer named (name) has stopped.

This event is posted any time the timer stops, whether it stops because it ended or because it was stopped early by some other event.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

ticks The current tick number this timer is at.

ticks_remaining The number of ticks in this timer remaining.

Event is posted by *timers:*

timer_(name)_tick

MPF Event

The timer named (name) has just counted down (or up, depending on its settings).

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

ticks The new tick number this timer is at.

ticks_remaining The new number of ticks in this timer remaining.

Event is posted by *timers:*

timer_(name)_time_added

MPF Event

The timer named (name) has just had time added to it.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

ticks The new tick number this timer is at.

ticks_added How many ticks were just added.

ticks_remaining The new number of ticks in this timer remaining.

Event is posted by *timers:*

timer_(name)_time_subtracted

MPF Event

The timer named (name) just had some ticks removed.

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

ticks The new current tick number this timer is at.

ticks_remaining The new number of ticks in this timer remaining.

ticks_subtracted How many ticks were just subtracted from this timer. (This number will be positive, indicating the ticks subtracted.)

Event is posted by *timers:*

twitch_bit_donation

MPF Event

A chat user has donated bits on Twitch

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

bits The number of bits donated

message Chat message text

user The chat user name who subscribed

twitch_chat_message

MPF Event

A chat message was received via Twitch

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

line_1 Split line 1

line_2 Split line 2

line_3 Split line 3

line_4 Split line 4

line_5 Split line 5

line_6 Split line 6

line_count The number of lines that the text splitter produced

message Full chat message text

user The chat user name who subscribed

twitch_command

MPF Event

A user typed a line that begins with ! or ?

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

command The text after the ! or ?

user The chat user who executed the command

twitch_raid

MPF Event

Another Twitch user has raided your channel

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

raid_count The count of viewers in the raid

raid_user The user who raided

twitch_subscription

MPF Event

A chat user has subscribed or resubscribed on Twitch

Keyword arguments

(See the *Conditional Events* guide for details for how to create entries in your config file that only respond to certain combinations of the arguments below.)

gift True if this sub was gifted by another user

message Chat message text

months The number of months that the user has been a subscriber

sub_plan The subscription tier (Prime, 1000, 2000, 3000)

sub_plan_name The streamer specific name for the sub tier

sub_recipient The user who is subscribing

subscriber_message The message the user typed when subscribing

user The chat user name who paid for the subscription

unexpected_ball_on_(name)

MPF Event

The playfield named (name) just had a switch hit, meaning a ball is on it, but that ball was not expected.

This event does not have any keyword arguments

Event is posted by *playfields:*

Device Indexes

achievement

See: *achievements:*

- *achievement_(name)_changed_state*
- *achievement_(name)_state_(state)*

ball_device

See: *ball_devices:*

- *balldevice_(name)_ball_count_changed*
- *balldevice_(name)_ball_eject_attempt*
- *balldevice_(name)_ball_eject_failed*
- *balldevice_(name)_ejecting_ball*
- *balldevice_(name)_ball_eject_success*
- *balldevice_(name)_broken*
- *balldevice_captured_from_(captures_from)*
- *balldevice_(name)_ball_enter*
- *balldevice_(name)_ball_entered*
- *balldevice_(name)_ball_missing*
- *balldevice_ball_missing*
- *balldevice_balls_available*

ball_hold

See: *ball_holds:*

- *ball_hold_(name)_held_ball*
- *ball_hold_(name)_full*
- *ball_hold_(name)_balls_released*

ball_save

See: *ball_saves:*

- *ball_save_(name)_enabled*
- *ball_save_(name)_disabled*
- *ball_save_(name)_timer_start*
- *ball_save_(name)_hurry_up*
- *ball_save_(name)_grace_period*
- *ball_save_(name)_saving_ball*

combo_switch

See: *combo_switches:*

- *(name)_one*
- *(name)_both*
- *(name)_inactive*
- *(name)_switches_1*
- *(name)_switches_2*

display

See: *displays:*

- *display_(name)_initialized*
- *display_(name)_ready*

diverter

See: *diverters:*

- *diverter_(name)_enabling*
- *diverter_(name)_disabling*
- *diverter_(name)_activating*
- *diverter_(name)_deactivating*

drop_target_bank

See: *drop_target_banks:*

- *drop_target_bank_(name)_down*
- *drop_target_bank_(name)_up*
- *drop_target_bank_(name)_mixed*

drop_target

See: *drop_targets:*

- *drop_target_(name)_down*
- *drop_target_(name)_up*

extra_ball_group

See: *extra_ball_groups:*

- *extra_ball_group_(name)_awarded*
- *extra_ball_group_(name)_lit*
- *extra_ball_group_(name)_unlit*
- *extra_ball_group_(name)_award_disabled*
- *extra_ball_group_(name)_lit_awarded*

extra_ball

See: *extra_balls:*

- *extra_ball_award_disabled*
- *extra_ball_(name)_award_disabled*
- *extra_ball_(name)_lit*
- *extra_ball_(name)_awarded*
- *extra_ball_awarded*

kickback

See: *kickbacks:*

- *kickback_(name)_fired*

machine_var

See: *machine_vars:*

- *machine_var_(name)*

magnet

See: *magnets:*

- *magnet_(name)_grabbing_ball*
- *magnet_(name)_grabbed_ball*
- *magnet_(name)_releasing_ball*
- *magnet_(name)_released_ball*
- *magnet_(name)_flinging_ball*
- *magnet_(name)_flinged_ball*

motor

See: *motors:*

- *motor_(name)_reached_(position)*

multiball_lock

See: *multiball_locks:*

- *multiball_lock_(name)_locked_ball*
- *multiball_lock_(name)_full*

multiball

See: *multiballs:*

- *multiball_(name)_started*
- *ball_save_(name)_timer_start*
- *multiball_(name)_hurry_up*
- *multiball_(name)_grace_period*
- *multiball_(name)_shoot_again*
- *multiball_(name)_lost_ball*
- *multiball_(name)_shoot_again_ended*
- *ball_save_(name)_add_a_ball_timer_start*
- *multiball_(name)_ended*

player_var

See: *player_vars:*

- *player_(name)*

playfield_transfer

See: *playfield_transfers:*

- *playfield_transfer_(playfield_transfer)_ball_transferred*

playfield

See: *playfields:*

- *(name)_ball_count_change*
- *(name)_active*
- *sw_(name)_active*
- *unexpected_ball_on_(name)*

score_reel

See: *score_reels:*

- *reel_(name)_advanced*

sequence_shot

See: *sequence_shots:*

- *(name)_hit*

shot_group

See: *shot_groups:*

- *(name)_complete*
- *(name)_(state)_complete*
- *(name)_hit*
- *(name)_(state)_hit*

shot

See: *shots:*

- *(name)_hit*
- *(name)_(profile)_hit*
- *(name)_(profile)_(state)_hit*
- *(name)_(state)_hit*

slide

See: *slides:*

- *slide_(name)_created*
- *slide_(name)_removed*
- *slide_(name)_active*

spinner

See: *spinners:*

- *spinner_(name)_hit*
- *spinner_(name)_inactive*
- *spinner_(name)_idle*
- *spinner_(name)_active*
- *spinner_(name)_(label)_hit*
- *spinner_(name)_(label)_active*

switch

See: *switches:*

- *(name)_active*
- *(name)_inactive*
- *sw_(tag)*
- *sw_(tag)_active*
- *sw_(tag)_inactive*
- *switch_(name)_active*
- *switch_(name)_inactive*

timed_switch

See: *timed_switches:*

- *(name)_active*
- *(name)_released*

timer

See: *timers:*

- *timer_(name)_started*
- *timer_(name)_stopped*

- *timer_(name)_paused*
- *timer_(name)_complete*
- *timer_(name)_time_added*
- *timer_(name)_time_subtracted*
- *timer_(name)_tick*

CHAPTER 27

Game Variables

Game variables allow you to query specifics while a game is in play.

Like player and machine variables, you can use the game variables in your config files and can be particularly useful for conditional arguments.

max_players

MPF Game variable

The maximum players currently allowed at one time.

num_players

MPF Game variable

The number of players currently playing.

balls_per_game

MPF Game variable

The number of balls per player, per game. This is usually 3 or 5.

balls_in_play

MPF Game variable

The current number of balls in play.

tilted

MPF Game variable

A boolean variable that will return 'True' if the game has been tilted.

slam_tilted

MPF Game variable

A boolean variable that will return 'True' if the game has been slam tilted.

CHAPTER 28

Machine Variables

MPF uses the concept of *machine variables* to track dynamically- created variables that apply on a machine-wide basis. Machine variables are similar in concept to *player variables*, except machine variables are machine-wide instead of per-player. Examples of things that are stored in machine variables include:

- The number of credits on the machine (if you're using the credits mode and not set to free play)
- The scores of the last game played (which are typically shown in the attract mode display loop)
- The names and scores of the high scores (which are also shown in the attract mode display loop and in the "status" screen when a player holds a flipper button in during a game).

Machine variables can be set to `persist`, meaning they are saved to disk and available to MPF the next time it boots up. (For example, if you first turn on a pinball machine, it will still show the scores of the last game played in the attract mode.) These machine variables are stored in the <your_machine_folder>/data/machine_vars.yaml file. Machine variables that are saved to disk can optionally be written with an expiration time which means they're cleared out if MPF boots after the time has passed. (For example, the number of credits on the machine might only persist for a few hours.)

Like player variables, you can use machine variables in your config files, particularly in text display widgets, to show things on your display.

If you want to use a machine variable in a slide player you can access it similarly to normal variables, you need to use the syntax `(machine|my_var_name)` where `my_var_name` obviously has to be replaced with your variable name.

Video about player and machine variables:

https://youtu.be/PUxEsNUGXPY

You can create your own machine variables in your configs. There are also several machine variables that are automatically created. Here's a list of the machine variables that are "built in" and available for use in your configs:

credit_units

MPF machine variable

How many credit units are on the machine. Note that credit units are not useful for display purposes since they represent the number of credits in a ration related to the lowest common denominator of the partial credit fraction. See the related *credits_string* and *credits_value* machine variables for more useful formats.

credits_denominator

MPF machine variable

The denominator portion of the total credits on the machine. For example, if the machine has 4 1/2 credits, this value is "2".

credits_numerator

MPF machine variable

The numerator portion of the total credits on the machine. For example, if the machine has 4 1/2 credits, this value is "1".

credits_string

MPF machine variable

Holds a displayable string which shows how many credits are on the machine. For example, "CREDITS: 1". If the machine is set to free play, the value of this string will be "FREE PLAY".

You can change the format and value of this string in the `credits:` section of the machine config file.

credits_value

MPF machine variable

The human readable string form which shows the number value of how many credits are on the machine, including whole and fractional credits, for example "1" or "2 1/2" or "3 3/4".

If you want the full string with the word "CREDITS" in it, use the "credits_string" machine variable.

credits_whole_num

MPF machine variable

The whole number portion of the total credits on the machine. For example, if the machine has 3 1/2 credits, this value is "3".

fast_(x)_firmware

MPF machine variable

Holds the version number of the firmware for the processor on the FAST Pinball controller that's connected. The "x" is replaced with either "dmd", "net", or "rgb", one for each processor that's attached.

fast_(x)_model

MPF machine variable

Holds the model number of the board for the processor on the FAST Pinball controller that's connected. The "x" is replaced with either "dmd", "net", or "rgb", one for each processor that's attached.

(high_score_category)(position)_label

MPF machine variable

The "label" of the high score for that specific score category and position. For example, `score1_label` holds the label for the #1 position of the "score" player variable (which might be "GRAND CHAMPION").

(high_score_category)(position)_name

MPF machine variable

Holds the player's name (or initials) for the high score for that category and position.

(high_score_category)(position)_value

MPF machine variable

Holds the numeric value for the high score for that category and position.

lisy_api_version

MPF machine variable

LISY API version.

lisy_hardware

MPF machine variable

Connected LISY hardware (I.e. LISY1, LISY80 or APC).

lisy_version

MPF machine variable

LISY version.

mc_extended_version

MPF machine variable

Extended version of MC. This is set after MC got connected. Contains BCP and show version numbers.

mc_version

MPF machine variable

Version of MC. This is set after MC got connected.

mpf_extended_version

MPF machine variable

Extended version string for MPF. Contains show and bcp version as well.

mpf_version

MPF machine variable

Full version string for MPF.

p_roc_hardware_version

MPF machine variable

Holds the hardware version number of the P-ROC or P3-ROC controller that's attached to MPF.

p_roc_revision

MPF machine variable

Holds the firmware revision number of the P-ROC or P3-ROC controller that's attached to MPF.

p_roc_version

MPF machine variable

Holds the firmware version number of the P-ROC or P3-ROC controller that's attached to MPF.

pkone_firmware

MPF machine variable

Holds the version number of the firmware for the Penny K Pinball PKONE controller that's connected.

pkone_hardware

MPF machine variable

Holds the model name and hardware revision number of the Penny K Pinball PKONE controller board that's connected.

platform

MPF machine variable

A single string identifying the underlying platform with as much useful information as possible.

platform_machine

MPF machine variable

Architecture of your machine (32bit/64bit).

platform_release

MPF machine variable

Release of your operating system.

platform_system

MPF machine variable

Your system (Linux/Windows/Mac).

platform_version

MPF machine variable

Version of your operating system.

player(x)_score

MPF machine variable

Holds the numeric value of a player's score from the last game. The "x" is the player number, so this actual machine variable is `player1_score` or `player2_score`.

Since these are machine variables, they are maintained even after a game is over. Therefore you can use these machine variables in your attract mode display show to show the scores of the last game that was played.

These machine variables are updated at the end of the game, and they persist on disk so they are restored the next time MPF starts up.

python_version

MPF machine variable

Python version.

Related Events

- *machine_var_(name)*

CHAPTER 29

Player Variables Reference

Here's a list of all the different "built in" *player variables* that MPF uses.

You can use these in your config files to trigger game logic or to display as text on your display.

Video about player and machine variables:

https://youtu.be/PUxEsNUGXPY

Note that you can also create your own player variables in your configs, and most likely your machine will have several orders of magnitude more player variables than this list here.

That said, here's a list of the "built in" player variables and how they work:

index

MPF player variable

The index of this player, starting with 0. For example, Player 1 has an index of 0, Player 2 has an index of 1, etc.

If you want to get the player number, use the "number" player variable instead.

ball

MPF player variable

The ball number for this player. If a player gets an extra ball, this number won't change when they start the extra ball.

extra_ball_(name)_awarded

MPF player variable

The number of times this extra ball has been awarded to the player in this game. Note that the default max is one (meaning that each extra ball can be awarded once per game), so this value will only be 0 or 1 unless you change the max setting for this extra ball.

extra_balls

MPF player variable

The number of remaining extra balls which the player will be awarded after draining his current ball.

config_section: counters, accruals, sequences <logic_block_state config_section: counters, accruals, sequences>

> (mode)_(timer)_tick <mode_timer_tick> number <number> random_(x).(y) <random_x.y> restart_modes_on_next_ball <restart_modes_on_next_ball> score <score>

Related Events

- *player_(name)*

CHAPTER 30

Log and Error Descriptions

CFE-coils-1: Driver must have a number

Related Config File Sections
coils:

This error occurs when MPF loads a coil which is has an empty number or misses a number entry. Unfortunately, hardware needs a switch number to address your coil and it cannot continue without a number.

Examples

Physical Coils

This is how a coil should look:

```
coils:
  your_coil:
    number: 1
```

The actual number depends on your hardware platform. See the *How to configure "number:" settings* guide for details.

Virtual Coils

Sometimes you did not wire up a coil but you know that you will need it later. This is a problem for your physical hardware controller but you can tell MPF to use the `virtual` hardware platform for one particular coil:

```
coils:
  your_virtual_coil:
    number:
    platform: virtual
```

In this case the number can be empty.

Need more help troubleshooting?

Have a look at our *Troubleshooting* section. It might give you some hints for certain classes of problems.

What if this did not fix your problem?

Please tell us about your error in the MPF Users Google Group and we might be able to update this page afterwards. Or even better: *You help us to update it afterwards*.

Is something missing here? Do you have a helpful hint for others experiencing this error?

Please *create a Pull Request and add it*. Alternatively, please tell us in the MPF Users Google Group.

Related How To guides

- *Coils (Solenoids)*

CFE-ConfigValidator-1: Section not valid outside of game modes

This error occurs when MPF needs to reference player variables in a device but you defined the device in a non-game *mode* (i.e. with `game_mode: false`) such as the attract mode. Game modes will always end when the game ends. Non-game modes can run all the time but they should not access player variables as they do not exist outside of a game. Certain devices enforce the latter.

Examples

For instance, a *counter* can store its state in a player variable which is only possible in a game mode:

```
##! mode: my_game_mode
mode:
  start_events: ball_started
  stop_events: ball_stopped
  game_mode: true    # this is the default

counters:
  counter_per_player:
    count_events: count_up
    persist_state: true
```

However, if you set `persist_state: False` in your counter it can also be used outside of a mode:

```
##! mode: attract
mode:
  game_mode: false

counters:
  counter_outside_of_a_game:
    count_events: count_up
    persist_state: false
```

Those settings are described in the *config reference* of your device.

Common Pitfalls

Variable_Players

Variable player will by default use player variables. However, if you use `action: add_machine` or `action: set_machine` you can also use it to add/set machine variables which work in non-game modes.

Attract Mode

Attract mode only runs outside of a game so you cannot reference player here. However, you can use machine variables

Match Mode and High Score Mode

Those modes run at game end and are technically no longer game modes. Therefore, you cannot reference a player here. You might want to put your stuff into a custom mode which run at ball end (but not game end) instead (i.e. the bonus mode). Alternatively, you might want to use machine variables instead of player variables.

Need more help troubleshooting?

Have a look at our *Troubleshooting* section. It might give you some hints for certain classes of problems.

What if this did not fix your problem?

Please tell us about your error in the MPF Users Google Group and we might be able to update this page afterwards. Or even better: *You help us to update it afterwards*.

Is something missing here? Do you have a helpful hint for others experiencing this error?

Please *create a Pull Request and add it*. Alternatively, please tell us in the MPF Users Google Group.

Related How To guides

- *Logic Blocks*
- *config reference*

CFE-ConfigValidator-2: Your config contains a value for the setting, but this is not a valid setting name

This error occurs when MPF does not know a setting you specified in a device.

Examples

For instance, a *switch* knows certain settings:

```
switches:
  s_flipper_left:
    number: 1
    label: My Left Flipper Switch Example
    tags: some_custom_tag
```

You can see which settings are allowed in the *config reference* of your device.

Common Pitfalls

Typos

The most common source for this kind of error are typos. Check the name of your referenced device with the setting. Casing matters here (i.e. upper/lower case). Using an *IDE with the MPF language server* can help here.

Mixing Devices

Maybe you accidentially copied config attributes from a different type of devices? Double check if you refered to the documentation of the correct device. If you find incorrect documentation please tell us in the forum.

Incorrect Indent

With nested configs (i.e. slide_player or widget_player) you might have used an option which should be indented one level further or one level less. This can sometimes be a bit tricky. Using an *IDE with the MPF language server* can help here.

Running Config from a different MPF Version

Sometimes MPF config specifications change. Check if your MPF version fits the config. If in doubt check the *config reference* for your device.

Need more help troubleshooting?

Have a look at our *Troubleshooting* section. It might give you some hints for certain classes of problems.

What if this did not fix your problem?

Please tell us about your error in the MPF Users Google Group and we might be able to update this page afterwards. Or even better: *You help us to update it afterwards*.

Is something missing here? Do you have a helpful hint for others experiencing this error?

Please *create a Pull Request and add it*. Alternatively, please tell us in the MPF Users Google Group.

Related How To guides

- *config reference*

CFE-ConfigValidator-4: Invalid Validator in config spec

This error occurs when MPF does not understand the config specification for a device. Unless you created custom config specs in a mode, (external) platform or custom device, this is certainly a bug in MPF. Please report this in our forum in that case!

Need more help troubleshooting?

Have a look at our *Troubleshooting* section. It might give you some hints for certain classes of problems.

What if this did not fix your problem?

Please tell us about your error in the MPF Users Google Group and we might be able to update this page afterwards. Or even better: *You help us to update it afterwards*.

Is something missing here? Do you have a helpful hint for others experiencing this error?

Please *create a Pull Request and add it*. Alternatively, please tell us in the MPF Users Google Group.

Related How To guides

- *config reference*

CFE-ConfigValidator-6: Device not found in section in your config

This error occurs when MPF does not find a device which is referenced by one of your settings in your config.

