



MISSION PINBALL FRAMEWORK DOCUMENTATION

MPF Version this documentation applies to: 0.21.3
Document Creation Date: March 27, 2016

This document was auto-generated from the online documentation at missionpinball.com/docs.

Created by Brian Madden and Gabe Knuth

Contents

[MPF Overview](#)

[Understanding the MPF Package & Versions](#)

[File & Folder Structure](#)

[Machine Configuration Files](#)

[Running your game](#)

[Current Features](#)

[Working with physical pinball machines](#)

[Choosing a pinball machine](#)

[Installing MPF](#)

[Windows](#)

[Linux \(Debian / Ubuntu\)](#)

[Mac OS X](#)

[Tutorial](#)

[Step 1. Prerequisites](#)

[Step 2. Download & Install MPF](#)

[Step 3. Create your machine folder](#)

[Step 4. Get flipping!](#)

[Step 5. Adjust your flipper power](#)

[Step 6. Add a display](#)

[Step 7. Add keyboard control](#)

[Step 8. Create your trough](#)

[Step 9. Create your plunger lane](#)

[Step 10. Add a start button](#)

[Step 11. Run a real game!](#)

[Step 12. Add the rest of your coils & switches](#)

[Step 13. Add the rest of your ball devices](#)

[Step 14: Add slingshots & pop bumpers](#)

[Step 15: Add your first game mode](#)

[Step 16: Add scoring to your base mode](#)

[Step 17. Create an attract mode DMD slide show](#)

[Step 18: Add lights or LEDs](#)

[Step 19: Create an attract mode light show](#)

[Step 20: Create your first shot](#)

[How To Guides](#)

[How To: Check your version of MPF](#)

[How To: Migrate from MPF 0.21 to MPF 0.30](#)

[How To: Use a P-ROC with MPF](#)

[How To: Use a P3-ROC with MPF](#)

[How To: Use a FAST Pinball controller with MPF](#)

[How To: Use a System 11 machine with MPF](#)

[How To: Configure RGB LED SmartMatrix displays](#)

[How To: Use an event to disable a switch](#)

[How To: High Scores](#)
[How To: Add Coins & Credits](#)
[How To: Show scores in the attract DMD mode](#)
[How To: Create a Bonus mode](#)
[How To: Use FadeCandy for LEDs](#)
[How To: Create a Tilt](#)
[How To: Lane Change / Lit Shot Rotation](#)
[How To: Create a Timed Skill Shot](#)
[How To: Create a Kickback Mode](#)
[How To: Configure a 1980s-style trough](#)
[How To: Create an Attract Mode DMD slide show](#)
[How To: Create a "super jets" countdown mode](#)
[How To: Add TrueType Fonts](#)
[How To: Configure a Color DMD](#)
[How To: What if your plunger lane has no switch?](#)

[MPF Architecture Concepts](#)

[The MPF Core Engine](#)

[Platform Interfaces](#)

[Smart Virtual Platform](#)

[FAST Pinball](#)

[Multimorphic P-ROC / P3-ROC](#)

[Virtual Platform](#)

[Snux](#)

[FadeCandy](#)

[Open Pixel](#)

[SmartMatrix](#)

[System Modules](#)

[Asset Manager](#)

[Ball Controller](#)

[BCP Interface](#)

[Config Manager](#)

[Delay Manager](#)

[Device Manager](#)

[Data Manager](#)

[Event Manager](#)

[File Manager](#)

[Hardware Show Controller](#)

[Logic Blocks Manager](#)

[Machine Controller](#)

[Mode Controller](#)

[Score Controller](#)

[Shot Profile Manager](#)

[Switch Controller](#)

[System Timing & Timers](#)