Examples

For instance, a flipper device references a switch and a coil:

```
switches:
  s_flipper_left:
    number:
coils:
  c_flipper_left:
    number:
flippers:
  left_flipper:
    main_coil: c_flipper_left
    activation_switch: s_flipper_left
```

You can see to which type of device a setting references in the *config reference* of your device. For this instance, check the *flipper config reference* and you will find that `main_coil` references a *coil* and `activation_switch` references a *switch*.

Common Pitfalls

Typos

The most common source for this kind of error are typos. Check the name of your referenced device with the setting. Casing matters here (i.e. upper/lower case). Using an *IDE with the MPF language server* can help here.

Copy and Paste

We all do this and there is nothing wrong with copying configs from the docs. Almost all examples in the docs are tested and should not give this kind of error. However, sometimes we hide certain devices in the docs (i.e. switches and coils which are referenced by an examplary flipper device as above). This is done to improve readability but when copying those examples you might get this error. Nevertheless, you can click "Click to show full config" below all examples to see the full tested example which is tested to work in the MPF version corresponding to the docs.

Running Config from a different MPF Version

Sometimes MPF config specifications change. Check if your MPF version fits the config. If in doubt check the *config reference* for your device.

Referencing a different type of device

If you reference a different device MPF won't find it and show this error. Check the *config reference* of your device to see which device is expected or setup your *IDE with the MPF language server*.

Need more help troubleshooting?

Have a look at our *Troubleshooting* section. It might give you some hints for certain classes of problems.

What if this did not fix your problem?

Please tell us about your error in the MPF Users Google Group and we might be able to update this page afterwards. Or even better: *You help us to update it afterwards*.

Is something missing here? Do you have a helpful hint for others experiencing this error?

Please *create a Pull Request and add it*. Alternatively, please tell us in the MPF Users Google Group.

Related How To guides

- *config reference*

CFE-ConfigValidator-9: Required setting is missing from section in your config

This error occurs when MPF does not find a required setting in one of your config sections.

Examples

For instance, every switch has to have a number in MPF:

```
switches:
  s_ball_switch1:
    number: 1
```

You can see which settings are required in the *config reference* of your device. For this instance, check the *switch config reference* and you will find that only `number` is a required setting.

Common Pitfalls

Omitting one of the required settings

If you omit on of the required settings you will see this error. To this this browse to the *config reference* of your device and add all the required settings. Alternatively, you could use your *IDE with the MPF language server* to auto-complete all required settings.

Need more help troubleshooting?

Have a look at our *Troubleshooting* section. It might give you some hints for certain classes of problems.

What if this did not fix your problem?

Please tell us about your error in the MPF Users Google Group and we might be able to update this page afterwards. Or even better: *You help us to update it afterwards*.

Is something missing here? Do you have a helpful hint for others experiencing this error?

Please *create a Pull Request and add it*. Alternatively, please tell us in the MPF Users Google Group.

Related How To guides

- *config reference*

CFE-ConfigValidator-12: Item is not a dict

This error occurs when MPF expects a dictionary in a config setting but found something else.

Examples

For instance, `show_tokens` in a *show_player* has to be a dictionary:

```
show_player:
  some_event:
    your_show_name:
      show_tokens:
        dict_key1: "dict_value1"
        dict_key2: "dict_value2"
```

You can see which settings are dicts in the *config reference* of your device.

Common Pitfalls

Using a List instead of a Dict

This is a list in yaml:

```
your_setting:
  - item1_in_list
  - item2_in_list
```

This is a dictionary:

```
your_setting:
  key1_in_dict: value1_in_dict
  key2_in_dict: value2_in_dict
```

This is a list of dictionaries (used in shows for example):

```
your_setting:
  - key1_in_dict_in_list1: value1_in_dict_in_list1
  - key1_in_dict_in_list2: value1_in_dict_in_list2
    key2_in_dict_in_list2: value2_in_dict_in_list2
```

Incorrect Indent

With nested configs (i.e. show_player, slide_player or widget_player) you might have used an option which should be indented one level further or one level less. This can sometimes be a bit tricky. Using an *IDE with the MPF language server* can help here.

Need more help troubleshooting?

Have a look at our *Troubleshooting* section. It might give you some hints for certain classes of problems.

What if this did not fix your problem?

Please tell us about your error in the MPF Users Google Group and we might be able to update this page afterwards. Or even better: *You help us to update it afterwards*.

Is something missing here? Do you have a helpful hint for others experiencing this error?

Please *create a Pull Request and add it*. Alternatively, please tell us in the MPF Users Google Group.

Related How To guides

- *config reference*

CFE-ConfigValidator-13: Cannot convert value to boolean

This error occurs when MPF expects a boolean value (i.e. `true` or `false`) for a config setting but got a value of a different type.

Examples

For instance, the debug setting for a *switch* is a boolean:

```
switches:
  s_flipper_left:
    number: 1
    debug: true   # we want all the details about this switch in the logs
```

You can see which settings are boolean in the *config reference* of your device.

Common Pitfalls

Widget Animations Repeat

In MPF versions before 0.53 `repeat` in widgets has been an integer which has been converted to boolean internally. A lot of examples (and the tutorial) contained `repeat: -1`. You need to change this to `repeat: false` to fix this error.

Using Quotes

If you use `debug: "false"` (with quotes around `false`) MPF will not recognize `false` as a boolean but as a string. Remove the quotes to fix this.

Need more help troubleshooting?

Have a look at our *Troubleshooting* section. It might give you some hints for certain classes of problems.

What if this did not fix your problem?

Please tell us about your error in the MPF Users Google Group and we might be able to update this page afterwards. Or even better: *You help us to update it afterwards*.

Is something missing here? Do you have a helpful hint for others experiencing this error?

Please *create a Pull Request and add it*. Alternatively, please tell us in the MPF Users Google Group.

Related How To guides

- *config reference*

CFE-DeviceManager-3: Device does not have a valid config. Expected a dictionary.

This error occurs when MPF expects a dictionary in a config of a device but found something else.

Examples

For instance, the settings of a *switch* are a dictionary (switches -> s_flipper_left).

```
switches:
  s_flipper_left:
    number: 1
    label: My Left Flipper
```

Common Pitfalls

Forgetting the Device Name

This error usually occurs when you omit the device name. For example if you omit `s_flipper_left` this would look like this:

```
# BROKEN CONFIG
switches:
  number: 1
  label: My Left Flipper
```

Here MPF would see two switches with the names `number` and `label`. Each of them has an invalid config (just a single value but not a dictionary).

YAML Formatting Issues

See *CFE-ConfigValidator-12: Item is not a dict* for more general common pitfalls.

Need more help troubleshooting?

Have a look at our *Troubleshooting* section. It might give you some hints for certain classes of problems.

What if this did not fix your problem?

Please tell us about your error in the MPF Users Google Group and we might be able to update this page afterwards. Or even better: *You help us to update it afterwards*.

Is something missing here? Do you have a helpful hint for others experiencing this error?

Please *create a Pull Request and add it*. Alternatively, please tell us in the MPF Users Google Group.

Related How To guides

- *CFE-ConfigValidator-12: Item is not a dict*
- *config reference*

CFE-show-1: Show does not appear to be a valid show config

Related Config File Sections
shows:

This error occurs when MPF loads a show which is not a list of steps. There are two ways to add shows to your machine: either as file or inside your config. Both can happen inside a mode or machine-wide inside your global config folder.

Examples

File Shows

This is how a file show should look:

```
##! show: flash_red
#show_version=5
- duration: 1
  lights:
    led1: red
- duration: 1
  lights:
    led1: off
```

Please note that there can be only one show per dedicated show file as MPF uses the filename as show name. See *Creating standalone show files* for details.

Config Shows

This is how a show inside your config should look:

```
shows:
  flash_red:
    - duration: 1
      lights:
        led1: red
    - duration: 1
      lights:
        led1: off
```

See *Shows in files versus shows in configs* for details.

Common Pitfalls

Multiple shows inside one file show

This is NOT valid as file show:

```
# INVALID FILE SHOW
flash_red:
  - duration: 1
    # [...]
flash_blue:
  - duration: 1
    # [...]
```

Instead you have to create two files `flash_red.yaml` and `flash_blue.yaml`.

Missing hyphen for your step

You might have missed the hyphon in front of your first step (or in front of all steps):

```
# INVALID FILE SHOW
#show_version=5
duration: 1    # note the missing dash here
lights:
  led1: red
```

The same can happen in config shows:

```
# INVALID CONFIG SHOW
shows:
  flash_red:
    duration: 1   # hyphen missing here
    lights:
      led1: red
```

This often happens with one step shows. See above for working examples.

Need more help troubleshooting?

Have a look at our *Troubleshooting* section. It might give you some hints for certain classes of problems.

What if this did not fix your problem?

Please tell us about your error in the MPF Users Google Group and we might be able to update this page afterwards. Or even better: *You help us to update it afterwards*.

Is something missing here? Do you have a helpful hint for others experiencing this error?

Please *create a Pull Request and add it*. Alternatively, please tell us in the MPF Users Google Group.

Related How To guides

- *Shows*

CFE-Smart_Virtual_Platform-1: Switch used in virtual_platform_start_active_switches was not found in switches section

Related Config File Sections
switches:
virtual_platform_start_active_switches:

This error occurs when you use a switch in `virtual_platform_start_active_switches` which is not defined in your `switches` section.

Examples

This is how it should look:

```
switches:
  s_ball_switch1:
    number:
  s_ball_switch2:
    number:
  s_ball_switch3:
    number:
# Two switches should be active at start
virtual_platform_start_active_switches:
  - s_ball_switch1
  - s_ball_switch2
```

Alternatively, this could be a comma separated list:

```
switches:
  s_ball_switch1:
    number:
  s_ball_switch2:
    number:
  s_ball_switch3:
    number:
# Two switches should be active at start
virtual_platform_start_active_switches: s_ball_switch1, s_ball_switch2
```

Common Pitfalls

Using spaces instead of commas

In MPF versions before 0.54 you could also use spaces instead of commas. Even though this syntax was never officially supported in lists it still was supported code. This was also used in previous versions of the documentation and the tutorial.

```
# INVALID SYNTAX
virtual_platform_start_active_switches: s_ball_switch1 s_ball_switch2  # note the space instead of a␣
↪comma
```

To fix this turn it into one of the two syntaxes above. See *How to add lists to config files* for details.

Need more help troubleshooting?

Have a look at our *Troubleshooting* section. It might give you some hints for certain classes of problems.

What if this did not fix your problem?

Please tell us about your error in the MPF Users Google Group and we might be able to update this page afterwards. Or even better: *You help us to update it afterwards*.

Is something missing here? Do you have a helpful hint for others experiencing this error?

Please *create a Pull Request and add it*. Alternatively, please tell us in the MPF Users Google Group.

Related How To guides

- *virtual_platform_start_active_switches:*
- *How to add lists to config files*

CFE-Virtual_Platform-1: Switch used in virtual_platform_start_active_switches was not found in switches section

See *CFE-Smart_Virtual_Platform-1: Switch used in virtual_platform_start_active_switches was not found in switches section* which is exactly the same error.

Need more help troubleshooting?

Have a look at our *Troubleshooting* section. It might give you some hints for certain classes of problems.

What if this did not fix your problem?

Please tell us about your error in the MPF Users Google Group and we might be able to update this page afterwards. Or even better: *You help us to update it afterwards*.

Is something missing here? Do you have a helpful hint for others experiencing this error?

Please *create a Pull Request and add it*. Alternatively, please tell us in the MPF Users Google Group.

Log-SwitchController-1: Received duplicate switch state for switch

Related Config File Sections
switches:

MPF expects to get only state changes from platforms. That is part of the platform interface contract. This warning indicates that the contract is violated (i.e. because MPF got a switch close but the switch has been closed before). This might indicate bugs in the platform firmware, our platform interface or the communication in between.

MPF handles this gracefully so there is no need to worry. It will just ignore the second hit and carry on.

There are conditions where you will see this. Our smart virtual platform will sometimes trigger this. Those are kind of bugs. Usually harmless but we will fix them if you report them.

Additionally, you can trigger those warnings if you use more than source of switch states at once for the same switch. That could be any two of a hardware platform, MPF monitor or keyboard mappings.

Lastly, the P-Roc is known for sending switches twice when using debounced switches. This has to do with its internal state machine and is usually harmless.

Need more help troubleshooting?

Have a look at our *Troubleshooting* section. It might give you some hints for certain classes of problems.

What if this did not fix your problem?

Please tell us about your error in the MPF Users Google Group and we might be able to update this page afterwards. Or even better: *You help us to update it afterwards*.

Is something missing here? Do you have a helpful hint for others experiencing this error?

Please *create a Pull Request and add it*. Alternatively, please tell us in the MPF Users Google Group.

RE-MPF-MC_BCP_Server-1: Failed to bind BCP Socket to localhost on port 5050

See *RE-MPF_BCP_Server-1: Failed to bind BCP Socket to 127.0.0.1 on port 5051* which is exactly the same error.

Need more help troubleshooting?

Have a look at our *Troubleshooting* section. It might give you some hints for certain classes of problems.

What if this did not fix your problem?

Please tell us about your error in the MPF Users Google Group and we might be able to update this page afterwards. Or even better: *You help us to update it afterwards*.

Is something missing here? Do you have a helpful hint for others experiencing this error?

Please *create a Pull Request and add it*. Alternatively, please tell us in the MPF Users Google Group.

RE-MPF_BCP_Server-1: Failed to bind BCP Socket to 127.0.0.1 on port 5051

Related Config File Sections
bcp:

This error occurs when MPF cannot bind the port 5051 for incoming BCP connections. The same error can occur in MC when it cannot bind port 5050.

Common Pitfalls

Another Application is Running on that Port

Yahoo Messager uses 5050 and some Symantec application uses 5051. However, there might be other applications such a IIS which can also use those ports. Stop those applications or change the port in the *bcp config section*.

Firewalls and Antivirus Protection Soltions

Some firewalls might prevent MPF from binding ports. Also antivirus or threat protection software might do that. Try if disabling those help. If it helps see if you can add an exception for MPF.

Need more help troubleshooting?

Have a look at our *Troubleshooting* section. It might give you some hints for certain classes of problems.

What if this did not fix your problem?

Please tell us about your error in the MPF Users Google Group and we might be able to update this page afterwards. Or even better: *You help us to update it afterwards*.

Is something missing here? Do you have a helpful hint for others experiencing this error?

Please *create a Pull Request and add it*. Alternatively, please tell us in the MPF Users Google Group.

Related How To guides

- *bcp:*

RE-P-Roc-1 - Known Firmware Bug in P/P3-Roc

Related Config File Sections
p_roc:

This error occurs when you try to use `pulse_power` on drivers on the P3-Roc with firmware 2.14 or earlier and enable a rule with hold.

This can be solved by either removing `pulse_power` from the coil in question or by upgrading the firmware. Firmware can be obtained from the Multimorphic Member Area.

See *How to update the Firmware of the P-Roc or P3-Roc* for details about the upgrade process.

Need more help troubleshooting?

Have a look at our *Troubleshooting* section. It might give you some hints for certain classes of problems.

What if this did not fix your problem?

Please tell us about your error in the MPF Users Google Group and we might be able to update this page afterwards. Or even better: *You help us to update it afterwards*.

Is something missing here? Do you have a helpful hint for others experiencing this error?

Please *create a Pull Request and add it*. Alternatively, please tell us in the MPF Users Google Group.

Related How To guides

- *How to configure Multimorphic (P-ROC & P3-ROC) hardware*
- *Troubleshooting P-Roc/P3-Roc*
- *How to update the Firmware of the P-Roc or P3-Roc*

RE-P-Roc-2 - Communication with P/P3-Roc broke down

Related Config File Sections
p_roc:

In your log you will probably find a line such as:

```
OSError: Error in WriteData: wrote 0 of 8 bytes
```

This error occurs when `pinproc` (the library MPF uses to talk to the P/P3-Roc) reports an error while talking to the P/P3-Roc via USB. This is most likely a bad cable or a power supply issue. See *Troubleshooting P-Roc/P3-Roc* for potential causes and solutions.

Need more help troubleshooting?

Have a look at our *Troubleshooting* section. It might give you some hints for certain classes of problems.

What if this did not fix your problem?

Please tell us about your error in the MPF Users Google Group and we might be able to update this page afterwards. Or even better: *You help us to update it afterwards*.

Is something missing here? Do you have a helpful hint for others experiencing this error?

Please *create a Pull Request and add it*. Alternatively, please tell us in the MPF Users Google Group.

Related How To guides

- *Troubleshooting P-Roc/P3-Roc*
- *Wiring and Connectors in Pinball Machines*
- *Voltages and Power*

RE-P-Roc-3 - Failed to Import Pinproc

Related Config File Sections
p_roc:

This error occurs when MPF cannot load the `pinproc` library. See *Troubleshooting P-Roc/P3-Roc* for potential causes and solutions.

Need more help troubleshooting?

Have a look at our *Troubleshooting* section. It might give you some hints for certain classes of problems.

What if this did not fix your problem?

Please tell us about your error in the MPF Users Google Group and we might be able to update this page afterwards. Or even better: *You help us to update it afterwards*.

Is something missing here? Do you have a helpful hint for others experiencing this error?

Please *create a Pull Request and add it*. Alternatively, please tell us in the MPF Users Google Group.

Related How To guides

- *Troubleshooting P-Roc/P3-Roc*
- *How to use install drivers for the P-ROC / P3-ROC*

CHAPTER 31

Developer Documentation

We talk a lot about how you don't have to be an experienced software developer to use MPF. However, if you are an experienced developer, there are a few ways you can leverage your coding knowledge:

- You can add custom code to your machine for parts of your game where you'd rather write "real" code versus using config files.
- You can add custom code to handle unique and one-off hardware.
- You can write Python-based unit tests to test your machine.
- You can extend MPF to add features or to support new types of hardware.

Instructions for all of this, as well as an API reference, is available at the MPF Developer Documentation website:

http://developer.missionpinball.org

CHAPTER 32

About the MPF Documentation

If you'd like to help write or improve this documentation (even if it's a simple typo correction), see the *Contributing to the MPF documentation* guide for details.

MPF documentation authors

This MPF documentation was written by:

- Brian Madden (brian@missionpinball.org)
- Gabe Knuth (gabe@missionpinball.org)
- Quinn Capen (qcapen@gmail.com)
- Isaac Csandl (isaac.csandl@me.com)
- Jeremy Edwards (pinman2020@gmail.com)
- Jan Kantert (jan-mission-pinball@kantert.net)
- Ryan Richardson
- Anythony Winkle
- Dan Howard
- Mike O'Rourke
- Chris Benn
- Tim Wendt
- Travis Martin (travisbmartin@gmail.com)
- Coleman O Martin (colemanomartin@hotmail.com)
- Kevin Lee Drum (kevinleedrum@gmail.com)

- Aaron Matthies
- Jimmy Lipham
- Kyle Nahas
- Thomas Fulenwider (cobra18t)
- Brandon Hill (bghill.dev@gmail.com)
- Dave Ensminger
- Matt Kemp (matt@blz.co)
- Charles Duncan (nullbuilds)

Want to help with the docs? See our *Contributing to MPF's Documentation* page for details. It's easy!

MPF license & copyright

The Mission Pinball Framework code and all documentation is licensed in a way that basically means you can do whatever you want with it. The only real caveat is that you use it at your own risk, and we don't provide any warranties.

The code is licensed under the MIT license, and the documentation is licensed via Creative Commons Attribution 4.0 International (CC BY 4.0).

It a nutshell, you can use MPF and the docs however you want. You can use MPF in a commercial product. You can make changes to it, and you don't have to share the changes back with the community if you don't want to. You can make derivative works, sell it, build a business on it, etc. Go nuts!

At the end of the day, we created MPF because we want to see more pinball in the world, so we didn't put any restrictions on what you can do with it because we don't want anyone to hesitate jumping into the amazing world of pinball!

Help us to write it

Congratulations you found an opportunity to improve the documentation! If you are up to it write a few sentences, add an example or an image. Any help is welcome and don't be afraid we will review your change so you cannot accidentally break anything. Still interested? Then proceed to our guide on *How to contribute to MPF Docs*.

CHAPTER 33

MPF FAQ

FAQ: General

Why does this project exist?

The Mission Pinball Framework was started in 2014 by Brian Madden and Gabe Knuth. Both of them had dreamed of building their own pinball machines for years, and in 2013, they discovered the P-ROC and the wonderful community of home brew pinball builders and hackers.

The P-ROC pinball control system works with an open source project called `pyprocgame` which is a Python-based game framework. Pyprocgame is great, but it's pretty basic. (It's more of a pinball development environment versus a complete framework.) One of the challenges we saw was that people kept on having to "reinvent the wheel" with each game they built. After reading forum posts about "How do you write code for a trough?" about ten times, we thought, "Why isn't there a framework that just 'does that' for you?"

Pyprocgame also requires everything to be written in Python code, and we found that a lot of people who wanted to build their own pinball machines weren't software developers. So we thought it would be cool to create a framework where the majority of the "programming" could be done with text-based configuration files.

So in June 2014, we decided to start building the Mission Pinball Framework.

Around the same time, FAST Pinball came onto the market to offer an alternative control system to the P-ROC and P3-ROC. At that we thought, "Great, let's make the Mission Pinball Framework so that's it's hardware-independent and can work with the FAST Pinball or P-ROC systems (plus any other future systems that came out).

Isn't using config files limiting?

Finding the balance between "config files" and "real programming" is an age-old battle. We have a guide called *Config files versus "real" programming* which explains this in more detail, including our perspective on it and why we decided to make config files the focus on MPF.

Can I mix "real" code in with MPF config files?

Yes! See developer.missionpinball.org for details and examples.

Where does the name come from?

Brian lived in San Francisco's "Mission" neighborhood when MPF was first created. There are a lot of "Mission" things here, Mission Bowling, Mission Coffee, Mission Ice Cream. . . So we thought "Mission Pinball" had a great ring to it!

What pinball hardware does MPF work with?

The complete hardware compatibility list is *here*.

Who's behind this?

Even though MPF was started by Brian Madden and Gabe Knuth, our team has grown to involve lots of people. See the AUTHORS file in the MPF package for the latest list.

Is MPF stable?

MPF is open source software that is not yet at a 1.0 release. However we've been working on it since 2014, and several complete pinball machines have been built using it.

Furthermore, when we find crashes, we fix them. If you look at the list of commits (code additions, changes, and fixes that we check in) on GitHub, you'll see that we're busy with dozens of commits per week!

Is MPF beta? When will v1 be released?

MPF is open source and continuously developed. We're currently say, "Yes, it's beta" since we are not yet at a 1.0 release. However we release new versions every few months and don't expect that to change anytime soon.

We do expect to get to a 1.0 release at some point, but we don't have a specific time-frame for that. The important thing is to look at the code commit history and to notice that MPF is being very actively developed!

How can I download the documentation and read it offline?

Click the "Read the Docs" link in the lower-left corner of any page of the MPF documentation on docs.missionpinball.org for links to PDF, HTML, and Epub versions of the documentation.

What other options are there besides MPF?

While we think MPF is awesome, our main goal is to see more pinball in the world! Since all of us are working on MPF in our spare time (and not being paid for it), we won't be offended if you don't use MPF. Just please create more pinball!

At this time, if you don't want to use MPF, there are a few other options:

- pyprocgame (P-ROC/P3-ROC only; website defunct)
- PyProcGameHD+SkeletonGame (P-ROC/P3-ROC only, adds HD graphics and more to pyprogame)
- Open Pinball Project framework (Open Pinball Project hardware only)
- Rampant Slug Framework (P-ROC/P3-ROC only; website defunct)
- FreeWPC (WPC hardware only, lets you write new code in C, burn it to ROMS, and run it on original WPC hardware)

FAQ: Installation

How do I get started?

Start with the *Start Here* link in the menu on the left. That will explain an overview of how MPF works and then lead you through the features, the tutorial, and so on.

What are the prerequisites?

If you just want to start playing with MPF, you do not need a physical pinball machine. In fact we have a graphical tool (the MPF Monitor) which simulates a real pinball machine, so you can probably build an entire game without having an actual pinball machine.

If you want to use a real pinball machine (or build a real machine), you need to pick a pinball control system. (We have a list of supported control systems *here*.) If you want to get started as cheaply as possible, the Open Pinball Project hardware is open source which you can build yourself. You can probably build all the hardware you need for under $100.

What computer hardware do I need?

MPF supports Windows, Mac, and Linux, so pretty much any computer is fine. Most people do their development of MPF on whatever computer they use in their daily lives, then when they're getting close to done with their machine, they install a dedicated computer (or even a Raspberry Pi) in their machine to run MPF.

What Python version can I use with MPF?

Your need Python 3.5 or 3.6. Python 3.4 is end of life and will no longer be supported. Python 3.7 and newer are not yet supported. We walk you through getting Python installed in our *installation documentation*.

Should I use the stable version or development version?

We recommend that people use the latest "stable" (or "release") version of MPF unless you need specific features from the "dev" (next) version.

The current "stable" version of MPF is listed on the top of the MPF Users home page on Google Groups.

Where do I find information on older versions of MPF?

If you want information about an older version (0.30 and newer), click the "Read the Docs" link in the lower-left corner of any page on docs.missionpinball.org and select the version you want to read about.

You can install older versions of MPF with pip, like this:

```
pip install mpf-mc==0.31
```

Documentation for versions of MPF prior to 0.30 is available in this post

FAQ: Building your game

Where do I get help building my machine?

If you're looking for information about physically building your machine, check out the PinballMakers.com website.

I want to do something that's not in MPF. Now what?

Awesome!

First, you can check out the list of new features that we're tracking.

- MPF New Features
- MPF-MC New Features
- MPF Monitor New Features

If you see your feature there, you can click on it and then click the "Subscribe" button to receive email notifications of progress or when it's been added.

You can also read our *MPF Road Map, Vision & Future* for an idea of our longer-term plans for MPF.

If you still don't see your idea, of you'd like to talk about it or ask questions, feel free to post a message to the *MPF Users Google Group <https://groups.google.com/forum/#!forum/mpf-users>*.

FAQ: Getting help

Where can I go for help?

If you're stuck with something, feel free to post a message to the MPF Users Google Group.

I think I've found a bug. Now what?

Again, post it to the MPF Users Google Group.

I want MPF to work with a new piece of hardware

Awesome! We've designed MPF to be platform-independent, meaning that the core MPF software doesn't talk to hardware directly. Instead we have "platform interfaces" for different types of hardware.

The easiest way to understand how these work is to look through the code for the existing platform interfaces. This code is in the platforms folder in MPF.

As always, if you have questions, please post them to the MPF Users Google Group and we'll go from there!

CHAPTER 34

Glossary of MPF terms

Here's a list of terms you might come across in MPF. Note that this is not an exhaustive list of everything, rather, these are terms we use in MPF that might not be obvious.

display A logical target which holds slides. Displays are abstract–purely logical. You use the machine config to map logical displays to the physical on-screen window or a DMD.

machine folder The folder which holds your machine config files.

player variable A named value that is stored on a per-player basis, such as the current ball number or score.

watch dog A feature of a hardware control system that ensures you don't blow anything up if MPF crashes. Essentially it's a timer which runs on the hardware (typically set to a short amount of time, like 1 second) that has to be "pinged" by MPF constantly to reset the timer. If the timer runs out before its pinged, then the hardware system will shut off all power to its devices. In normal operation, MPF pings the watchdog constantly, but if MPF crashes or shuts down ungracefully, then the watchdog pings stop, the hardware timer expires, and the hardware controller shuts off all the power to the connected devices.

widget A thing that is put on a display. There are different types of widgets, such as text, images, videos, shapes, etc.

CHAPTER 35

Contributing to MPF

Want to add a feature? A missing event somewhere? Wrote a new device which might be useful for other users? Fixed a bug? Added some small missing piece?

We'd love to take your contribution upstream!

Found a bug which you can reproduce? Fill an issue:

- MPF Issues on github. Use this for game and platform related bugs
- MPF-MC Issues on github. Use this for media controller bugs such as problems with slides, widgets or audio.

If you want to discuss a feature or bug (or if you are unsure). Visit our forum: https://github.com/orgs/missionpinball/discussions

Install MPF in development mode

To make changes to MPF you need to install it in developer/editable mode:

1. Fork the mpf repo on GitHub. Do this by clicking on the Fork button in the top right corner.
2. Clone your fork of the mpf repo to your local machine. Determine the folder where you want this to reside. Consider using a different folder than where your personal MPF code resides. Then run the following command: (`git clone https://github.com/YOUR_GITHUB_HANDLE/mpf/`)
3. Install the MPF dependencies if you haven't installed mpf before. On linux you can run our mpf dependency installer. On other platforms check the *installation instructions* instructions.
4. Navigate to your cloned repository and run: `pip3 install mpf-mc --pre` to install MPF in editable mode.

Install MPF-MC in development mode

If you want to make changes to MPF-MC (Media Controller) you will need to install it in developer/editable mode:

1. Fork the mpf-mc repo on GitHub. Do this by clicking on the Fork button in the top right corner.
2. Clone your fork of the mpf-mc repo to your local machine (`git clone https://github.com/YOUR_GITHUB_HANDLE/mpf-mc/`).
3. Install the MPF-MC dependencies if you haven't installed mpf-mc before. On linux you can run our mpf mc dependency installer. On other platforms check the *installation instructions* instructions.
4. Navigate to your cloned repository and run:`pip3 install -e .` to install MPF-MC in editable mode

To work on both the MPF and the MPF-MC complete both sets of instructions. Make sure you don't install the

After cloning and installing the dependencies for the desired project follow these steps.

1. Switch your repository to the branch corresponding to the version you want to work with. This should be `dev` in most cases or the current release for smaller bug fixed. Do what works best for you. We can help to forward or backport your changes.
2. From your MPF folder that is connected with git, create a local branch to work on (`git checkout -b your_feature_name`).
3. Make your changes.
4. Add your name to the `AUTHORS` file.
5. If possible add an unit test. We can help with that and a first Pull Request without a test is definitely fine.
6. Run `python3 -m unittest discover -s mpf.tests` and check that all tests still pass.

   To check that all tests will still pass for mpf-mc run `python3 -m unittest discover -s mpfmc.tests`.

   If you get an error message that Python was not found, try running the following command: `python -m unittest discover -s mpfmc.tests`. This is the same basic command, but runs on python instead of python3.
7. Commit your changes (`git commit -a`)
8. In the git commit screen type your title on line 1. Leave a blank line, and then type out a description of what is included in this commit. Once you are done typing your commit notes, press escape. This will bring your cursor to the bottom of this panel. From there type (‘’:wq”) and press Enter. This will complete your commit notes.
9. Push your changes to your github (`git push origin your_feature_name`).
10. Open up your Fork on github and create and submit your pull request to merge from your local back to MPF.

We recommend you to use a decent IDE because it makes life easier. Most of the MPF developers use PyCharm but other IDEs will work as well.

Getting started with an open issue

We maintain a list of issues which are self-contained and good to solve on their own without too much interaction with core code. We label those as help wanted (although they do not have to be easy, just self-containted). If you want to work on one of them (or any other issue) comment on the issue or write in the forum and we will assist you along the way.

CHAPTER 36

Contributing to MPF's Documentation

Want to help make these docs better! Great! We'd love any help, whether it's as small as correcting a typo, adding to a section that isn't clear, adding your own How To guide, or whatever else you want to change.

Video about contributing to the documentation:

https://youtu.be/jj8uM25B_yY

If you got any questions please ask in the MPF Users Google Group. We are happy to help you with any contribution.

To make a quick change to an existing page

Quick changes to existing pages can be done right on the web!

To do that:

1. Browse to the page you want to update, and click the "Edit on GitHub" link in the upper right corner of the page.
2. Click the pencil icon in the upper-right corner of the page's text. (If this is grayed out, that means you need to create a GitHub account and/or login.) This will create a fork of mpf-docs in your GitHub account.
3. Make your change, and click the "Propose file change". This will create a pull request. Type a name describing your change, and click "Create pull request".
4. Details and screen shots of this entire process are here.

To make a suggestion for a new doc (or to point out an error)

Even if you don't feel comfortable actually changing or editing docs, you can still tell us about an error in the documentation or suggest new documentation that we should add. To do this:

1. Go to the "Issues" page of the mpf-docs repository on GitHub.
2. Create a GitHub account if you don't have one, and/or login.
3. Click the "New Issue" button and describe what you'd like us to fix or add!

How does the layout work?

The documentation uses reStructuredText (rst). You can read about possible elements in the rst documentation.

Some excerpts from the documentation above:

A bulleted list of items:

```
* element 1
* element 2
```

Looks like:

- element 1
- element 2

Highlighted yaml:

```
.. code-block:: yaml

  element:
    subelement: value
```

Looks like:

```
element:
  subelement: value
```

To clone the mpf-docs repo locally to make bigger changes

If you want to make bigger changes to the docs, or if you want to download the mpf-docs repo so you can work on it offline, do the following:

1. Clone the mpf-docs repo from GitHub.
2. Switch to the branch corresponding to the version of the docs you want to work with (usually *dev* or *latest*).
3. Makes your changes.
4. Add your name to the `/about/authors.rst` doc.

5. To test the docs locally, you'll need *sphinx* and *sphinx_bootstrap_theme*, both of which you can install via *pip*.
6. Run `make html` to ensure everything builds properly without any additional warnings from whatever docs you added or changed. (The built docs will be in the `_build/html` folder. You can open *index.html* in your local browser to preview your changes.)
7. Submit your pull request

CHAPTER 37

MPF Versions

MPF is a work-in-progress. New versions are released every so often. You can find your MPF version by running *mpf –version* from the command line.

If you're looking for a specific version of the documentation to match the older version of MPF you're using, look in the "Assets" link at the bottom of the release notes for any version in the MPF releases page on GitHub.

User Documentation for Older MPF Versions

If you're looking for the documentation for an older version of MPF, you can find it here. Each link below contains the docs in several formats, including PDF, text, and HTML zip bundles of the documentation for that version.

- MPF v0.33 Docs
- MPF v0.50 Docs
- MPF v0.51 Docs
- MPF v0.52 Docs
- MPF v0.53 Docs
- MPF v0.54 Docs
- MPF v0.55 Docs
- MPF v0.56 Docs

Understanding MPF version numbering

This page explains:

- How version numbering works in MPF, and
- How the MPF documentation versions map to the MPF versions.

MPF is under constant development. The core developers typically spend a combined 40 hours a week working on MPF with multiple fixes and enhancements made every day. You can see the stream of code "commits" on GitHub, here for MPF and here for MPF-MC. (Actually we work on the docs a lot too, check out the latest updates here.)

Anyway, we release a new version of MPF about every 6 months. (See the full release history *here*).

MPF version numbering follows a standard called semantic versioning which uses a "MAJOR.MINOR.PATCH" version number format. For example, the version number `0.31.8` is major version 0, minor version 31, and patch number 8.

Note: Version numbers in MPF are numbers separated by dots which are *not* mathematical decimals. In other words, MPF 0.30 is "zero point thirty", which is not the same as "0.3" which is "zero point three". Also, 0.30 is 27 versions newer than 0.3.

All the MAJOR versions of MPF start with "0" because we have not yet released a 1.0 version yet.

MPF features and configuration files can change between MINOR versions. For example, there were significant changes between versions 0.21 and 0.30.

The PATCH versions are bug fixes only which do not have functional or config file changes. So 0.30.0, 0.30.1, and 0.30.11 are all the same in terms of documentation and features. (Also 0.30.11 is ten patches newer than 0.30.1.)

You can see which version of MPF you have by adding a `--version` option to whatever command you use to launch MPF. For example:

```
mpf --version
```

Since MPF is actually two projects (MPF and MPF-MC), all of this version stuff applies to both of them. (Typically you'll use the same MAJOR.MINOR versions of both, but the PATCH number might be different. For example, the latest MPF version might be 0.31.11 while the latest MPF-MC version could be 0.31.8. That's fine.)

You can see which versions are the latest released versions at any time by visiting the MPF Users Google Group where we list the latest versions in the header of the page.

Documentation Versions

Since MPF versions are constantly changing, we're also constantly adding and improving the documentation.

Generally speaking, each documentation set covers multiple MPF versions. You can see the current version(s) of MPF the documentation you're reading is for by looking for the version numbers in the blue box in the upper left corner under the Mission Pinball logo of any page on the documentation site.

If you'd like to access documentation for an older version of MPF, you can click the "Read the Docs" link in the lower left corner of any page.

If you look in the URL for a page, you'll see the version(s) of MPF that page is for. Note that there's a special version of the docs called "latest" which always points to the latest version of the docs. (That way you can safely link to a page and know that in the future it will always be the most recent version.)

MPF Release Notes

Here's the history of the various release versions and changes of the Mission Pinball Framework. (Patch releases and bug fixes are not included in this list.)

0.56 (and newer)

Release notes have moved to the MPF releases page on GitHub.

Note you can also click the "Assets" section at the end of the notes for each release to download the PDF or HTML versions of the documentation for that specific release.

0.55

Released: June 25, 2021

Video about some of the changes in 0.55:

https://youtu.be/2rYxmdr3ZMU

Video about release creation:

https://youtu.be/EilOwZV94_8

Breaking changes in config

- Removed Python 3.5 support
- Added Python 3.8 and 3.9 support (default in Ubuntu 20.04)

New Features

- Flashing Segment Displays in P-Roc
- Segment Display Match Flashing
- Visual Pinball Engine (VPE) Support
- New argument "remaining" in counts
- Initial support for auto-generating wire harnesses
- Tilt improvements
- New hardware: Initial PKONE support
- Improved config validation
- More Service Mode Features
- Open Pinball Project 2.1 Firwmare (for Cobrapin)
- State Machines in non-game modes
- EOS repulse in software
- Better EOS support in FAST and P/P3-Roc

- Ball search only starts at boot when there is at least one ball
- Allow updating speed and manual_advance of shows
- Power management for enable on coils
- Production bundles for config in production machines
- RGB segment displays
- New hardware: FAST segment displays
- Segment displays emulator
- Animations for segment displays
- New command: "mpf hardware benchmark"
- Improved servo support
- Support switches in Pololu Tic
- Add more subscriptions and placeholders
- New spinner device
- New crash reporter
- More and better segment mappings
- Better drop target event behaviour

New Config Options

- New delay setting for all config players to delay execution
- New option enabled for displays
- New option max_hold_duration for coils to prevent burning your coils by accident
- Persist_frame on images
- logic_block_timeout for all logic blocks (counters, accruals and sequences)
- Added block in sound_player
- New option stop_timeout_after_last_move in servos

Bug Fixes

- Fixed color bugs during fading
- Fix P-Roc driver_pulsed_patter
- Fix bug where initial count of playfield has been wrong
- Ball lock fixes when physical lock has been full
- Highscore mode fix
- Fixed bug on ball tracking during eject with plunger
- Fixed crash on multiple returning balls in the trough
- Fixed crash in bonus

- Fixed crashes in service mode
- Fix timer on step_back and advance in shows
- Fix ball search behavior for diverters
- Fixed bitmap font bugs

0.54

Released: November 7, 2020

This release contains incremental improvements and a lot of bugfixes. We identified a few potential upgrade issues:

- Deprecated `ball_locks` device has been removed. Use `multiball_locks` or `ball_holds` instead.
- Space-separated lists have been removed. Use comma-separated lists or yaml lists instead (with or without spaces). MPF sticks to YAML conventions here and allows all kinds of legal YAML lists (which does not include space-separated lists).
- Deprecate `playfield_active` tags on shots. Those tags are only required for switches which are not part of shots or devices (so almost none). MPF will complain and you might have to remove the tag in that case.
- MPF will complain on event handlers with the same name as a switch. This should not happen in practice and has been done to catch typical user error (i.e. using the event `s_my_switch` instead of `s_my_switch_active`).
- Diagnostics menu (switch, coil, light) is now a sub-menu in service mode.