- [Tasks](#)
- [Devices](#)
 - [Low-Level Devices](#)
 - [Coil \(Solenoid\)](#)
 - [Driver-Enabled Device](#)
 - [Flasher](#)
 - [GI String](#)
 - [LED](#)
 - [Matrix Light](#)
 - [Switch](#)
 - [Logical Devices](#)
 - [Ball Device](#)
 - [Diverter](#)
 - [Drop Target](#)
 - [Autofire Coil](#)
 - [Flipper](#)
 - [Playfield](#)
 - [Playfield Transfer](#)
 - [Score Reel Group](#)
 - [Shot & Shot Group](#)
 - [Score Reel Controller](#)
 - [Score Reel](#)
 - [Abstract Devices](#)
 - [Ball Lock](#)
 - [Ball Save](#)
 - [Multiball](#)
- [Modes](#)
 - [Attract Mode](#)
 - [Game Mode](#)
 - [Credits Mode](#)
 - [High Score Mode](#)
 - [Tilt Mode](#)
- [Events](#)
- [Player Management](#)
- [Machine Variables](#)
- [Media Controllers](#)
 - [The MPF Media Controller](#)
 - [What's new in the MPF Media Controller \(mpf-mc\) in v0.30?](#)
 - [Unity Media Controller Architecture](#)
 - [Backbox Control Protocol \(BCP\) 1.0 Specification \(DRAFT\)](#)
- [Ball Tracking](#)
- [Lighting & Display Shows](#)
 - [Hardware Shows](#)
 - [Show Settings](#)
 - [Light Show file format](#)

- [Other important light concepts](#)
- [Playing & Stopping Shows](#)
- [Light Scripts](#)
- [Creating Playlists](#)
- [Displays & DMDs](#)
 - [Displays](#)
 - [DMD](#)
 - [Window Manager](#)
 - [Segmented Displays](#)
 - [Slides](#)
 - [Display Elements](#)
 - [Positioning & Placement](#)
 - [Text](#)
 - [Image](#)
 - [Animation](#)
 - [Movie](#)
 - [Shape](#)
 - [Virtual DMD](#)
 - [Character Picker](#)
 - [Entered Chars](#)
 - [Transitions](#)
 - [Move In](#)
 - [Move Out](#)
 - [Fonts](#)
 - [Decorators](#)
 - [Blink](#)
- [Sounds & Audio](#)
- [Machine HZ & Loop Rates](#)
- [Keyboard Interface](#)
- [Multi-language Support](#)
- [Plugins](#)
 - [Auditor](#)
 - [Info Lights](#)
 - [OSC Module](#)
 - [Configuring OSC Clients](#)
 - [Switch Player \(Test Automation\)](#)
 - [Creating your own Plugins](#)
- [System Flows](#)
 - [Initial Start Up & Boot Sequence](#)
 - [Ball Start Sequence](#)
 - [Ball End Sequence](#)
 - [Game Start Sequence](#)
 - [Mode Start Sequence](#)
 - [Mode Stop Sequence](#)

[Game Programming](#)

[Game Logic & Rules](#)

[Game Modes](#)

[Managing Assets](#)

[Advanced Programming](#)

[Scriptlets](#)

[Mode code](#)

[MPF Coding Best Practices](#)

[Tools](#)

[Config File Migrator](#)

[Font Tester](#)

[OSC Sender](#)

[Configuration File Reference](#)

[Important config file concepts](#)

[Case-sensitivity in config files](#)

[How to add lists to config files](#)

[How to configure "device control events"](#)

[How to format time values in config files](#)

[#config_version](#)

[#config_version=4](#)

[#config_version=3 changes](#)

[#config_version=2 changes](#)

[accruals](#)

[animations](#)

[asset_defaults](#)

[assets](#)

[auditor](#)

[autofire_coils](#)

[ball_devices](#)

[ball_locks](#)

[ball_saves](#)

[bcp](#)

[coils](#)

[config](#)

[counters](#)

[credits](#)

[dmd](#)

[diverters](#)

[driver_enabled](#)

[drop_targets](#)

[drop_target_banks](#)

[event_player](#)

[fast](#)

[fonts](#)
[flashers](#)
[flippers](#)
[game](#)
[gis](#)
[hardware](#)
[high_score](#)
[images](#)
[info_lights](#)
[keyboard](#)
[languages](#)
[language_strings](#)
[light_player](#)
[light_scripts](#)
[logic_blocks](#)
[leds](#)
[led_settings](#)
[machine](#)
[matrix_lights](#)
[media_controller](#)
[mode](#)
[modes](#)
[multiballs](#)
[mpf](#)
[open_pixel_control](#)
[osc](#)
[playfields](#)
[random_event_player](#)
[score_reels](#)
[score_reel_groups](#)
[scoring](#)
[scriptlets](#)
[sequences](#)
[shots](#)
[shot_groups](#)
[shot_profiles](#)
[show_player](#)
[shows](#)
[slide_player](#)
[smartmatrix](#)
[snux](#)
[sound_player](#)
[sounds](#)
[sound_system](#)
[switches](#)