MPF, MPF-MC, MPF-LS and MPF-Monitor

New Features

- Deduplicate asyncio code - jab
- Support more Pin2DMD hardware options - jab
- Do not flush in pypinproc - jab
- Do not call flush on write_data in pypinproc to speed up LEDs on PD-LED - jab
- Better default logging for ball devices - jab
- Support event args in show_tokens - jab
- Log virtual time in unit tests - jab
- New “mpf format” command to format configs - jab
- Refactor hardware fades for performance - jab
- Driverboards per platform to support FAST and P-Roc in parallel in one machine - jab
- Crash asset loader thread on exception - jab
- Validate widgets and targets in slide_player - jab
- Validate slides in widget_player - jab
- Refactor pypinproc to use PRWriteDataUnbuffered - jab

- Refactor libpinproc to use PRWriteDataUnbuffered - jab
- Util cleanup - jab
- Turn off incands at start for OPP - jab
- Remove space separated lists - jab
- Support delayed pulses in autofires and kickbacks and implement it for OPP - jab
- Refactor config loading - jab
- Support serial LEDs in OPP on new boards - jab
- Enable dot priority syntax everywhere - jab
- Remove dash syntax for control events - jab
- Unity config spec loading for mpf and mc - jab
- Remove ball locks as they have been replaced by multiball_locks and ball_holds - jab
- Dynamic value for keep_multiplier in bonus mode - seanirby
- Batch commands for PD-LED - jab
- Inputs on Neopixel wings in OPP - jab
- Add mpf build production_bundle - jab
- Log config load times - jab
- Interface for binary data storage (instead of yaml) for high scores and audits - jab
- Test software update in service mode - jab
- Fix asset loading in overloaded modes - jab
- Remove space separated lists in MC - jab
- Refactor Config Loading in MC - jab
- Build MC on Python 3.5 to 3.7 - jab
- Support Production Config Bundles in MC - jab
- Better error messages for incorrectly formatted shows - jab
- Retry connect to LISY/APC serial - jab
- Validate shows in achievements - jab
- Improve smart_virtual errors - jab
- Improve error when a required setting is missing - jab
- Improve generic validator errors - jab
- Support switches in OSC platform - jab
- Implement events in OSC platform - jab
- Support BCD, 14-segment and 16-segment displays as segment_display - jab
- Improve empty device collection error - jab
- Validate playfield_active tags on shot switches - jab (breaking change - you have to remove those tags)

- Point users to our fork of apigpio (called apigpio-mpf) - jab
- Validate platforms and prevent configuring features which do not exist on platform - jab
- Runtime errors with documentation links - jab
- Add glow effect and 2 - seanirby (see blog post about glow effect)
- Add font for 14-segment displays similar to Williams System 11 displays - seanirby
- Pin all dependencies - jab
- Commandline config generator - F4b1-
- Add end_ball and end_game events to game - jab
- Prevent true and false in placeholder (use True and False) - jab
- Expose more P/P3-Roc errors - jab
- mpf hardware scan for LISY - jab
- Refactor driver lights to properly encapsulate internals - jab
- Parallel device initialisation - jab
- Implement chained lights - jab (see separate blog post)
- Add spread spectrum modulation (SSM) PWM for fast coil for low-noise hold - jab
- Improve error message on failed template evaluation - jab
- Add debug output to state_machines - jab
- Better config validator error paths - jab
- Support new templates syntax for all template_str - jab
- Add subscriptions in variable_player - jab
- Pass timestamps from platform for switch changes - jab
- Refactor hot switch path for performance, 2, 3, 4 - jab
- Add sound_loop_start_at/end_at and implement them in MC - qcapen
- Allow multiple entrance_switches - jab
- Prevent event handler with the same name as switches (to catch common beginner mistakes) - jab (breaking change in theory but unlikely for real machines)
- Performance improvements - jab
- Add show_queues to serialize shows - jab
- Support pinproc in Python 3.7 and 3.8 on Windows - qcapen
- Recompiled pinproc for Python 3.5 and 3.6 on Windows to include recent improvements - qcapen
- Improve memory leak finder - jab
- Add debug button in iMC - jab
- Load named_colors in mc and test them - jab
- Require ffpyplayer for all platforms as it seems to solve video issues - jab
- Better type hints in mpf-ls - jab

- Autocomplete events and go to definition for events - jab
- Support more events in mpf-ls - jab
- Install latest kivy in debian installer - jab
- Better error handling in debian installer - jab
- Add source_devices to multiball_locks - jab
- Select pulse_ms based on ball count during eject - jab
- Add start_running option to shows - avanwinkle
- Support pulse_power in P/P3-Roc where possible - jab
- Better log output for P/P3-Roc - jab
- Always log OPP chain serial - jab
- Support GPIO inputs on P3-Roc - jab
- Faster and better light batching - jab
- Support Neopixel Wings on OPP - jab
- Prevent fades to the previous color - jab
- Deterministic fades - jab
- Allow platforms to set batch granularity for fades - jab
- Improve ball counters - jab
- Python 3.8 compatibility (only MPF not MC because of kivy) - jab
- Support Repulse on EOS in MPF (only supported in Spike so far) - jab
- Event to reset high scores - jab
- Event to reset audits - jab
- Event to reset earnings records - jab
- Event to reset credits - jab
- More modern service mode - jab
- Add twitch bot support - Mark Seiden
- Improve twitch bot - Mark Seiden
- Add advance_random_events to accruals - jab
- Show a nice error when communication with P/P3-Roc breaks down - jab
- Support more than 256 lights in LISY API > 10 - jab
- Extend motor device - jab
- Add shop jump - avanwinkle
- Add settle_time_ms to entrance switch counter to prevent ejecting thin air - jab
- First version of VPE platform (not finished yet) - jab
- Test and build on Ubuntu 20.04 - jab
- Support conditional events and fallback for random_event_player - avanwinkle

- Python 3.8 support in MPF-MC (except kivy) - qcapen
- Faster image loading in sequences - jab
- Add block events to text_input and use them in carousel - avanwinkle
- Nicer errors in MC - avanwinkle
- Expose switch config in pypinproc - jab
- Support loading light shapes from MPF Monitor in showcreator - markinc
- Add Mac build for showcreator - markinc
- Improve logging in MPF Spike Bridge - jab
- Extend MPF Monitor with a lot of new features - kylenahas
- Monitor performance improvements - kylenahas
- More monitor perf improvements - jab
- Add config arg to MPF Monitor - avanwinkle

Bug fixes & code improvements

- Fix fast shutdown bug when an error occured - jab
- Prevent crashes from empty platform configs - jab
- Fix crash in some MC players - jab
- Fix multiple subscriptions in show_player - jab
- Fix new fades in VPX - Wolfmarsh
- Add test for VPX platform - jab
- Fix multiple subscriptions in light_player - jab
- Fix gamma test slide - jherrm
- Add test for gamma_test_slide - jab
- Do not crash test when sound system is not loaded - jab
- Test and fix end_bonus_event - jab
- Only validate widgets when using the add action - jab
- Fix master volume bug - qcapen
- Fix asset loading when overloading a mode fixes bug 1366 - jab
- Detect missing curly backets in conditional events fix bug 1497 - jab
- Prevent adding player during high score of a one ball game - seanirby
- Fix config spec for hardware section - jab
- Fix servos on PD-LED with new libpinproc and add a test - jab
- Fix subscriptions in logic blocks - jab
- Fix broken subscriptions during player change - jab
- Disable Mac Wheels as they caused install issues - jab

- Fix crash in smart_virtual with entrance_switches - jab
- Fix achievement_group auto_select with allow_selection_change_while_disabled - jab
- Fix BCP encoding crash - seanirby
- Remove lower-casing of colors because it breaks placeholders - jab
- Fix crash in variable_player - seanirby
- Fix non-connected switches for P3-Roc - seanirby
- Fix initial switch state for RPi platform - jab
- Fix OSC crashes with complex event parameters - jab
- Fix ball count in multiball_lock full event with physical_only strategy
- Do not poll OPP boards without switches - jab
- Fix input mask for OPP Neopixel wings - jab
- Allow duration for wipe transition - jab
- Fix crash when not specifying keep_multiplier in bonus entry - jab
- Fix random argument order in OSC events - jab
- Fix crash in drop_target - jab
- Respect switch and coils defaults for autofire rules - jab
- Fix init race in steppers - jab
- Fix number crash in FAST - jab
- Fix late crash during shutdown - jab
- Fix crash in digital_outputs with FAST platform settings - jab
- Consistent fade_out for display_light_player - jab
- Fix bash export in installer - jab
- Fix crash when a ball is lost (because of the next bug) - jab
- Prevent ball skipping when target is not a ball device - jab
- Consistent jam switch handling in ball counter - jab
- Prevent incorrect playfield activation by drop_target_bank resets - jab
- Fix light ordering for fades - jab
- Fix config parsing for developers.missionpinball.org - jab
- Use the correct commands for the correct Spike Firmare (Spike System 1 vs System 2) - jab
- Correct Active Mode Updates to MPF Monitor - jab
- Fix config validation issues with System 11 - jab
- Fix potential crash - jab
- Always configure both banks of all PD-16s on P/P3-Roc to prevent polarity issues and stuck on coil on the hardware - jab
- Fix sound loop bug - qcapen

- Fix loop bug when stealing/replacing a playing sound with a higher priority sound - qcapen
- Fix animations when two slides animate the same image - jab
- Do not crash on empty config collections - jab
- Fix animations in slides in shows - jab
- Prevent crash in sound_player with placeholders - jab
- Expose video control events to MPF - jab
- Fix crashes in image pool and regression test them - jab
- Fix Spike 2 Init Sequence - jab
- Fix incorrect active modes in MPF Monitor - jab
- Prevent crash in Monitor - jab

MPF Documentation

- Release notes to 0.53 - jab
- Extend fadecandy documentation - jab
- Document Pin2DMD - jab
- Faster docs generation - jab
- Remove stuff from roadmap which has been implemented - jab
- Link to our libpinproc fork - jab
- Add link to VS Redistributables for pypinproc on Windows - jab
- Fix DMD font style names - kevwilde
- Support assets in doc tests - jab
- Support virtual platform in doc test cases - jab
- Document common problems with Numlock when using keyboard in MPF - jab
- Example for multiball without physical lock - jab
- Reformat all examples for good copy and paste experience - jab
- Extend PD-LED FET documentation and drawing - colemanomartin
- Test and fix mc examples, more and more - jab
- Test all slides in the tutorial - jab
- Improve PD-LED documentation - seanirby
- Fix typo - driskel
- Fix settings name - enteryourinitials
- Update docs for driverboards per platform - jab
- Test and fix DMD style names in examples - jab
- Test and fix all kinds of slightly broken examples - jab
- Test and fix animation examples - jab

- Test and fix widget examples, more and more - jab
- Test and fix slide examples and more - jab
- Test and fix display examples - jab
- Test remaining mc examples - jab
- Add dual_wound_coil example for diverters - SwizzleFish
- Document solution for common Windows install problem - AdrianD72
- Add mystery award example - aaronmatthies
- Fix broken links and references to ball_locks - aaronmatthies
- Link to APC video - jab
- Remove old-syntax list examples from docs - jab
- Use commas to separate lists - jab
- Dual-coil diverters - jab
- Add generic part numbers - jab
- Document Motors - jab
- Document Shakers - jab
- Add Pop Bumper Images - aaronmatthies
- Add example how to end a game by long-pressing start - jab
- Describe PSU magic - jab
- How to fix drop target reset issues - jab
- Document Pololu Tic - jab
- Reference placeholders in bonus mode - seanirby
- Keyboard tutorial - jab
- Integrating Logic Blocks and Lights - jab
- Tutorial on Counter and Slide integration - jab
- Update all config references: OPP, Pin2DMD and P-Roc and many more - jab
- How to drain all balls and keep the ball live, 2 - mwiz
- Improve achievments documentation - atummons
- Fix event annotations - jab
- Remove old section about shot reuse - seanirby
- Update config references for all kinds of devices, 5, 6, 7 - jab
- Document color_correction_profiles - jab
- Notes about style for text sizes - jab
- Update tutorial - jab
- Update motors - jab
- Render nice 404 with helpful links jab

- Links to list of documented error messages - jab
- Document show format errors - jab
- More errors and document MPF language server - jab
- Update BCP reference - jab
- Update multiball_locks reference - jab
- Update steprocker reference - jab
- Update achievements reference - jab
- Update widget_style reference - jab
- Improve state_machine - atummons
- Document common errors - jab
- Update videos reference - jab
- Add VPX to tutorial - jab
- Document OSC platform - jab
- Update variable_player reference - jab
- Update snux reference - jab
- Update player_vars and shot_groups reference - jab
- Document light_segment_display - jab
- Document WS2812 specifics and similar chips - jab
- Document CFE-ConfigValidator-4 - jab
- Document CFE-ConfigValidator-2 - jab
- Document CFE-ConfigValidator-1 - jab
- Update logic_blocks reference - jab
- Document CFE-ConfigValidator-12 - jab
- Document CFE-ConfigValidator-13 - jab
- Document CFE-DeviceManager-3 - jab
- Document mpf build production_bundle - jab
- Update track_player reference - jab
- Update sounds reference - jab
- Improve ball_device reference - chris20-20
- Improve switches reference and more - chris20-20
- Fix typo and more typos - chris20-20
- Update sound_system reference - jab
- Update sound_player reference - jab
- Document defaults in references - jab
- Add links to tutorial and more links - chris20-20

- Improve tutorial - chris20-20
- Improve coil_player documentation - chris20-20
- Fix LCD width and height - chris20-20
- Document MC errors - jab
- Fix link in docs - F4b1-
- Document glow effect - seanirby
- Improve event reference - jab
- Add physical building section - jab
- Improve common ground warning - jab
- Add common issues section for Multimorphic - jab
- Playfield layout considerations from Jimmy - jab (content from Compy)
- More on common ground from Gerry Stellenberg - jab (content from Gerry)
- Update instructions to build docs locally - seanirby
- More playfield layout and images - Compy
- Example on how to end a game properly using events - jab
- More details and considerations on coils - jab
- Properly document MPF language server - jab
- Clarify that a RPi is not a pinball controller without further hardware - jab
- Related links for all driver howtos - jab
- Bring back Indy Lane tutorial from old website - jab (based on content from Brian)
- Warn about current Python 3.8 issues - BENETNATH
- Fix typo in udevadm command - BENETNATH
- General hardware troubleshooting guide - jab
- mpf hardware scan example for the P-Roc - jab
- Document common P/P3-Roc issues - jab
- Link troubleshooting section from more places - jab
- Troubleshooting guide for FAST hardware - jab
- Correct addressing section for P3-Roc - Coleman
- More hardware troubleshooting for P3-Roc boards and cables - Coleman
- Document new game events - jab
- Document -t command line option - jab
- Troubleshooting guide for OPP hardware - jab
- Troubleshooting guide for LISY/APC - jab
- How to ask questions in the forum for hardware issues - jab
- Example for transition_out - jab

- Better widget examples - public-profile
- CSSC instructions on Linux - jab (content from Scott Danesi)
- More OPP troubleshooting - jab
- Document default_pulse_power/default_hold_power limitations in P3-Roc - seanirby
- Troubleshooting for Fadecandy - jab
- Pin2DMD troubleshooting - jab
- Suggest firmware updates for P/P3-Roc and FAST - jab
- Extend high voltage warning - jab
- Document default recycle times in P/P3-Roc - jab (content from Gerry)
- Document debounce and recycle behaviour of autofire_coils - jab
- Document chained lights and numbers vs channels for all platforms - jab (see separate blog post)
- Coil troubleshooting - jab
- FAST on Linux troubleshooting - jab
- Document debounce and recycle behaviour of flippers - jab
- Notes on RGB and colored inserts - jab
- How to install Debian with MPF in VirtualBox - kylenahas
- Example for state_machines with placeholders - jab
- Document start_loop_at/end_loop_at on sounds - qcapen
- Document rotation animations - Coleman
- Readd tutorial to mpf-examples and test it - jab
- Fix sound references in demo_man - kylenahas
- Add monitor image and config to demo_man - kylenahas
- How to wire coils and scoops - jab
- Magnet example - jab
- How to debug MPF Spike Bridge - jab
- Add Physical Building Section - Nate
- Add Stern Magnet Board - jab
- Document start_running in shows (with examples) - avanwinkle
- How to capture spike net bus - jab
- How to replace FETs on FAST hardware - jab
- Dedicated Magnet Driver boards - jab
- Fix typos - bghill
- Update Windows Install Instruction for Multimorphic - qcapen
- Add part numbers - bghill
- Fix snux docs and more - jab

- Remarks on referencing slides in a show from outside - jab
- Document twitch bot - Mark Seiden
- Add details about keys and widgets - atummons
- Enhance twitch docs - Mark Seiden
- Document known P/P3-Roc errors - jab
- Link correct demo man from docs - jab
- Document common demo man issues - jab
- Document advance_random_events - jab
- Document reset_audit_events - jab
- Document repulse on EOS for flippers - jab
- Document reset_high_score_events - jab
- Document light chaining with previous and start_channel - jab
- Document source_device in multiball_locks - jab
- Update Motor documentation - Lance-o-nator
- Improve tutorial - flamtime
- Add driver troubleshooting - jab
- Document P/P3-Roc runtime errors - jab
- P/P3-Roc Firmware Upgrade section - jab
- Document CobraPin platform - cobra18t
- Fix reset_when_complete in docs - avanwinkle
- Document carousel block_events - avanwinkle
- Document more common errors - jab
- More breakout boards - jab
- Ubuntu 20.04 install instructions - jab
- Add missing config references for release - avanwinkle
- Renamed end_loop_at and start_loop_at to loop_end_at and loop_start_at - qcapen

0.53

Released: January 11, 2020

This is a 0.52 maintenance release with cleanups and some refactorings. We identified a few potential upgrade issues:

- We fixed validation of animations. You might get a validation error with *repeat: -1*. Change it to *repeat: false*. See the change in the docs.
- We changed *active_time* of ball_save from ms to secs. In case you did not use a unit here this might change the time. Details.
- Machine variables changed if you accessed them from code (but not via config).

- Achievement state changed if you accessed it from code (but not via config or placeholders).

MPF and MPF-MC

New Features

- Support segment displays connected to normal light of a platform - jab
- Batch LED updates for PD-LED and P/P3-Roc to prevent bus overflows - jab
- Make separate thread configurable in P/P3-Roc and reduce IPC overhead - jab
- Highlight settings in service mode - avanwinkle
- Spike-MPF bridge in Rust - jab
- Use new Spike-MPF bridge in MPF - jab
- Use a better default for max_servo_value on PD-LEDs - jab
- Allow reverse sorted highscore categories - yensho
- Light batching in Spike for better light sync - jab based on request by Dave
- Read ticks_per_second per node for Spike - jab
- Reliable speed/flow control in Spike - jab
- Initial Spike 2 support for the mpf-spike bridge - jab
- Limit light batch size in Spike to prevent bus desync - jab
- Ignore duplicate handler warnings during init - avanwinkle
- Add support for steppers in Spike - jab
- Support Spike 2 backlight - jab
- Support Spike 1 and Spike 2 backlight in bridge - jab
- Servo and Steppers as Diverters - jab
- Separate event handlers and code to catch incorrect arguments in custom code - jab
- Auto launch when machine is tilted - jab based on question from Philip D
- Show player and machine variables in the Text UI - woosle1234
- Allow dynamic values in timer control events - avanwinkle based on report by wilder
- Reduce default batch size for Spike LEDs - jab based on tests by Dave
- Custom events_when_added and events_when_removed for widgets [2] - qcapen based on feature request by cfbenn
- Better cache invalidation of config_spec cache - jab
- Refactor Text UI to prevent text clutter - jab
- Allow user to disable ball search in a ball device - dziedada
- Better signal handlers and shutdown logging during crashes - jab to fix some exit issues
- Improve show and lights performance - jab
- Refactor DelayManager - jab

- Exit MPF when the FAST Nano reboots/crashes during a game - jab
- Add a setting for free play to service mode when credits mode is loaded - jab based on request by Greg
- Allow newer FAST firmware versions - jab based on problems with Firmware 1.05 by Brian Cox
- Support inverted switches and non-numeric drivers in Virtual Pinball - mfuegemann
- Extend README and add hardware rules to VPX Bridge and Test- mfuegemann
- Placeholders in credits mode - jab
- Placeholders in tilt mode - jab
- RGB LEDs and flashers in Virtal Pinball - mfuegemann
- Update asciimatics - jab
- Add –vpx commandline option to mpf and mc- jab
- Add VPX demo table with MPF config - mfuegemann
- Placeholders for StateMachine devices - jab
- Initial support for the Arduino Pinball Platform - jab, bontango and blackknight
- More debug in FAST platform and longer wait times - jab to support more FAST firmwares
- Generic System 11 A/C Relay handling (for APC and Snux) - jab
- Improve duplicate event handler message - jab as it caused confusion for Sepp
- Better error message when number is empty - jab based on report by Sepp
- Placeholders in show_tokens in show_player - jab to allow dynamic values in all widgets
- More useful and accurate validation errors in dicts - jab
- Add links to the docs to warnings and errors - jab
- Improve fake game in tests to handle multiball drains - jab
- Remove Windows Python 3.4 build of MPF-MC - qcapen
- Improve sound_loop_player design - qcapen
- Python 3.7 support for Windows in MPF-MC - qcapen
- Add placeholder conditions for items in carousel mode - avanwinkle
- Add control events to counters - dziedada
- Support for the APC platform - jab, bontango and blackknight
- Validate switch numbers in LISY/APC - jab
- Set DTS to low on connect for APC and clear serial after reset - jab
- Modern lights for LISY/APC - jab
- Refactor sound loop - qcapen
- Allow tokens for widgets in shows - jab based on request from Sean-Paul
- Don’t activate diverter if activate_event present - GabeKnuth
- Add enabled and rotation_enabled to placeholders for shots/shot_groups - jab based on request from Mike

- Throws Error when attempting to define more than one default display - GranolaDaniel
- Update unity-bcp-server to latest version - qcapen
- Segment display support for APC - jab
- Add token to slide_player to pass variables and MC - jab based on request in the forum by Greg
- Increased light update throughput - jab
- Add express syntax for sound_player - jab
- Refactor machine variables - pmansukhani
- Tune shows and events - jab
- Setup improvements and wheels for OSX - qcapen
- Nicer errors on syntax errors in conditions - jab
- Improve debug log of early messages in OPP - jab
- Option to send length bytes in LISY protocol - jab
- Better error message on invalid displays in LISY - jab
- Load modes from subfolders - pmansukhani
- Move code out of the hot path for light updates - jab
- Reserve all show_player options in show_tokens to prevent indent mistakes - jab based on bug report by Alex
- Improve linter and remove previously undetected unused imports - jab
- Better debug output for LISY platform - jab
- Fix segment display mapping for APC - jab
- Configuration setting for player_vars and machine_vars to show in text ui - avanwinkle
- Better command logging for the P/P3-Roc - jab
- Support daisy chaining in the Pololu Maestro - jab
- Expose P-Roc hardware version as machine variable - jab
- Placeholders for shoot_again in multiball - pmansukhani
- Support show_tokens with placeholders in shot_profiles - jab
- Regression Test for Diverters (for a bug which was fixed during refactoring) - jab
- Expose MPF and MC version in MPF-MC on connect - jab
- Support pulse power in P/P3-Roc - jab
- Add Scaffolding CLI to MPF - jab
- Optimized Service Mode for LCDs - jab
- Suggestions on config typos - jab
- Copy light positions in scaffolding CLI from monitor to MPF for display_light_player - jab
- Add start_enabled to achievements and refactor code - jab
- Add unselect_events to achievements and more cleanup - jab

- More achievement refactoring - jab
- Refactored test cases - jab
- Drop Python 3.4 support - jab
- Turn device collections into native dicts - jab
- Led_color default show now supports all default show_tokens - jab
- Log asset loading times for tuning - jab
- Show shot state in MPF-monitor - jab
- Validate transitions in state_machines - jab
- Improve config parsing/validation - jab
- Nicer errors and suggestions in shows - jab
- Improve install and dependency manangement for Max and Linux - jab
- Improve build and install on Windows - jab
- Lazy loading for zipped image sequences to speed up game startup - jab
- New experimental language server support for IDEs - jab
- Generic high score mode which works for DMD and LCD, 2 - jab
- Improve correctness, speed and error messages of config validation - jab
- Option to ignore checksum errors in Spike - jab
- Support new input command for Spike FW 0.49+ - jab
- Implement over current detection for Spike - jab
- Arbitrary start state for state_machines - avanwinkle
- Configurable debounce times and FW 0.49+ for Spike - jab
- Coil priorities in hw rules for Spike FW 0.49+ - densminger and jab
- Placeholders in ball save active_time - avanwinkle
- Autodetect FAST ports - avanwinkle
- Improve robustness of LISY protocol - jab
- Emacs instructions - seanirby
- Support goto definition and hover + mode support - jab
- Basic diagnostics - jab
- Improve placeholder performance by evaluating them only when needed - jab
- Update ruamel.yaml to improve the install experience on Windows - jab
- Benchmark and tune/cache placeholder parsing - jab
- Priorities in ball_holds and ball_locks - avanwinkle
- Batch light for PD-LED - jab
- Benchmark and tune event performance - jab
- Extend combo_switches to include the triggering switch in the event - avanwinkle

- Initial Pin2DMD support (not yet working) - jab
- Option to ignore FAST RGB CPU crashes - avanwinkle
- Tracing for libpinproc calls - jab
- Software update via Service mode - jab
- Add tests for accrual restarts - jab

Bug fixes & code improvements

- Fix some yaml parsing errors - jab
- Fix error with Python 3.7 - avanwinkle
- Fix driver stuck on in rules in P/P3-Roc - jab
- Do not crash in service cli when playing invalid shows - jab
- Fix crash in debug message for duplicate priorities - jab based on report from Dave
- Fix crash after config error - jab based on report by Wilder
- Properly use priority in widget_player when the slide is not active and becomes active later - avanwinkle
- Do not crash when failing to read stepper position in Spike - jab
- Allow carousel mode during attract - avanwinkle
- Do not start highscore mode without a game - jab based on report by wilder
- Properly save window positions in MPF Monitor - jab based on report by Greg
- Lock with physical_only strategy would never be full and count is off by one - jab based on report by Coleman
- Do not keep ball in outhole after tilt - jab based on report by Matt
- Fix crash in bonus mode with uvloop - jab based on report by Matt
- Prevent shutdown glitches in FAST - jab with the help of Dave
- Prevent crash during early errors in P-Roc - jab based on report by Coleman
- Preserve curly brakets in string_to_list - avanwinkle
- Fix bug preventing access to settings in custom code - avanwinkle
- Properly implement disable_random event in random_event_player - avanwinkle
- Fix enable attribute for placeholders in devices - avanwinkle
- Fix regression in multiball counting - avanwinkle
- Fix sound_loop_player bugs - qcapen
- Fix Mac build - qcapen
- Fix Kivy recursion erros in Kivy 1.11 - qcapen
- Fix events_when_xxx on sounds and 2 - qcapen and jab based on report by Greg
- Fix parsing regression in OPP with matrix input cards and more - jab

- Fix sound about to finish notification bug - qcapen
- Fixes for latest Spike Firmware and bridge - jab
- Always send a multiple of three LEDs to the Fadecandy to fix RGBW - jab based on bug report by Cadrion
- Fix polarity issue on P-Roc with WPC hardware - jab
- LISY command fixes in protocol v0.9 and 2 - jab
- Fix image unload crash in MC - avanwinkle
- Fix inverted condition on show player conditions - avanwinkle
- Prevent false positive duplicate numbers in virtual platform - jab
- Prevent crash in Text UI - jab
- Scaffolding from any path (just like other commands) - jab
- Set default enable/disable_event for magnets - jab
- Bring back state_names_to_not_rotate in shot_profiles - jab to fix bug reported by Greg
- Prevent false positive duplicate events handlers - jab based on report from Greg
- Fix crash in show player - jab
- Fix config validation - kevinleedrum
- Fix reenabling of achievement_groups - jab
- Improve error urls - jab
- Fix call to libpinproc for pulse_power - jab
- Do not crash on headless display_light_player - jab
- Fix setting number of LEDs per node in Spike FW 0.49+ - densminger and jab
- High score mode should run before match mode - jab
- Prevent crash in text ui on unknown switch event - jab
- Also advance score reels for non-active players - jab
- Consider OPP firmware version per chain instead of globally - jab
- Fix sequence_shots with a single switch and delay - jab
- Fix crash in score reels - jab
- Prevent crash in variable player when adding a variable for a non-exising player - jab
- Prevent duplicate BCP messages which could trigger duplicate sounds or widgets - jab

MPF Documentation

- Extend Multimorphic PowerEntry board documentation - colemanomartin
- Center Post Ball Save Example - mwiz
- Part numbers for trough opto boards - jab
- Image for Center Post - swizzlefish

- Improve game mode example - gregsealby
- Fix typos, fix2 - densminger
- Extend documentation for multiple screens - jab based on question by Haggis and solution by Snux
- Fix tutorial step 18 - jab based on question by Pablo
- Document new Spike bridge - jab
- Document steppers and add images - colemanomartin
- Image an image of a servo - colemanomartin
- Better stepper example code - colemanomartin
- Details about PD-LED servo fine tuning - colemanomartin
- Clarify monitorable servo properties - colemanomartin
- Document showcreator - jab
- Fix typo - cfbenn
- Docs for named_colors and example for dynamic widgets - avanwinkle based on request by Philip
- Better examples for sequence_shots - colemanomartin
- More text for the showcreator - jab
- Light_player examples - jab
- How to use shows in shows - jab
- Windows install error and fix - jab based on error from Jordan
- Document common logic block questions - jab based on question in forum from iizi
- Document servos and steppers as diverters - jab based on question in forum
- Document parameters of extra ball events - avanwinkle
- Document start_game_event and add_player_event - jab
- Add warnings about common ground to all coils - jab
- More tags vs tokens in shows - jab
- How to embed high score in attract mode - jab based on example by Greg
- How to display a timer on a slide - jab based on example from Coleman
- Common pitfall with accruals - colemanomartin
- Enable of StepStick needs to be low not high - colemanomartin
- Add Multimorphic part numbers for breakout boards and LEDs - jab
- Document breakout boards for switches - jab
- More homebrew part numbers - jab
- Thermal considerations about resistors on Optos - colemanomartin
- Document rotation on widgets - colemanomartin based on question in forum
- Update notes on rotation of widgets - colemanomartin

- Document custom widget events - qcapen
- How to configure tilt and change tilt slides - jab based on example/question in the forum
- Stern Spike Steppers - jab
- More examples for delaying game/ball ending - jab based on question by Coleman
- DIP 6 and Servos on the PD-LED - colemanomartin
- How to add a slam_tilt slide - jab based on suggestion in forum
- How to use sequence_shots in shot_groups - jab based on example by Greg
- Document shot_profiles - jab based on question by Jordy
- How to use virtual env on Mac with Kivy - driskel
- Improve dynamic values example - MarkInc666
- How to add credits settings to service mode - jab
- How to add tilt settings to service mode - jab
- Document placeholders for StateMachine devices - jab
- Document state machine configs - jab
- Add more config links and document timer transitions - jab
- Fixes in the tutorial and more - ironspider
- Document LISY protocol - jab
- Update example links - GabeKnuth
- Fix Mac install instructions - GabeKnuth
- Typos, Bad English and more - ironspider
- Rotation is counter-clock wise not clockwise - colemanomartin
- Document game variables - cfbenn
- Improve tutorial and fix typos - soraxxo
- Log mesage reference section - jab
- Add score slide to tutorial step 17 - Coleman
- Fix instructions on how to install a specific MPF version 2 - mfulleratlassian
- Improved and test multiball example - jab based on question by Sepp
- Fix typos - nhardt
- Document wire-to-wire connectors - ironspider
- Add wiresheet for 7-segment displays with mypinballs controller - unRARed
- When Two Drop Targets Are Hit Simultaneously How Do I Keep Two Sounds From Playing - qcapen
- Typos, 2, 3 - ironspider
- Notes on Mac install - bowilliams
- Remind users about venv when installing pypinproc - bowilliams

- Document modes in subfolders - pmansukhani
- Wording improvments, grammar fixes, typos, more typos, more grammar, simple past, proper count - ironspider (a lot of fixes)
- More precise description - ironspider
- Add modern Stern Opto Trough - ironspider
- Fix segment_displays in shows - snux
- Document LISY35 flipper enable - jab based on question by Dave
- Document local outputs on the P-Roc when using PDB boards - jab
- Update LISY procotol - jab
- Add LISY35 to WPC section - jab
- Document machine variables and more - jab
- Add images for coils, buttons, flasher, up-down-ramps and diverters - kevinleedrum
- Improve skill shot documentation - jab
- Improve service mode documentation - jab
- Document text_ui section - avanwinkle
- Fix typos and grammar - catrinaisahuman
- Fix typo in path - arthurlutz
- Added flipper image - tpilewicz
- Documentation (integration) tests with MC to make sure examples always work - jab
- Integration test for shots and widgets - jab
- Remove Python 3.4 references from docs - cfbenn
- Upgrade instructions for old to new kivy version - jab
- Document numlock keyboard issue - mwiz
- Document common problems with OPP on Ubuntu - jab
- Extend APC documentation - jab
- Document how to install MPF Spike bridge with FW 0.49+ - densminger
- Improve OPP docs - jab
- APC documentation - jab
- Document how to use newer Spike 1 firmwares with MPF - densminger
- Typo - jab
- Show config tests in docs - jab
- Example for other player scoring - jab

0.52

Released: February 02, 2019

This is a 0.51 maintenance release with cleanups and some refactorings. There should not be any breaking changes but a lot of bug fixes.

MPF

New Features

- OSC platform to control external lights - jab based on request in forum
- Validate variables in variable_player - jab based on config in example
- Placeholders for shots and shot_groups - jab based on question from mike wiz
- Better error messages for placeholders - jab
- Show proper error when fadecandy server is not running - jab based on request from Brian Cox
- Nicer output on startup errors - jab
- Show shutdown reason on exit of MPF - jab
- Show import error for pinproc - jab
- Upstream Raspberry Pi DMD support - jab based on external platform from Michael Betz
- Support for Spike Trough via SPI Bit Bang - jab
- Move libpinproc to a separate thread - jab
- Score Queues for SS style scoring - jab based on request in forum
- Check for OPP firmware mismatch on start - jab based on bug report in forum
- Evaluate placeholders from service cli - jab
- Improve USB latency for I2C in pypinproc - jab based on suggestion by rosh
- Only enable AC relay by default during the game. Keep it off in attract - snux
- Ball Routing device to route balls to certain devices - jab
- Support for the Pololu Tic stepper controller - wolfmarsh
- Update Smartmatrix Teensy Code Example for New Cookie - aaronmatthies and eli
- Placeholders in event_player based on event parameters - avanwinkle
- Update ruamel yaml parser - jab
- Use newer cython to support Python 3.7 - jab
- Add Python 3.7 support to MPF - jab

Bug fixes & code improvements

- Fix audio problems - jab (based on 0.50 fix)
- Fix name clashes between multiple anonymous slides - jab based on bug report by pinballpeople

- Properly support external platforms in MC - jab based on report by TheLegoMoviePinball
- Honour -a and -A option when loading config_spec in MPF and MC - jab based on report by TheLegoMoviePinball
- Honour slide parameter in inactive slides - avanwinkle
- Fix iMC startup crash - jab based on report by snux
- Remove use_sound_setting from default options - avanwinkle

MPF-MC

New Features

- Add a segment display font - jab based on example from BorgDog
- Conditionals on add_to_slide animations - avanwinkle

Bug fixes & code improvements

- Fix audio problems - jab (based on 0.50 fix)
- Fix name clashes between multiple anonymous slides - jab based on bug report by pinballpeople
- Properly support external platforms in MC - jab based on report by TheLegoMoviePinball
- Honour -a and -A option when loading config_spec in MPF and MC - jab based on report by TheLegoMoviePinball
- Honour slide parameter in inactive slides - avanwinkle
- Fix iMC startup crash - jab based on report by snux
- Remove use_sound_setting from default options - avanwinkle

MPF Documentation

- How to change the size of switches and light in the MPF monitor - jab based on questions from Jack Danger and Dan
- Document StepStick stepper drivers in MPF - jab based on request from Tom
- How to show virtual segment displays in MC - jab based on example from BorgDog
- How to use multiple displays - jab based on question in forum by Chris B and Snux
- Credits mode tutorial - jab based on old tutorial
- Tutorial on debugging memory leaks - jab based on question from Brian Cox
- Document RPi DMD platform - jab
- How to subscribe variables in config players - jab based on question
- Documenting the snux platform - snux
- How to use a Stern Spike Trough in other platforms than Stern Spike - jab
- How to use Solid State Style Score Queues - jab based on request in forum

- Document event handler priorities - jab
- How to use multiple locks in a multiball - jab
- Monitorable properties for shots and shot_groups - jab based on question by snux
- Document recycle settings for more platforms - jab based on question by Cole M
- Explain logic and modes in MPF - colemanomartin
- Notes on case-sensitivity - colemanomartin
- Explain A and C side preference in System11/Snux - snux
- Fix typos - travisbmartin
- Document monitorable properties and event in logic blocks - jab
- Example for conditionals in log - jab
- Update Smartmatrix documentation for new cookie - aaronmatthies
- Document start/launcher/tournament buttons - jab
- Document part numbers and voltages for bulbs, flashers, GIs and popbumpers and LEDs - jab
- Up-Down ramps - jab
- Updated Mac Install Instructions - avanwinkle
- Image for WS2812 LEDs - kylenahas

0.51

Released: November 24, 2018

This is a 0.50 maintenance release with cleanups and some refactorings. Breaking changes in common features are minimal but some minor changes might be required in some cases (e.g. we removed some defunctional options). It comes with lots of performance improvements and new settings for production machines.

MPF

New Features

- Configurable match number - jab
- Support I2C on the RPi via pigpio - jab
- Improve event order - jab
- Refactor accelerometers - jab (breaking change)
- Support burst IRs and local inputs/outputs on the P3-Roc - jab
- Validate P-Roc direct input numbers - jab
- Rename scriptlets to custom_code - jab
- Add json logging - muffler-aus
- Improve startup performance - jab

- Allow lists of flashers - avanwinkle
- Prevent spaces in event handlers - avanwinkle (breaking change)
- Allow float in timers - jab
- Major performance improvements for switch handlers - jab
- Major performance improvements in lights and shows - jab
- Add option to disable sound output - avanwinkle
- Support multiple I2C servo controllers - jab (breaking change)
- Improve performance without logging - jab
- Add support for P3-Roc burst optos - jab
- Allow users to disable ball search rounds - jab
- Define alignment for segment displays - jab
- Add restart_events to shots and shot groups - avanwinkle
- Add placeholder support to event_player - avanwinkle
- Prevent warnings during init and batch incandescant update for OPP - jab
- Improve FAST behaviour during MPF init - jab
- Entrance switch ignore window - avanwinkle
- Improved README.md for the MPF project - austinbgill
- Prevent bad switch config for drop_targets, shots and autofires - jab
- Validate that ball_count for multiballs is the right range - jab based on question from Alex
- Allow variable_players outside game modes for machine variables - jab
- Only reset drop target banks if a target is down - jab based on request from Mark M
- Add support for flipper tapping for OPP - jab and Hugh based on forums discussion
- Serial LEDs support for PD-LED - jab with help from gstellenberg
- Only send lamp updates when lamps change in LISY - jab
- mpf test can now parse example/tests from rst files - jab
- sw_flip_events and sw_release_events for flipper to flip from software - jab based on request from Philip D
- Add event handlers to start game and add players - jab based on request from Cole M
- Add new mode_will_start hook for custom code - Lamoraldus based on discussion in forum
- Support external platforms via entry_points - jab
- Refresh Smartmatrix DMDs periodically - jab
- Support Servos on PD-LED - jab with help from gstellenberg (announcement)
- Support Steppers on PD-LED/New stepper device interface - jab with help from gstellenberg
- Support config specs for external platforms via entry_points - jab

Bug fixes & code improvements

- Prevent crash on empty machine vars in MC - jab
- Sync shows with sync_ms on stop - jab
- Fix pulse on drop target reset - jab
- Prevent flicker on show replace - jab
- Fix logging verbosity - avanwinkle
- Fix placeholder crash - jab
- Restore diverter state after ball search - jab
- Fix debug flag in P-Roc and P3-Roc - jab
- Prevent achivements from enabling after restoring state - avanwinkle
- Fix ms vs sec in timer pause - avanwinkle
- Fix mode events when starting/stopping mode from BCP - jab based on report by Travis Martin
- Fix display_light_player crash when used in mode - jab
- Fix crash in BCP with MPF Monitor - jab based on report from alex
- Fix pulse calculation error in Stern Spike - jab
- Actually use poll_hz in lisy section - jab
- Prevent broken flipper rules when using multiple flipper devices in FAST/OPP - jab
- Prevent lags in LISY - jab

MPF-MC

New Features

- Disable multi touch - qcapen
- Add json logging to MC - mfulleratlassian
- Improve startup performance - jab
- Add animations based on event parameters - jab
- Add option to disable sound output - avanwinkle
- Rename mc_scriptlets to mc_custom_code - jab
- Support other channel orders than RGB for all RGB DMDs - jab based on request from Cadrion
- Update kivy to version 1.10.1 - jab
- Support multiple (stacked) style values for widgets - avanwinkle
- Better error when showing images too early - jab based on question from Brian C
- Allow widget styles to set z values - avanwinkle
- Update kivy dependencies - jab
- Reusing named widgets - avanwinkle