[switch_player](#)
[system1](#)
[tilt](#)
[timers](#)
[timing](#)
[triggers](#)
[videos](#)
[virtual_platform_start_active_switches](#)
[volume](#)
[window](#)

[List of Machines MPF has been used with](#)

[Road Map & Current Status](#)

[Changes in MPF v0.30](#)

[Version History](#)

[MPF 0.30 Release Notes](#)

1. The Mission Pinball Framework Project

The Mission Pinball Framework ("MPF") is a Python-based pinball software framework that's used to run real pinball machines. Our goal is to enable both casual builders and hard-core programmers to create the software to run their pinball machines—whether it's new game code for an existing pinball machine, a "re-theme" of an old machine, or totally custom / homebrew machine built from scratch. 95%+ of what you need to do to program a pinball machine in MPF is done with text-based configuration files, so even if you have no prior programming experience, you can still do it! (We also have a well documented API if you're a developer who prefers to write against that. Use as much or as little of the config files as you want. It's all good.)

MPF is completely free and open source, released under the [MIT License](#). We starting writing pinball code in 2013 and officially began the MPF project in June 2014. At this point the framework is incomplete, but we're making fast progress and we've already demonstrated a complete game running on it. (Check out MPF's current [feature list](#) here to see what's done and what's left to do.)

The MPF project was originally started by *Brian Madden* and *Gabe Knuth*, though the MPF project team has grown as more awesome people have contributed their time and coding skills. (Check out the *AUTHORS* file in the download package for the latest complete list.) Everyone working on MPF is doing it as a labor of love, and there are typically 40-60 hours per week of total coding, testing, and documentation writing. At this point MPF represents over 5,000 hours of work!

Think about that. If you've ever dreamed of building your own pinball machine, MPF can give you a 5,000 hour head start on the software. :)

Project Goals

We have lofty (and ever evolving goals), including:

- Be easy to use, so people with no prior programming experience can create a pinball machine.
- Be powerful and flexible, so experienced programmers can easily understand its structure and shape it to their needs.
- Support all "eras" of pinball machine hardware, from EM to early solid state with segmented displays to DMD to modern LCD-based and color DMD games.

- Support converting existing machines to "new" code and brand-new games built from scratch.
- Support as many different pinball controller hardware platforms as we can. (P-ROC, P3-ROC, FAST Pinball, Open Pinball Hardware, etc.) This should be done in a hardware-independent way, so you can literally switch between hardware controllers without having to make changes to your configs or code.
- Support all the "traditional" pinball components, including ball and game handling, ball search, ball save, tilt, drop targets, lane change, pop bumpers, credit and coin counting, audits, service menus, game modes, multiball, high scores, etc.
- Support the "new" pinball concepts, like player profiles, web-based reporting and control for operators, social media integration, iPhone integration, etc.
- Most device, configuration, game logic, and modes should be done via simple-to-understand text-based configuration files.
- Provide a solid API for experienced programmers who prefer to build games that way instead of using MPF's text-based configuration files.
- Games should be easily customizable and should not look "stock" or like they're from the same template. (Fonts, text, images, animations, game flow, logic, display layout, modes, etc.) In other words, no one should be able to tell that a game is powered by MPF. :)
- Be well-documented, both in terms of step-by-step "how to" guides and tutorials for beginners, as well as more advanced references for hard-core programmers.
- Be extremely modular, allowing game programmers to subclass, extend, plug-in, and replace certain built-in functionality without breaking the everything or relying on "hacks".
- Be robust enough to power commercial games, including games that are in revenue service in public locations.
- Be 100% free with a liberal license that lets you do whatever you want, including selling games based on MPF without having to pay a license fee or give away your game's source code.

This website is very much a work in progress. (As is the MPF software.) You follow latest updates via our [blog](#) or [@MissionPinball](#) on Twitter.

Compatible Pinball Machines and Hardware

The MPF project is software only. To use MPF with a physical pinball machine, you need a hardware pinball controller which acts as the "link" between the computer running MPF and the pinball machine's driver and switch boards. (More details on how that works [here](#).)

MPF currently supports several different modern pinball controllers, including:

- [Multimorphic](#): P-ROC and P3-ROC controllers
- [FAST Pinball](#): Core, WPC, and Nano controllers
- [Open Pinball Project](#): Gen2 open source hardware hardware

MPF's architecture is hardware-independent, meaning we can write interfaces for any pinball controller hardware out there.