Bug fixes & code improvements

- Properly update text widgets on text change - MarkInc666
- Fix crash on empty machine var - jab
- Reset animation on remove of image - jab
- Fix iMC crash - jab
- Fix widget leaks - jab
- Fix playlist crash - qcapen
- Fix that you cannot edit the last highscore character - jab
- Prevent multiple text handlers/Improve performance - avanwinkle
- Fix depreation warnings in kivy scale - avanwinkle
- Fix iMC initialisation - avanwinkle

MPF-Monitor

New Features

- Add config option for device size in monitor - jab
- Improve monitor performance - jab

Bug fixes & code improvements

- Fix bcp crashes - jab
- Obey machine path - John

MPF Documentation

- Document state_machines - jab
- Document hardware_sound_player for older machines - jab
- Document bitmap_fonts - qcapen
- Document motors and digital_outputs - jab
- Document SPIKE DMDs - jab
- Example for logic blocks - jab
- Add documentation on game design - jab
- Update I2C accelerometer documentation - jab
- Add mode examples - jab
- Improved windows install instructions for the monitor - sliderpoint
- Document burst IR and direct inputs/ouputs on the P3-Roc - jab
- Fix smartmatrix documentation - driskel

- Document tilt mode - jab
- Document conditionals and placeholders - jab
- Document multipliers in scoring - jab
- Document color correction - jab
- Document spinners - jab
- Document shows on ball start/end - jab
- Document bonus mode - jab
- Howto on ball save on ball start - jab
- Document high score mode - jab
- Document MPF service cli - jab
- Document credits mode - jab
- Document common machine types - jab
- Document LISY - jab
- Document common modes in MPF - jab
- Add RPi debug notes on sound/video - matirwin
- Document match mode - jab
- How to use udev to ensure persistent devices on linux - jab
- Document text placeholders - jab
- Add examples for animations based on player vars - jab
- Add light examples - jab
- Clarify monitorable properties - avanwinkle
- Added a guide on mode layering - avanwinkle
- Document how to run MPF in production - jab
- Improve light_strips, ball_holds, image_widgets, widget_styles, switch_player, drop_target_banks, drop_targets, logic_blocks, coil_player, counters, switches, ball_devices, PSUs, coils, smart_virtual_platforms, multi_balls, light_rings and more - jab
- Document volatages in pinball machines - jab
- Documentation about EMC/EMI and common ground - jab
- Document FAST power filter board - jab
- Document Multimorphic power entry board - jab
- Document servo sequences - jab
- Images for targets and FAST and Multimorphic, drop_targets and optos, switches, spinners and magnets, vari-targets - with help from the fast slack
- Add part numbers for optos and switches - jab
- Add common PSU part numbers - jab
- Document uninstall - colemanomartin

- Document how to cancel a show using flipper_cancel events - mwiz
- Document wiring and voltages - jab
- Mode corrections - mwseiden
- Document electrical details of optos - jab
- Update shot group profiles documentation - avanwinkle
- Document how to use player variables with counters - mwseiden
- Document appliance classes and common ground - jab
- Added examples for PD-LED - jab
- Document appliance classes and common ground - jab
- Added examples for PD-LED - jab
- Improved bonus mode documentation - avanwinkle
- Document ball and game end mode blocking - jab inspired by Lynn
- Extra ball based on score example - jab based on example from Lynn
- How to use high score mode in EMs - jab based on example from Lynn
- Document RGB DMD channel_order parameter - jab
- Added example of game mode which increases multiplier when lanes are complete - travisbmartin
- No longer claim Python 3.4 support - it is EOL - jab
- Document PC power on/off - jab
- Typos, Typos - travisbmartin
- Improve skill shot example to prevent race condition and add timeout - jab based on question from mike wiz
- Document scoring based on logic blocks - jab based on question from alex
- Describe how to debug crashes with GDB - jab
- How to tune eject_timeouts in ball devices - jab
- Understanding tags in MPF - cfbenn
- Example for using MC with multiple screens - jab based on example from Brian Cox/cfbenn/qcapen
- Document how to use machine and player variables from code - jab
- Document multiple styles for widgets - avanwinkle
- Document how to use start button for mode selection without added new players - jab based on example provided by alex
- Document which hardware rules are used in MPF - jab based on discussion in the forum
- Document Molex KK part numbers for connectors - jab
- Document how to maintain a stable high voltage rail - jab based on suggestion by Hugh in discussion
- Common events and example for shots - jab based on question from Alex

- Autogenerated event lists for events - jab
- Initial documentation for sequence_shots - jab
- Fixed typos - travisbmartin
- Weak flippers mode - jab based question by Brian C and Philip D
- Document how to use widgets from code - cloudjor
- Extend event documentation for game_start - colemanomartin
- Doctor Who carousel example - travisbmartin
- Document sw_flip_events and sw_release_events - jab
- Example game mode with multiple shots which need to be active a the same time - jab and improvements by coleman based on question by Cole M
- Cookbook/tutorial for a super jets mode - travisbmartin
- Document how to send data from MPF to MPF-MC in custom code - cloudjor
- Added a minimal OSC plugin - jab
- Fix typos and links - zach27
- Notes on using multiple playfields - jab based on discussion in forum
- Animating a progress bar - based on discussion in forum
- Adding a picture of a drop target bank - coleman
- Fix typos - travisbmartin
- Update stepper documentation - jab
- Document PD-LED steppers, servos and serial LEDs - jab

Others

New Features

- Experimental external Philips Hue platform - jab based on code from Philip D

0.50

Released: April 23, 2018

MPF

New Features

- Consolidated LEDs, matrix lights, GI, and flashers into a single "light" device. Much cleaner, less code, and unified features across all light types.
- Added RGBA color support (RGB colors plus an alpha channel)
- Hardware fade support for all light (fade-in and fade-out).
- Added segmented displays support

- Added LISY hardware platform support (for Gottlieb System 1 and System 80 machines)
- Added MyPinballs 7 segment display support
- Added P-Roc alphanumeric displays support
- Added Raspberry Pi as a platform (remote via ethernet or local using pigpio)
- Added stepper motor device
- Added motor device (with position and/or end switches)
- Added Trinamics Steprocker platform
- Added SPIKE DMD support
- Support for FAST RGB DMD support
- Added digital output support (either mapped as drivers or lights)
- Added native I2C support on linux (via SMBus)
- Added NXP MMA8451 accelerometer support (via I2C)
- Support fuzz testing (to find crashes in a machine without playing it)
- Added PSU support to manage maximum power usage. Coil pulses can specify a maximum delay which is used to reorder pulses (used by ball devices, score reels and drop targets).
- Improved and broke out game lifecycle events (will start, starting, started, etc.) for game, ball, and turn starts and stops.
- Made many more settings "templatable"
- Logging to syslog
- Cleaned up and simplified shots
- Added Text UI
- Added replay credits
- Added developer documentation website (developer.missionpinball.org)
- Added support for custom named colors
- Added pluggable ejectors and ball counters in ball devices
- Added "mpf service" command to spawn a service cli (similar to service mode or SPIKE game cli)
- Added "mpf hardware scan" to enumerate all hardware platforms
- Added "mpf hardware update_firmware" to send firmware updates to all hardware platforms

Bug fixes & code improvements

- Support for Python 3.5 and 3.6 on Windows (including P-ROC libraries)
- Much more type checking
- Improved logic around how playfields are marked active
- Improved how device monitors work
- Improved and added config template values

- Improved multiball locks
- Improved machine variable internals
- Improved ball tracking
- Improved ball handling in ball devices
- Improved Stern SPIKE platform
- Refactored mode device loading, config validation, and config player loading
- Renamed “scoring” to “variable_player”
- Improved high score mode
- More robust score reels
- Performance improvements for fadecandy LED updates
- Performance improvements for smartmatrix devices (separate sender thread)

MPF-MC

New Features

- Major display refactoring
- Bitmap fonts
- Relative animation values
- Added widget rotation & scale animations
- Animation values respect initial anchor points
- Simplified, consolidated, & unified DMD, color DMD, and slide frame widgets into displays and display widgets
- New ‘sound_loop’ audio track type optimized for live looping music control driven by events. This specialized audio track type can synchronize playback of multiple looping sounds simultaneously in layers and provides gapless switching to a new set of loops. It is designed to build music that dynamically changes based on events in your game. Only supports in-memory sounds (no streaming).
- New ‘sound_loop_set’ asset type. A sound_loop_set is an asset used to play sounds in a sound_loop track that is basically a grouping of one or more sound assets. The sounds in a loop set are arranged in layers. The master layer contains the sound that establishes the length of the entire loop set. Whenever the sound in the master layer loops, all other sounds in the sound_loop_set will also loop back to the beginning.
- New ‘sound_loop_player’ config_player. The sound_loop_player is a config player that is used to control the playback of sound_loop_sets in a sound_loop audio track. The track_player can also be used with a sound_loop track to control volume and playback state.
- New ‘playlist’ audio track type is designed to provide a comprehensive set of music playing capabilities that include named playlists, playback mode (sequence, random, etc.), cross-fades between sounds/songs/playlists, and more.
- New ‘playlist’ asset type. A playlist is an asset used to group and play sound assets on a playlist track. A playlist is basically an ordered group of sounds/songs typically used to playback music.

- New 'playlist_player' config player. The playlist_player is a config player that is used to control the playback of playlists (and their component sounds) in a playlist track. The track_player can also be used with a playlist track to control volume and playback state.
- New sound 'about_to_finish' events (configurable for each sound). These post events at a specified time before the sound ends.
- New display_light_player to use your playfield lights as display in MC. Also supports transparency to overlay a graphic/animation above your light shows.

Bug fixes & code improvements

- Support for Python 3.5 and 3.6 on Windows
- Significant performance improvements
- Fixed many leaks (especially widgets)
- Animation steps can be run simultaneously
- Bail out when a video codec is missing
- Refactored the entire audio engine code (broke audio_interface.pyx into many different files, individual source files for each track type and base class, eliminated .pxi files and established use of .pxd files)
- Switched back to SDL_Mixer for main audio playback, mixing, and in-memory sound asset loading functions (provide more reliable and faster loading of .ogg and .flac files)
- Allow unlimited sound asset event markers (previously only allowed a fixed number)

MPF-Monitor

New Features

- Device list shows all monitorable attributes

Bug fixes & code improvements

- Improved performance of light updates/Smooth light shows

0.33

Released: April 10, 2017

MPF

New Features

- "Ball hold" device (Temporarily hold a ball while something else is happening)
- "Multiball lock" device (Track ball locks towards multiball, including virtual locks, across balls and players)

- Multiball "add a ball" feature
- Added support for Stern SPIKE platform
- *Revamped logging*
- Additional achievements control events
- BCP ports & interfaces are now configurable
- Drop target "keep up" feature (PWMs reset coil to "lock" target up)
- "Async" events (Events that wait for all handlers to finish before continuing)
- Additional multiball events
- More functions for people building games to use to write tests
- Built-in modes with code can have their code overloaded
- Added score reels to the smart virtual platform
- Allow machine variables to be set via BCP
- Allow setting default high scores
- Add "early save" events to ball saves
- Add all monitorable device properties to conditional events
- Use placeholders in mode timer start & end values
- More options for bonus (hurry ups, skip slides with 0 value, placeholders for score calculations, etc.)
- Improved ball search
- OPP - support for firmware 2.0 and dual wound coils
- MC scriptlets for video modes and code on the MC side
- Support for conditional events
- Template variables which are evaluated during runtime and can use placeholders (timers, logic_blocks, tilt, scoring, bonus_mode, and more)
- Early ball save
- Advanced bonus_mode
- TimedSwitch device - built-in event for flipper cradling and releasing
- Asynchronous logging - This is especially important on windows because logging previously slowed down the game. However, also important in production when under high I/O load or with slow discs.
- Timers work outside of the game now
- New "mpf diagnosis" command
- Scoring to machine variables
- Scoring for other players
- Weights in random_event_player
- Unlimited delay in ball_save to allow video modes or mode selection

- Added Machine vars for all kinds of versions
- Drop Target keep up support
- Multiball add a ball support
- New multiball_lock device which handles virtual saves for multiplayer game
- Allow BCP to bind on all IPs

Bug fixes & code improvements

- A lot of miscellaneous bug fixes
- Exiting service mode always put the machine back on free play
- Fixed a ball lock crash
- File loader will not try to load temp files
- Manual plunger in smart virtual platform now works properly
- Refactored ball devices to allow for different types of ball counters & be more robust for unexpected ball situations and different types of eject failures
- Made achievements and achievement groups smarter and more robust (also backported to 0.32)
- Improved log messages for BCP encoding errors
- “Hz” setting is gone (since MPF is now tickless)
- Active eject process trackers are canceled on shutdown
- Randomizer now works with a single element
- Fixed a bunch of small things that caused crashes
- Changed default on-screen DMD pixel settings
- Removed OSC plug-in since it hasn’t worked in over a year and no one uses it
- Better errors on invalid configs
- Catching a lot more config problems
- Improved ball search. Drop Target reset no longer resets ball search
- Better start/stop procedures for modes. no more event races
- Improved extra ball
- Better yaml parsing for unescaped strings
- Performance improvements through better fast paths and offloading of logging from the synchronous path
- BCP version 1.1 with synchronisation during reset
- Improved handling of ball devices with entrance_switch
- Force UTF-8 for configs on windows
- Better errors when loading assets

MPF-MC

New Features

- Added a camera widget (live video)
- Allow placeholders and settings
- Added keyboard debugging
- Added warnings if window size & display size aspect ratios are not the same
- MPF-MC now checks to make sure the MPF version it's talking to is compatible
- Change the default display size to 800x600 if a displays: section is not in the config
- Re-vamped Mac installation procedure. It's now a "real" install and does not use MPF.app anymore.
- Added a "volume" machine variable
- Added Interactive Media Controller (iMC)
- Added "anchor_y: baseline" option for text widgets
- Added gamma setting for physical DMDs
- Added new relative animation target values

Bug fixes & code improvements

- Improved sound asset loading speed (uses SDL_Mixer for loading to memory rather than GStreamer)
- Sound assets can be loaded while videos are playing
- Sound assets can be located in sub-folders as many levels deep as desired (not just a single level)
- Fixed points widget
- Improvements to automated testing on Travis
- widget_player positioning fixed
- Better error messages for malformed slide configs
- Prevent crash in text widget when empty and back is selected
- Changes to support BCP 1.1

0.32

Released: Dec 1, 2016

MPF

- Improved *achievements* and added *achievement groups*.
- Added relay events and relay queues
- Improved *smart virtual platform*

- Improved support for *System 11* and *Gottlieb System 3 style* troughs (including using the ball drain as a ball storage location to get one additional ball capacity with no hardware changes).
- Verify that duplicate sections don't exist in config files
- Check that event handlers are properly formatted before they're registered
- Added conditional events (handlers that only fire if certain conditions are met)
- You can *set starting values for player variables*
- Fixed the *physical mono DMD* and *physical RGB (color) DMD*
- Added *multiball lost event*
- Allow devices to have inline config specs
- Added shots with events
- Better OPP platform parsing
- Fixed & improved the high score mode
- Improved service mode
- Added options for "random" events (force next, force all, save per-player, etc.)
- Added events to the BCP monitor (meaning they can be viewed in the MPF Monitor app)
- Added `-f` command line option to force all assets to load on boot for testing purposes
- Added scoring options (add, replace, block)
- Use color "on" for LED default colors
- Allow multiple config player entries to fire from the same event
- Ensure that events created by the MC are sent to MPF
- Added machine vars for P-ROC and FAST hardware revisions
- Added *combo switches* (for "flipper cancel", two-button skill shots, etc.)
- Lots of little bug fixes. . .

MPF-MC

- Fixed the widget z-order layering bug (this has been backported to 0.31). Widget orders are now higher value z: settings are on top of lower value ones.
- Negative z: values are no longer used to target parent slide frames. Instead, `target: (name)` is used.
- Cleaned up debug logging so BCP frames are not included in it by default
- Events that are natively posted in the MC are now sent to MPF
- Fixed a bug to ensure that the slide_active event is only posted once per frame
- Fixed a bug that prevented slide frames from being animated
- Fixed a bug where videos were not stopping
- Allow the same slide to be used on multiple displays

- Switch to GStreamer instead of SDL_Mixer for loading and streaming sounds. (SDL2 still used for all sound output.)
- Sound file streaming is now supported from any track (streamed from disk instead of preloaded into memory)
- New “track_player” config controls sounds at the track-level (fade, volume, play, pause, stop, etc.)
- Custom loading & unloading events at the individual sound level.
- Lots of little bug fixes. . .

0.31

Released: Sept 19, 2016

MPF

- MPF is now “tickless”, meaning everything runs faster, but with less overhead
- Improved flow control for FAST hardware serial communication
- Improved BCP communications
- Improved serial communications for all devices which use serial
- Additional options for ball saves
- Removed many threads which makes everything simpler and faster under the hood
- Improved “virtual” and “smart virtual” platforms
- Prevent broken data files from crashing MPF
- Added a basic service mode (this is just a start, much more to come)
- Detect balls that jump between playfields
- Prevent duplicate rules being written to P-ROC and P3-ROC controllers
- Allow mode config files to be broken into multiple files
- Allow multiple multiball modes to run at once and add options for how it tracks them
- Allow ball locks to wait for a ball to drain before releasing their locked balls
- Added the ability to use matrix lamps/LEDs at individual channels for RGB LEDs
- Re-added high score mode (Which was in 0.21 and removed in 0.30)
- OPP platform improvements
- Improved error messages for config file errors
- Improved the way the “mpf both” command works on all platforms
- Added ability to step backwards in shows
- Refactored and improved show player
- Added ball search for servos
- Added default colors to RGB LEDs

- Added support for nested shows
- Added the “LED Group” device (am easily-configured strip of LEDs which can be strobed, pulsed, etc.)
- Added kickback mechanisms
- Added magnets
- Added blocking show queues
- Many bug fixes. . .

MPF-MC

- Audio library improvements (sound fading, markers, start position, instance limiting, ducking improvements)
- Allow widget events based on when slides are shown, hidden, etc.
- Improved error if you try to target a widget to an invalid slide
- Added default DMD fonts
- Many bug fixes. . .

0.30

Released: July 15, 2016

- Python 3 required
- Mac OS X support
- The Media Controller is now a separate package from MPF
- The MPF-MC has been completely rewritten from scratch (based on Kivy, SDL2, OpenGL, and Gstreamer)
- GPU is used for graphics
- Brand-new audio interface specifically written for pinball audio, which includes advanced feature like ducking, attack, attenuation, etc.
- Proper Python package installers, and inclusion in PyPI so install can be done via *pip*.
- System-wide *mpf* launcher utility with pluggable commands
- New MPF clock module replaces the old timing and timers
- All shows are driven by MPF
- Show content is “played” by the standard config_players
- Playlists become shows
- “Tocks” are gone, shows now operate on real-world time
- Light scripts are gone, replaced by placeholder “tokens” in shows
- Named colors
- Hardware accelerated LED fades

- Asset Pools
- Ball Search
- Accelerometer-based tilts
- Servo support
- Text string support
- Player achievements

0.21

Released: Dec 1, 2015

- SmartMatrix "real" RGB LED Color DMD support.
- System 11 support.
- High Score mode.
- Credits mode.
- Tilt mode.
- Smart virtual platform. (This is the new default platform.)
- New display elements: Character Picker and Entered Characters.
- Devices can be created and changed per mode.
- Machine variables.
- Untracked player variables.
- Central config processor, data manager, file manager, and file interfaces. This paves the way for config files in formats other than YAML.
- Added support for combo manual/auto plungers.
- Events for ball collection process.
- Driver-enabled devices.
- External light shows, controllable via BCP. (Thanks Quinn Capen!)
- Created a starter game machine config template you can use for your own machines.
- Started adding unit tests. (We're at the very beginning of this, but we have full coverage of the ball device, the event manager, and the tutorial configuration files.)
- Rewritten driver/coil device interface.
- Rewritten ball device and ball controller code. (Thanks Jan Kantert!)
- Rewritten score controller.
- Rewritten display & slides modules.
- Many improvements and features added to ball saves.
- Python 2.7 is now required. (Previous releases would also run on Python 2.6)
- Logic blocks can now persist between balls

- Fixed & enhanced the asset loading process.
- Many improvements and features added to modes and the mode controller
- Multiple config files can be chained together at the command line
- Improved text display element.
- Improved event manager and event dispatch queue
- Moved all utility functions to their own class.