Some of these hardware pinball controllers are replacements for the CPU boards in existing machines, meaning you can use MPF to write your own software to control existing machines. Williams / Bally / Midway WPC machines, as well as Stern Whitestar and S.A.M. machines are the easiest. Williams / Bally System 11 and Data East are pretty easy too. (You have to replace the driver board in those too.)

That said, MPF can control any machine if you're up for building some custom wiring harnesses to interface a modern pinball controller to the existing machine. (We even [rewired a 1974 EM machine](#) which runs MPF great!)

You can also use these modern controllers with modern driver boards to power your own completely custom-built brand new machines.

Alternatives to MPF

When we first started searching for people writing their own software for pinball machines in 2013, we found a lot of other projects. So if you're just getting into this, we want to make sure you know about all the options out there. (Many of these are free & open source.) Our ultimate goal is to see more pinball in the world—regardless of whether it's done with MPF!

So instead of the Mission Pinball Framework (which is what this site is about), you could also use one of the following other software frameworks:

- [PyProcGame](#) (Python framework for the Multimorphic P-ROC and P3-ROC pinball controllers, from Gerry Stellenberg and Adam Preble.)
- [PyProcGameHD](#) (Fork of Pyprogame which uses an LCD-based HD full color DMD instead of the traditional DMD, from Michael Ocean and Josh Kugler. Includes a "Skeleton Game" to get you started quickly. Works with P-ROC and P3-ROC hardware only.)
- [NetProcGame](#) (.NET version of PyProcGame for the P-ROC and P3-ROC, written by Jimmy Lipham)
- [Rampant Slug Pinball Framework](#) (Very early framework written in .NET. Has an awesome GUI. Works with P-ROC and P3-ROC hardware only.)
- [FreeWPC](#) (Write code in C and compile & burn it to ROMs which you can use in existing WPC machines, from Brian Dominy)

- [Open Pinball Project](#) (Open source pinball framework for controlling OPP hardware.)
- [PinKit](#) (Software & hardware combination from Kerry Imming)
- [milliSoft PINterface](#) (Software & hardware combination from German company milliSot. Here's an [English manual](#) if you want to see what they're about.)

Please [contact us](#) if we've missed any!

There's also a great wiki at [PinballMakers.com](#) with lots of information for people who want to create their own pinball machines.

Documentation

Pinball machines are complex, as is the software that runs them. We're trying to make everything as easy as possible, and so far we have over 800 pages of documentation explaining everything step-by-step. (Concepts, how to guides, tutorials, programming references, etc.) The user documentation lives at [missionpinball.com/docs](#), and the API reference is available at [missionpinball.com/apidocs](#). You can explore the documentation via the tree view menu in the left-hand sidebar of this page, or download a PDF bundle of all the docs from our [downloads](#) page.

Step-by-Step Tutorial

We have a [getting started tutorial](#) which walks you through everything, from downloading MPF to getting a full game playing (with scoring, DMD, multiple players, game modes, etc.). You can literally go from never having heard of MPF to a working game on a physical pinball machine in a few days. (And you can go from "zero to flipping" in a few hours.)

How-To guides

Once you complete the tutorial, you can browse our very detailed and specific [How To guides](#) which show you how to add the features you want. We have several guides available today, with dozens more in the works. (If you want to learn how to do something, post a question in our forum and we'll write a how to guide for it!)

Installing MPF

We have an [installation guide](#) that walks you through installing MPF, either with our automated all-in-one installers (for Windows, Mac, and Linux) or by doing it manually. The all-in-one installers should get you from zero to running MPF in under three minutes, and even manually installing everything should only take about 10 minutes. It's all really easy.

MPF Projects & Source Code

The Mission Pinball Framework project has a few different components which are available as separate GitHub repos:

The Mission Pinball Framework

This MPF core engine is in the [mpf](#) repo. The MPF core is responsible for:



- Interfacing with the physical pinball hardware
- Managing and controlling all devices, including drivers, lights, LEDs, switches, ball devices, diverters, drop targets, servos, etc.
- Managing and tracking all shots, shot groups, status, progress, etc.
- Game, ball, player, and machine management
- All game logic
- Starting and stopping modes
- Shots, scoring, times, countdowns, bonus calculations, audits, etc.
- Running and coordinating all effects shows
- Interfacing with files, including the game config files, audits, game data, high scores, etc.
- Running the main game loop, clock, and events system