0.20

Sept 14, 2015

- The *targets* and *shots* modules have been combined into a single module called *shots*.
- The new shots module adds several new features, including:
  - Shots can be members of more than one shot group, and added and removed dynamically.
  - Sequence shots can track more than one simultaneous sequences. (e.g. two balls going into an orbit at essentially the same time will now count as two shots made.)
  - Shots are mode-aware and will automatically enable or disable themselves based on modes starting and stopping.
- Modes now work outside of a game.
  - "Machine modes" have been removed. Attract and game machine modes are now regular modes.
  - This makes it easier to have always-running modes (volume control, coin door open, coin & credit tracking).
  - This makes it possible to configure custom branching of mode-flow logic. (i.e. long-press the start button to load a different game mode, etc.)
- Significant performance improvements for both starting MPF and starting a game:
  - Reading the initial states of switches on a P-ROC is significantly faster.
  - The auditor now waits a few seconds before writing its audit file, and it does it as a separate thread. Previously this was slowing down the game start and player rotation events.
  - The way modules that need to track "all" the switches (like the auditor and OSC) was changed and now it doesn't bog things down.
- A device manager now manages all devices. (This will enable future GUI apps to easily be able to browse the device tree.)
- Devices can be "hot added" and removed while MPF is running. This includes automatic support to add and remove devices per mode.
- All device configuration is specified and validated via a central configuration service. This has several advantages:
  - The config files are now validated as they're loaded. For example, if there a device has a settings entry for "switches", MPF will now validate that the strings you enter in the are actual switch names. It will give you a smart error if not.

  - This paves the way for supporting config files in formats other than YAML. (JSON, XML, INI, etc.)
  - This led to the removal of about 500 lines of code since all the config processing was done manually in each module before.
  - The config processing is more efficient and less-error prone since it's not written from scratch for each module.
  - There's now a master list (in *mpfconfig.yaml*) of all config settings for all device types.
  - The config processor and validator can run as a service to support the back-end business logic behind future GUI tools which could be used to build machines.
  - If you're configuration has an unrecognized setting, the config validator will load the config file migrator to tell you what the updated name is for the section it doesn't recognized.
- Shot rotation has been improved:
  - You can now specify the states of shots you'd like to include or exclude. (i.e. only rotate between incomplete shots.)
  - You can specify custom rotation patterns (i.e. a "sweep" back-and- forth instead of a simple left or right rotation)
- A ball lock device was added to make it easy to specify ball locks.
- A multiball device was added.
- A simple ball save device was added.
- Created a "random_event_player" that lets you trigger random events based on another event being posted.
- Centralized debugging
- Drop targets and drop target banks have been simplified and separated from shots.
- The states of switches tagged with 'player' will be passed to the game start mode, allowing branching based on which combinations of switches were held in when the start button was pressed. (The amount of time the start button was held in for is also sent.)
- Official support for multiple playfields via config files
- Added x, y, and z positions to lights and leds
- Exposed wait queue events to mode configs, allowing code-less creation of modes that can hook into game flow (bonus, etc.)

0.19

Released: August 6, 2015

- Completely rewritten target and drop target device module, including:
  - Per-player state tracking for targets
  - Target "profiles" that control how targets behave, completely integrated with the mode system
- Light show "sync_ms" which allows new light shows to sync up with existing running shows.
- Timed switch events can be set up via the config files.

- Added “recycle_time” to switches. (Switches can be configured to not report multiple events until a cool-down time has passed.)
- Created an events_player module
- Player variables in slides automatically update themselves when they change. (No more need to find an event to tie the slide to in order for it to update!)
- Device control events exposed via the config files
- Automatic control of GI
- Activation and deactivation events can be automatically created for every switch.
- Allow multiple playfield objects to be created at once (for head-to- head pinball)
- Added support for FAST Pinball’s new WPC controller
- Added a Linuxshell script to launch mc.py and mpf.py
- Created the config file migration tool
- Added per-timer debug loggers
- Standardization of many non-standard config file naming conventions
- Color logging to LEDs
- Added P3-ROC switch test tool
- Added reset to mode timer action list
- Added restart feature to mode timers
- Flipper Device: Add debug logging to rules
- FAST:Added minimum firmware version checking for IO boards
- Added “restart” method to logic blocks
- Text display element min_digits
- Allow system modules to be replaced and subclassed
- Added configurable event names for switch tag events
- Added callback kwargs to switch handlers
- Added light and LED reset on machine mode start
- Added default machine and mode delay managers

0.18

Released: June 2, 2015

- FadeCandy and Open Pixel Control (OPC) support. This means you can use a FadeCandy or other OPC devices to control the LEDs in your machine.
- Rewritten FAST platform interface. It’s now “driverless,” meaning you no longer need to download and compile drivers to make it work.
- Added support to allow multiple hardware platforms to be used at once. (e.g. LEDs can be from a FadeCandy while coils are from a P-ROC.) You can even use multiple different platform interfaces for the same types of devices at once (e.g. some LEDs are FadeCandy and others are FAST).

- Added support for GI and flashers to light shows
- Added activation and deactivation events to switches
- Added support for sounds in media shows
- Added per-sound volume control
- Added support for P-ROC / P3-ROC non-debounced switches
- Exceptions and bugs that causeMPF to crash are now captured in the log file. (This will be great for troubleshooting since you can just send your log. No more needing to capture a screenshot of the crash.)
- If a child thread crashes, MPF will also crash. (Previously child threads were crashing but people didn't know it, so things were breaking but it was hard to tell why.)
- MPF can now be used without switches or coils defined. (Makes getting started even easier.)
- "Preload" assets loading process is tracked as MPF boots, allowing display to show a countdown of the asset loading process
- Added *restart_on_complete* to mode timers
- Smarter handling of player-controlled eject requests while existing eject requests are in progress
- *eject_all()* returns *True* if it was able to eject any balls
- Playfield "add ball" requests are queued if there's a current player eject request in progress
- Created a smarter asset loading process
- The attract mode start is held until all the "preload" assets are loaded
- Updated how the game controller tracks balls in play

0.17

Released: May 4, 2015

- Broke MPF into two pieces: The MPF core engine and the MPF media player
- Added support for the Backbox Control Protocol (BCP)
- Added device-specific debugging for LEDs.
- Added version control to config files.
- Added volume control.
- Switches that you want to start active when using virtual hardware are now added to the *virtual platform start active switches:* section instead of being a property of the *keyboard:* entry.
- Converted several former plugins to system modules, including shots, scoring, bcp, and logic blocks.
- General performance improvements. (Running MPF on my machine used to take about 50% CPU. Now it's down to 15%.)

0.16

Released: April 9, 2015

- Added slide "expire" time settings to the Slide Player.
- Added *Demo Man* as the sample game code.
- Added start_time configuration parameter for music in the StreamTrack
- Added the SocketEvents plugin
- Created the LightScripts and LightPlayer functionality.
- Change light script "time" to "tocks"
- Created a centralized config processing module

0.15

Released: March 9, 2015

- Added support for game modes.
- Converted several existing modules to be mode-specific, including:
  - LogicBlocks
  - SoundPlayer
  - SlidePlayer
  - ShowPlayer
  - Scoring
  - Shots
- Created an Asset Manager and converted the images, animations, sound, and show modules to use it instead of each handling their own assets.
- Created an asset loader which creates a background thread to load each type of asset.
- Added an AssetDefaults section to the asset loader to specify per- folder asset settings
- Created a universal player variable system
- Added movie support (for playing MPEG videos on the LCD and DMD). They're available as a standard display element type which means they can be positioned, layered as backgrounds, etc.
- Created a generic ModeTimers class that can be used for timed modes and goals. (With variable count rates, support for counting up and down, multiple actions which can start, stop, pause, and add time, etc.)
- Changed logic blocks so they maintain all their states and progress on a per-user basis.
- Added a "double zero" text filter. (Used to show zero-value scores as "00" instead of "0".)
- Updated the display code so that it doesn't show a slide until all that slides assets have been loaded.
- Renamed the "sphinx" folder to "docs".
- Broke the three phases of machine initialization into 5 phases.

- Created the mode timer
- Renamed the "HitCounter" logic block to "Counter" and updated it to be more flexible so it can track general player-specific counts (both up and down), for example, total shots made, combos, progress towards goals, etc.
- Changed window section of config so it uses the slide builder.
- Added the ability to control lights and LEDs by tag name in shows.
- Modified the switch controller so events from undefined switches simply log a warning rather than raises an exception and halting MPF.

0.14

Released: February 9, 2015

- Completely rewritten ball controller.
- Completely rewritten ball device code.
- Major updates to the diverter device code.
- Creation of a new playfield module that's responsible for managing the playfield and any balls loose on it.
- Completely rewrote the "player eject" logic. (This is what happens when the game needs to wait for the player to push a button to eject a ball from a device.)
- The ball search code was moved from the game controller to the playfield device module.
- Different types of events were broken out into their own methods. For example, to post a boolean event, instead of calling *event.post(type='boolean')*, you now use *event.post_boolean()*. There are similar new methods for other event types, like *post_relay()* and *post_queue()*.
- Added a debug option for ball devices which enables extra debug logging for problem devices.
- Tilt status was removed from the machine controller. (It was inappropriate there. Tilt is a game-specific thing, not a machine- specific thing.)
- Virtual Platform: default NC switch states fixed

0.13

Released: January 16, 2015

- Major update to the sound system, including:
  - Support for multiple sound tracks ("voice", "sfx", "music", etc.), each with their own channels, settings, volume, etc.
  - Using background threads to automatically load sound files from disk in the background without slowing down the main game loop.
  - Support for streaming sounds from disk versus preloading the entire sounds in memory.
  - Support for sound priorities and queues, so sounds can pre-empt other sounds if they have a higher priority.
  - System-wide volume control with settable steps.

- Support for the v1.0 update of FAST Pinball's libfastpinball library. (Basically we updated the FAST platform interface to support their latest firmware and drivers)
- Support for flashers. (Previously flashers were just driven like any other driver. Now they are their own device with their own flasher- specific settings.)
- Game Controller: Changed the player rotate routine to be driven from the game_started event so the player object isn't actually set up until the game has finished being set up.
- Pygame: Moved the Pygame event loop to the machine controller and out of the window manager. This lets us use Pygame events even if we don't have an on screen window. (This is needed for the sound system.)
- Display: Moved the SlideBuilder instantiation earlier in the boot process so it's available to other modules who want to use it when they're starting up. This will let us get the "loading" screen up earlier in the boot process.
- Switch Controller: Added a method to dump the initial active states of switches to the log. This is needed for our automated log playback utility so it can set the initial switches properly.
- Ball Devices: fixed a typo on the cancel ball request event

0.12

Released: December 31, 2014

- Added full display and DMD support, with support for physical DMDs, on screen virtual DMDs, color DMDs, and high res LCD displays.
- Added transitions which flip between display slides with cool effects.
- Added decorators which are used to "decorate" display elements (make them blink, etc.)
- Added display support to shows so that shows can now combine display and lighting effects
- Added a Slide Builder which can assemble slides from text, image, animation, and shapes from shows and the config files.
- Added a SlidePlayer config setting which can show slides based on MPF events
- Modified the Virtual DMD display element so that it can render on screen DMDs that look more like real pixelated DMDs
- Added a font manager that lets you define font names and specify default settings (sizes, antialias, color, etc.)
- Added TrueType font support
- Added support for stand image types to be displayed on the DMD
- Added .dmd file type support for images and animations
- Addedthe OSC Sender tool
- Added the Font Tester tool
- Added the multi-language module which can replace text strings with alternate versions for multi-language environments and other (e.g. "family-friendly") text replacements
- Improved the diverter devices so they have knowledge of what ball devices and diverters are upstream and downstream, allowing them to automatically activate and deactivate based on where balls need to go.

- Improved the ball device class so ball devices are smarter about how they interact with target devices. (e.g. a ball device will automatically eject a ball if its target device wants a ball.)
- Added support for the P3-ROC
- Added many more events
- Modified displays so they can each have independent refresh rates

0.11

Released: December 1, 2014

- Created a Display Controller module which is responsible for handling all interactions with all types of displays, including DMD, LCD, alphanumeric, 7-segment, etc.
- Created a DMD display module which controls both physical DMDs as well as on screen representations of physical DMDs
- Created a Window Manager, a centralized module which manages the on screen window, including full screen and resizable support
- P-ROC platform interface: Built the DMD control code
- FAST platform interface: Built the DMD control code
- Switched from Pyglet to Pygame
- Created a Sound Controller
- Created a Game Sounds plug-in that lets you control which sounds are played and looped based on MPF events
- Added PD-LED support
- Added support for P3-ROC SW-16 switch boards
- Switch Controller: Added verify_switches() method which verifies that switches are in the hardware state that MPF expects.
- Switch Controller: Adding logging so it can track when duplicate switch events were received
- LEDs: added on() and off() methods and "default color" support
- Ball Device: created _ball_added_to_feeder() and made it so the device watches for a ball entering and will request it if it needs it.
- Changed the command line options so you don't have to specify the .yaml extension for your configuration file
- Changed the command line options so you (optionally) don't have to specify the "machine_files" folder location
- Created default machine_files folder location settings in the config file
- Added support for absolute or relative paths in the command line options
- Added support for X/Y coordinates to LEDs and Lights for future light show mapping awesomeness.
- Created an early, early version of the Playfield Lights display interface which lets you "play" Pygame shows on your playfield lights

- Added system default font support
- Added a player number parameter to the player_add_success event
- Added a default MPF background image for the on screen window
- Added many more default settings to the system default mpfconfig.yaml file
- Virtual platform interface: Updated it so that it works when hardware DMDs are specified in the config files

0.10

Released: October 25, 2014

- Added enable_events, disable_events, and reset_events to devices.
- Removed the First Flips plug-in. (Since the thing above replaces it)
- Added support for network switches and drivers for FAST Pinball controllers.
- Added support for multiple USB connections to FAST Pinball controllers to separate main controller traffic from RGB LED traffic.
- Changed default debounce on and off times to 20ms for FAST Pinball controllers.
- Individual targets hit in target groups will now post events
- Changed the default show priority to 1 so it will restore lights that weren't set with a priority by default
- Driver: Added a power parameter to driver.pulse()
- Score Reel: Added resync events to individual reels
- Score Reel: Changed repeat_pulse_ms config setting to repeat_pulse_time.
- Score Reel: Changed hw_confirm_ms config setting to hw_confirm_time.
- Changed default pulse time for all coils to 10ms
- Coils: (Fast): Added separate debounce_on and debounce_off settings
- Info Lights: Forced game_over light to off when game starts
- LEDs: Added force parameter to the off() method

0.9

Released: October 7, 2014

- Added a "Logic Blocks" plug-in which lets game programmers build flowchart-like game logic with the config files. No Python programming required!
- Created a "First Flips" plug-in which you can use to get your machine flipping as fast as possible. (This was written as part of our Step-by-Step Tutorial for getting started with MPF.)
- Added Tilt and Slam Tilt support. (This is built via our Logic Blocks, so they're very advanced, supporting grouping multiple quick hits as a single hit, settling time (to make sure the plumb bob is not still swinging when the next ball is started, etc.).
- Added Extra Ball / Shoot Again support

- Created OSC interfaces for /audits
- MAJOR rewrite to the ball controller and ball device modules
- Created a non-instrumented optimized software loop which is as lean as possible if you're running your game on a slow computer. (I'm looking at you Raspberry Pi!) Note: other single board computers are fine, like the BeagleBone Black or the ODOID, but man the Pi is slow.
- Added the ability to pull "data" from MPF via the OSC interface, so we can put player scores, ball in player, etc. on an iPhone, iPad, or Android device.
- Added an OSC audit interface so you can view audit data via your mobile device.
- Created an "Info Lights" plug-in which turns on or off lights automatically based on things that happen in the game. (Which player is up, current ball, tilt, game over, etc.) This is typically used in EM games, but of course the plug-in can be used wherever you need it.
- Finished the code for our Big Shot EM-to-SS conversion. This is included as a sample game in MPF, so you can see our config files and
- Logic Blocks which can be helpful when creating your own game.
- Fixed up drop targets to support the new lit/unlit scheme
- Added support for default states to targets and target groups (stand ups, rollovers, drop targets, etc.), including events that are posted when they are hit while lit or unlit, and the ability to light or unlight them via events
- Added Start Button press parameters which are automatically sent to the game when the start button is pressed. This is for things like how long the button was held and what other buttons where active at the time. (Start * Right Flipper, etc.)
- Added a "pre-load check) to plug-ins that allows them to test whether they're able to run before they load and only load if everything checks out. (This means that a plug-in will no longer crash if a required Python module is missing.)
- Added 'no_audit' tag support. (If you add 'no_audit' as a tag to a switch, then the Auditor will not include that switch in the audit logs.)
- Created Action Events for shutting down the machine and added shutdown tag support (so you can cleanly shut down the machine simply by posting and event or pressing a button which is tagged with "shutdown")
- Added performance data logging to the machine run loop (so it now tracks the percentage of time spent doing MPF tasks, hardware tasks, and idle).
- Added a reload() method to Shows which causes that show to reload itself from disk. This is nice for testing shows since you can reload them without having to restart the machine each time.
- Added support for null steps in shows (literally a step that performs no action). This makes it easier to get timing right for music shows.
- Added the ability to force a light or LED to move to a given state, regardless of its current priority or cache.
- Added a method to test whether a device is valid. This will be used for our config file validator
- Added option for restart on long start button press
- Added option to allow game start with loose balls

- Score reels maintain a valid status, allowing other modules to know whether the score reels are showing the right data or not.
- Score reels now post an event when they're resyncing, allowing other modules to act on it. (For example the score reel controller uses this to turn off the lights for a score reel while it's resyncing.)
- Added option to remove all handlers for an event regardless of what their registered **kwargs are.
- Added mpf command line options for verbose to console and optimized loops. (Now we can support different logging levels to the console and log file, meaning you can configure it so you only see important things on the console but you can see everything in the log file.)
- Added light on/off action events
- Added action events and methods to award the extra ball
- Created ball device disable_auto_eject() and enable_auto_eject() methods. This is how we handle player-controlled ejects (like when a ball starts or they're launching a ball out of a cannon).
- Changed scoring from "shots" to "events"
- Changed the hardware rules for clearing a rule so it disables any drivers that were currently active from that rule
- Updated are_balls_gathered() so that if you pass it a tag which doesn't exist, it always returns True
- Added management of switch handlers to machine modes so they can be automatically removed
- Changed switch handlers so they process delays from new handlers that are added
- Removed "standup" target device type (it was redundant with "target")
- Moved auditor, scoring, and shots out of system and into plugins

0.8

Released: September 15, 2015

- Platform support for FAST Pinball hardware
- RGB LED support, including settings colors and fades
- Created target and target group device drivers for drop targets, standups, and rollovers (including events on complete, lit shot rotation, etc.)
- Created an OSC interface to view & control your pinball machine from OSC client software running on a phone or tablet
- Changed our "light controller" to a "show controller" and added support for things other than lights (like coils and events). So now a show can be a coordinated series of lights, RGB LEDs, coil firings, and events.
- Created an "event triggers" plugin which lets you configure series of switches that trigger events, including custom timings, decays, and resets. (We use this for our titlt functionality but it's useful in other ways too.)
- Created the auditor module
- Created an intelligent diverter device driver (with hardware switch trigger integration)

- Created GI device drivers
- Created a system-wide MPF 'defaults' configuration file
- Created templates for new machines, new scriptlets, and new plugins
- Modified the on screen window to become a "real" LCD display plugin.
- Renamed "hacklets" to "scriptlets"
- Created a scriptlet parent class to make them even easier to use
- Broke the hardware module into "platforms" and "devices"
- Major rewrite of how the machine controller loads system modules and devices
- Shows now auto load
- Added the ability to attach handlers to lights so you can receive notifications of light status changes
- Reworked the EM score reel update process to simplify and streamline it

0.7

Released: September 4, 2014

- Support for lights and light shows.
- An on-screen display of game metrics like score, player, and ball number.
- A "hacklet" extension architecture which lets you add python code to finish up the "last 10%" of your game that you can't control via the machine configuration files.
- A formal plug-in architecture which allows easy creation and modification of plug-ins that will survive core MPF framework updates.
- Cleaned up the machine flow and made that controllable via the config files
- Changed the -x command line option so it doesn't use fakepinproc, got rid of the p_roc methods that detected fakepinproc. (Now even with the P-ROC platform it will use our virtual platform interface when no physical hardware is present. This means you don't need pyprocgame to use fakepinproc.
- Changed the command line options to break out machine root from config files
- Moved command line options to their own python dictionary
- Changed time.clock() back to time.time() since clock was not real world which affected the light shows
- Created new events to capture start and stop of machine flow modes
- Added light support to P-ROC platform interface
- Reorganized the machine files into machine-specific subfolders
- Created an int_to_pwm() static method in Timing

0.6

Released: August 19, 2014

- Addition of a Shot Controller, allowing you to configure and group switches which become shots in the machine. (Read more about the concept of shots in our blog post from last week.)
- Addition of a Scoring Controller, allowing you to map score values to shots (and general scoring support for the machine).
- Addition of the Score Reel Controller, Score Reel devices, and Score Reel Group devices for mechanical score reels in EM-style machines. (Details here.)Switched entire framework timing over to real time system clock times (time.clock()) instead of ticks (for delays, tasks, switch waits, etc.)
- Changed ball controller that if it counts more balls than it thought it had, it will invoke ball_found()
- Changed the switch controller so it will ignore new switch events if they come in with the current status the switch already is
- The switch controller will ignore repeat switch events from the hardware if they are the same state that the switch was in before
- Added chime support for EM-style machines
- Changed game_start event to a queue
- Change game_start event name to game_starting (some of these entries might seem trivial, but I also use this list to track the changes I need to make to the documentation)
- Created a queue for adding new tasks so our set won't change while iterating

0.5

Released: August 5, 2014

- Created a single device parent class that's used for all devices.
- Rewrote and cleaned up devices. Now coils, switches, and lights are all devices, as are the more complex ones.
- Added "events" to the keyboard interface. This means you can use the keyboard to post MPF events (along with parameters).
- Separated out ball live confirmation and valid playfield
- Built a bunch of valid playfield methods
- Changed ball_add_live_request from direct calls to events so they'd be slotted in properly
- Broke valid playfield out into its own module
- Made the ball device "entrance" switch work
- Built a quick "coil test" mode
- Added kwargs to event handlers (meaning you can register a handler with kwargs)
- Figured out how to handle the "first time" counts of ball devices

- Added checks to attract mode to make sure all balls are home, and to the ball controller to prevent game start if all balls are not home
- Changed ejects to events. (So if you want to request that a device ejects a ball, you post an event rather than calling the device)
- Changed the balldevice_name_eject_request to be the event you use to call it, rather than the notification of the eject attempt.
- Created a get_status() method for ball devices
- Created a gather_balls() method and wrote the code that will send all the balls home before a game can be started.
- Updated stage_ball() code so it didn't ask for another ball if there was already an eject in progress
- Moved detection of how balls fall back in out of devices and into the events that watch for the entrance
- Create player and event based ejects. (This is a system to allow players or events to eject balls from ball devices. Useful for cannons like in STTNG.)
- Got stealth and auto eject out of the ball device code since they shouldn't care about that.
- Rewrote a lot of the ball device stuff.
- Added a manual eject capability for devices without eject coils
- Moved around some things between the ball controller and ball devices so that everything lives where it 'makes sense'
- Added method to check whether an event has any handlers registered for it.
- Ball devices now post events based on tags when balls enter them
- Ball devices can now eject their ball if no event is registered. This will prevent balls from getting "stuck" in unconfigured devices and will make prototyping on new machines faster.
- Changed event logging to show "friendly" names of handlers
- Converted flippers to use a config dictionary instead of variables
- Cleaned up the eject confirmation and valid playfield functionality
- Added a remove_switch_handler method to the switch controller

0.4

Released: July 25, 2014

- MAJOR rewrite of how the hardware platform modules interact with the framework's hardware module and how hardware is configured in general. It's way simpler and cleaner now. :)
- Created a parent class for Devices
- Cleaned up the way hardware objects use their parent class
- Fixed the ball controller so it doesn't get confused on the initial count after machine start up.
- Cleaned up switch processing and added a logical parameter so we only have to do all the conversion for NC or NO in one place

- Renamed the none interface to virtual. Rewrote it with the new platform interface way of working.
- Added support for holdPatter in coils
- Change add_live() to use tags instead of the plunger device
- Made it so many things, like ball search, autofires, etc. would not crash the machine if they weren't there.

0.3

Released: July 16, 2014

- Changed the way config files are loaded by making Config a normal section of any config file instead of using a special initial configuration file that did nothing but point to additional files. Details here.
- Created a virtualhardware platform for virtual / software only testing that does not require P-ROC or FAST drivers.

0.2

Released: July 11, 2014

- Added docstring documentation
- Added /sphinx folder and got the sphinx html docs included
- Created the first version of the documentation

0.1

Released: June 27, 2014

- Command line parameters to select real or fake (simulated) controller hardware.
- Command line parameters to select logging level
- Command line parameters to select the location of the initial config file
- Reads an initial config file which is a list of additional config files
- Processes those config files in order to build a config dictionary
- All platform-specific hardware code is isolated into its own module. Config files specify which platform is used. All game code is 100%interchangeable between platforms.
- Game loop runs with configurable loop rate. System timer tick event is raised every tick.
- Periodic and one-time use timers can be setup
- Switches, Coils, Lamps, and LEDs are read in and configured from the config files
- Switch events are read from the hardware
- Driver commands can be sent to the hardware
- Autofire drivers are automatically configured from the config files. They can be enabled, disabled, and reconfigured as needed.

- Flippers are automatically configured based on config files. They can use EOS or not, and be based on two coils (main/hold) or one coil with pulse+pwm. Multiple coils can be connected to the same switch, and vice-versa.
- The computer keyboard can be used to simulate switch presses. Key map configuration information is stored in the config dictionary. It supports momentary, toggle (push on / push off), and inverted (key press = open) key modes. Also supports combo key mapping (Shift, Ctrl, etc.)
- A switch controller receives all notifications of debounced hardware switch events.
- Can specify timed switch modes that trigger certain methods. (i.e. do blah() when switch_1 is active for 500ms.)
- Event manager handles system events, including registering handlers, priorities, aborting events, and maintaining a queue.

MPF Road Map, Vision & Future

To set the stage for our vision for the future of MPF, we'd like to start by saying that we love "traditional" pinball where you hit knock a physical ball into real targets.

While there's lots of talk about alternate concepts like Pinball 2000 and the Multimorphic P^3 (which replaces the bottom 2/3rds of the playfield with an LCD), our vision is focused on traditional-style pinball machines.

That said, we believe there is quite a bit of room for innovation even within the boundaries of classic pinball. For example:

Internet-connected pinball machines that report their own outages & problems

One of the problems with pinball on location today is that the machines often break. Unfortunately since most of these machines are owned by route operators, if a pinball machine in a bar breaks then the bartender just turns it off and the route operator has no idea that it's not earning. So if the operator is stopping by once a week to check on a machine, it might break an hour after he leaves and then be dark (and not earning) for the next 6 1/2 days until he comes back again.

We believe that pinball machines should be able to use the internet to report their current status. The operator should be able to log into a web portal to see all his machines and to view the current status. He should get text messages or iOS alerts with details of the "credit dot."

Furthermore, the ultimate indicator of whether a machine is working or not is whether it's earning. If a pinball machine only earns $20 a week, it's literally not worth an operator's time to drive to the location to check on it. So if he can see a report that the machine is earning as expected, he wouldn't have to waste his time and gas driving around to all his locations to check on his machines.

We can also be proactive when machines are turned off. The operator ought to be able to configure a schedule which basically says, "This machine should be powered on from noon until 2am every day," so if the cloud service ever loses connectivity with a machine during those hours, it can notify the operator (and maybe the location owner) that the machine is offline when it should be on, and the operator can make a phone call to see if the machine is ok before heading out. (And, if the machine is not ok, the operator can know that he's going out to the location for a reason.)

Of course their are plenty of times when a machine is powered on with no credit dot, but where the machine might still not be playable. (Maybe there's a stuck ball or a broken rubber.) In those cases we can go back to the earnings reports. If a machine is typically earning 5 dollars per day but half a day goes by without any money inserted, the machine can alert the operator that there's a problem.

Dynamic Pricing

Another cool thing about an internet-connected pinball machine is that operator settings can be centrally "pushed" to the machine. If a bar is rented out for a private party, the bar tender ought to be able to fire up an app on his or her smart phone to instantly set all the machines to free play. Or maybe there's an automatic schedule. "Wednesday night is free pinball," or "All pinball is free from 4-7pm." The operator ought to be able to set up a schedule and the machines should be able to change their pricing automatically based on the time of day.

We could even imagine "demand pricing," where the price is automatically adjusted up or down based on demand for a particular machine.

Player "Log in" for notification of high scores being beat

We love the idea of players being able to "log in" to a machine, most likely by "tapping in" to the machine with their Bluetooth or NFC-enabled smart phone. (This idea is not new of course. Pyprocgame creator Adam Preble blogged about this in 2014, and Dutch Pinball's Bride of Pin*Bot 2.0 and Big Lebowski have "Player Profiles" features.)

Regardless of how it's implemented, we love the idea of a particular player being able to login to a machine, since there are several cool things this could enable, including:

- Notification of high scores being beat. How cool would it be if you could get a text message or iOS notification when you lost your high score spot on your favorite machine?
- Accomplishments tracking. I would love to know what my high score was on different machines, or for a mobile app to tell me, "That's the most combos you've ever completed in Attack from Mars."
- Player preference settings. Most pinball machine settings are geared towards operators (number of balls per game, difficulty, etc.), but modern machines have plenty of options that don't matter to operators that hard core players are very passionate about. A pinball machine's app should allow players to set their own white balance for RGB LEDs (cool versus warm white), or the overall brightness of the LEDs, or even whether the LEDs "pop" on-and-off instantly or gently fade up and down like traditional incandescent bulbs. Players should be able set these preferences on their own or save their to their profile which they can have applied to whatever machine they walk up to.

All of this could be done on a per-player basis, with the machine taking on a different look and feel as each player steps up. Players could even set their color preferences with RGB LEDs in the apron lighting to indicate which player is up.

Mobile phone companion apps

We've already demonstrated a feature of the Mission Pinball Framework where we use an iPhone app as a "second screen" for a pinball machine. We can imagine players being able to customize their iOS app to show whatever data they want—score, ball, shots lit, etc.—which they can then set on the glass near the flippers. The machine could also send all DMD information and animations to that device and the player wouldn't have to take their eyes off the flipper area.

The mobile app could have a “helper” mode where it knows exactly what’s going on in the game and can tell you want to shoot for—kind of like if you had a world-class player standing over your shoulder and telling you want to do.

The mobile app could also let you know when it’s your turn (in case you walked away from the machine), or when a certain machine you’re waiting in line for is free. (Maybe you even pay for and “reserve” your place in line from your phone?)

It could also let you see all sorts of statistics for your game when while another player is playing (balls locks, goals remaining, etc.).

You’d also be able to collect very detailed metrics and analytics about your games. (Average time to hit a hurry-up, average ball time, number of shots, etc.) That could also be shared in a web-based dashboard and player ranking system.

Mobile phone audio integration

One of the things that stinks about playing pinball in a loud bar is that you can’t hear the machines. Some machines have headphone jacks, but that’s a separate piece of hardware.

What if you could pair your phone to the machine, and then the machine could stream its audio to your phone which you could listen to via headphones? You could even allow multiple people standing around to connect their audio to the same machine?

Another option is if you pair your phone with a machine, you could play a playlist from your phone instead of the machine’s music. The pinball machine could still add the voice call outs and sound effects, but just with your music. (This could be done via headphones or even through the pinball machine’s speakers.)

The machine could even have a mobile app which lists all the various music cues (waiting to plunge, base mode background, wizard mode background, etc.) and you could map those to individual tracks from your phone. Then whenever you walk up to a machine, you get your own custom music! (This could integrate with a cloud-based music service like Spotify or Apple Music and be configurable via the web so you get your own music any time you play that machine.)

Mobile phone “waiting player” actions

Traditional multi-player pinball machines alternate between players, with the non-playing players just watching the current player that’s up. The games themselves are very much about the “player versus the machine” more so than the “player versus player.”

But what if the waiting player could use their phone to mess with the current player who’s up? Maybe they have buttons that could temporarily shut off the flippers, or pop up drop targets which block shots, or release extra balls into play, or turn off all the lights. . .

These could be things that are granted to each player (you get one of each per game), or they could be earned by players for accomplishing certain achievements during the game.

Social media integration

Like it or not, people love posting random stupid things to social media, and their latest accomplishments on some pinball machine in a bar fit nicely into that. We can imagine a pinball machine tweeting high scores and jackpots made, perhaps even with a tiny camera in the top of the backbox which sends photos winning (and losing) moments to the players.

Most locations that have pinball machines also have social media accounts, and they struggle with ways to get their customers to "connect" with them. An internet-connected pinball machine could be part of that. Maybe they give players a free game (which they can redeem by tapping in with their phone) if the player lets the pinball machine tweet a photo of them winning.

"Offline" goals

An internet and social media connected pinball machine can also keep the relationship with the player going even when they're not at the machine. Maybe a player has to play a Facebook game or engage with a brand to "unlock" certain features of the game. Or maybe that's reversed, where people who play massive online games have to seek out a real world pinball machine to unlock certain goals in their online game.

Promos & advertising

We briefly mentioned the concept that locations could change their machines' pricing around special events and for happy hours. But why stop there? What if an advertiser, desperate to reach the 18-to-35 year old male, could buy their potential customers a free round of pinball? Imagine that tied to location services with the pinball players' app. You walk by a bar and your phone buzzes and it says "Lexus would like to buy you a free pinball game if you walk into this bar in the next 10 minutes." (Of course this is something that the bar could do too. Come in now and get a free game of pinball with every pint you buy.)

We could also imagine in-game advertising, maybe between balls or even integrated within the game. (Maybe a game has multiple pricing tiers, with the 25-cent game add supported while the 75-cent game remains "pure.")

Pinball only costs 75 cents or a dollar to play, and there are many types of advertising today where the advertisers pay far more than a dollar per impression. A pinball ad network could charge the advertiser one dollar per game, and the location and operator would make the same money they always did, the ad network could take their cut, and there would still be enough left over to increase the revenue a pinball machine could generate overall.

In-app purchases for game credits and power-ups

Even in 2014, we notice a lot of our friends saying, "I don't have any quarters," as an excuse not to play pinball. What if you could buy credits via an in-app purchase? There could be options for credits that expire, credits that are only good for one machine or one bar, bulk pricing discounts, and even credits that never expire. You could even structure it like a public transit card where a player's credits are automatically topped up when the balance gets low.

This could be used for much more than just credits. Players could buy options like extra balls, longer ball saves, tilt forgiveness, and other in-game goals all from their phones. The machines could keep track of which games used which options (important for keeping fair high scores), and the additional revenue could be shared with the location and operators.

Buh-bye four-button service menus!

It probably goes without saying that the four-button tap-tap-tap-tap-tap-tap-enter-tap-tap-tap service menu is going to be history. Every pinball machine moving forward should have a mobile app for operators that lets them configure settings and few reports and audits in an easy-to-use interface on the mobile device.

Even if they're not sitting at their machine, operators should be able to connect to a website to see all their machines, view Google Analytics-style earnings reports, remotely update software, push out configuration settings, and manage all aspects of the machine. Leaning down behind a coin door to configure things is almost laughable for a new machine in today's world!

Advanced tournament options

One of the problems with tournaments today is that if a machine malfunctions, it can break the current game in progress which isn't really fair to the current players.

What if the machine could maintain a sort of "transaction log" of everything that happened, so if a machine malfunctions, the tournament operator could hit a button to pause the machine, reset the ball or fix the problem, roll back the errant entries, and resume the game?

You'd also be able to integrate the actual machine scores and players with the tournament system. Super Selfie Leagues could automatically post scores and notify players when their scores have been beat or when they move down on the leaderboard.

Accelerometer integration

Modern machines with accelerometers can use them to track g-forces as well as to know the precise angle (in 3 axes) of the machine.

This means that the machine could notify the operator if the machine was not level. And when you were leveling the machine, it should show you that level on the display, or even read it out with text-to-speech as you were underneath the machine adjusting the legs.

The machine could also record the playfield angle for high scores (especially those posted online, maybe along with tilt sensitivity and outlane settings) to start to get a more universal baseline to high scores. (Though it still wouldn't be perfect due to wear, playfield wax, etc.)

The machine would also know if someone was lifting up the front of the machine (even slightly), which could make for some funny callouts. Maybe the points start draining until the player sets the machine down again.

You could even have a machine that can apply scoring multipliers based on the angle. (And maybe even have a machine where you can set the angle and scoring on your own?) Imagine "My high score on Ghostbusters is 200M at 6.5 degrees, but only 25M at 7 degrees."

More ideas from Jon Norris

Since we first wrote down our vision, someone let us know that pinball designer Jon Norris wrote about a bunch of ideas for innovation in classic pinball too. You can see his ideas at norrispinball.com. (Some are in the blog and some are in the "Re-Inventing" section of his site.)

Lots of cool stuff there too!

The future is bright!

One of the things we love most about pinball is that it's a real, physical thing. Traditional arcade games have lost much of their earnings power because everyone has a PS4 and 60" tv at home. But most people don't have pinball machines at home. And even though there are pinball apps for every device out there (which we LOVE, by the way), it just doesn't compare to actually banging a metal ball around with some mechanical levers.

Maybe it goes without saying, but we consider everything on this page to be our "to do" list for the Mission Pinball Framework.

The best part is that the Mission Pinball Framework is highly modular, so if you think some (or all) of these ideas are stupid, that's fine with us! You can pick-and-choose the parts of MPF that you like and throw out the rest.

Finally, we understand that a lot (ok, everything) we talked about here only applies to new pinball machines moving forward. But what about the hundreds of thousands of existing machines which are already in the world based on 20-year old technology? We have some ideas for them too. . . stay tuned!

Happy pinballing!

Late 2016 Update

We originally wrote this vision when we started MPF back in 2014 (though it's been updated since then). In late 2016, Jersey Jack Pinball announced Dialed In!, a machine that has some of the features we wrote about in our vision. At Expo, someone asked us if we were upset that Jersey Jack "ripped us off". Our answer is quite the opposite. We're thrilled! We love these ideas and love that they're making their way into pinball. (And frankly we hope that Stern and everyone else does these too.)

Everything about Mission Pinball is open and available for sharing, use, and ripping off. Take our ideas. Take our code. Copy our docs. We love it all!

MPF release checklist

What to do to make a MPF release?

- Update *MPF Release Notes* (mpf-docs repository dev branch)
- Create draft blog post in `missionpinball-website` repository (in `_draft` folder)
- Create `a.bb.x` branch (e.g. 0.50.x) and push it based on dev
  - `mpf` repository
  - `mpf-mc` repository
  - `mpf-monitor` repository
  - `mpf-examples` repository
  - `mpf-debian-installer` repository
- Create a.bb branch (e.g. 0.50) and push it based on `latest` branch in `mpf-docs` repository
- Add `a.bb` to versions on readthedocs and wait until it finished building
- Remove `a.bb` from redirects in readthedocs

- Add `a.bb + 1` to redirects in readthedocs
- Increase version to a.bb.0 on `a.bb.x` branch
  - `mpf` repository
  - `mpf-mc` repository
  - `mpf-monitor` repository
- Set version to `a.bb.x` in `mpf-mc` repository in `appveyor.yml`
- Wait until all builds pass
- Increase version to `a.bb.0-dev0` (`bb + 1` or `a + 1`) on dev branch
  - `mpf` repository
  - `mpf-mc` repository
  - `mpf-monitor` repository
- Update `latest` branch on `mpf-docs`
  - Remove branch protection
  - Set `current_branch` to `a.bb.x` in `conf.py`
  - Set branch in `.travis.yml` to `a.bb.x`
  - Remove `--pre` from install notes
  - Update linux installer references to `a.bb.x` instead of dev
  - Push dev branch to latest (hard push)
  - Re-add branch protection
- Update `dev` branch on `mpf-docs`
  - Update version to next release in `conf.py`
- Protect branches
  - `a.bb.x` on `mpf` repository
  - `a.bb.x` on `mpf-mc` repository
  - `a.bb.x` on `mpf-monitor` repository
  - `a.bb.x` on `mpf-examples` repository
  - `a.bb` on `mpf-docs` repository
  - `a.bb.x` on `mpf-debian-installer` repository
- Publish release post on forum
- Increase version in forum header
- Publish release post on pinside
- Publish release post on slack
- Delete pre releases on pypi
  - `mpf`
  - `mpf-mc`

    - `mpf-monitor`
- Tag Release on github
    - `mpf`
    - `mpf-mc`

Index

D

M

P

W