The MPF Media Controller

The MPF media controller is in the [mpf-mc](#) repo. It's a standalone process (so it works nice on multi-core processors) which is responsible for:



- On screen display windows
- DMD / Color RGB DMD displays
- Images, animations, video, text, and all displays widgets
- Sounds & audio
- Multi-language translations (including alternate asset packages)
- All game UI
- Loading and unloading game assets (on demand, if needed)

The MPF Monitor



The MPF Monitor is an upcoming tool in the planning stages (it doesn't exist yet) that you will be able to use to connect to a live running instance of MPF to control, set, and view live information from a running instance of MPF. It will let you do things like seeing the current states of lights & LEDs, coils, and switches, as well as let you activate switches via your computer just by clicking on them.

The MPF Monitor will be useful when you're developing your MPF machine when you don't have your physical machine handy, and it will be great for troubleshooting, even letting you "pause" MPF to dig into its current state to find out exactly what's going on.

The MPF Monitor will let you view (and update) the states of devices, player and machine variables, modes, logic blocks, timers, events, hardware, and pretty much anything else that's part of MPF.

The MPF Wizard



The MPF Wizard is an upcoming project (in the [mpf-wizard](#) repo) which is graphical tool you can use to build and configure your MPF machine configurations. (Think "double click and change settings for a device" versus "find the device in a config file, look up the options, and re-save the file".)

Whereas the MPF Monitor will be used to view the state of a live running MPF instance, the MPF Wizard will be used to actually build your configs, slides, modes, and all your MPF game logic.

The Wizard is in the very early stages and doesn't really do anything yet (apart from being able to open your config files), but we're working on it!

MPF Examples



The MPF Examples (in the [mpf-examples](#) repo) contains code and config examples you can use as you're learning MPF and working on your own game.. There are demos and code that goes along with the How To guides and the tutorial, as well as hardware configurations for a bunch of real machines. None of these examples contained any copyrighted material, so you can use everything you find here in your own machines.

Support & Community & Fun People

We have an active [user forum](#) with over 3,000 posts where we talk about MPF use, development, ideas, and where people using MPF share photos and videos of their projects. People using MPF attend many of the major pinball conferences, and we're always happy to get together and talk pinball! (We have a [forum dedicated to upcoming events](#) where MPF users can meet in person.)

Who owns MPF? What's the license?

MPF is released under the ["MIT" license](#) (also called the "Expat license") which is extremely permissive. You can do just about anything you want with it. Use it. Copy it. Modify it. Merge, publish, distribute, sublicense and/or sell it. If you make changes, you can choose to share them back with the community. Or not. If you make a sell a commercial pinball machine based on MPF, you do not have to pay us anything. You also do not have to release your machine's source code if you don't want to.

We don't "own" MPF any more than you do. You can even make closed-source derivatives of MPF and sell them. You can do whatever you want with it.

So go nuts. Rip us off. Take the code. Make it yours. Don't give us credit... It's all good! Seriously. We're just putting the code out there. We just want to see more pinball in the world and we don't care what you do with MPF!

Next Steps & Getting Started with MPF

Feel free to browse the MPF documentation via the tree-view menu on the left side of this page, start reading our [MPF introduction](#), or jump right into our [step-by-step tutorial](#) to get started!

2. MPF Overview

Writing your own software for a pinball machine is a lot of work—regardless of whether you're writing brand-new code for your favorite '90s Williams machine or you're designing and building your own machine from scratch. In addition to configuring all your hardware devices, you have to figure out your shots, scoring, modes, and game features, and you also have to build light shows, DMD and/or LCD animations, sound effects, music, dialog callouts, and probably a million other little things.

Our goal with the Mission Pinball Framework (which we often refer to as *MPF*) is to create a pinball machine software platform you can use to run a real pinball machine. Our view is that many of the "standard" features of modern pinball machines are pretty similar from game-to-game, so we ought to be able to create a software platform that does 90% of the basics right out of the box—allowing you to get to the fun stuff as soon as possible.

We've built MPF so that it's hardware-independent. MPF currently supports [Multimorphic](#) (P-ROC and P3-ROC) and [FAST Pinball](#) (Core, WPC, and Nano) controllers, and if anyone else enters the pinball controller market, we'll try to support them as well!

Regardless of the hardware platform you choose, getting your game fully built on MPF is going to take some time. Downloading and installing the software is easy, but building and programming the actual configuration for your machine can be pretty involved. We're trying to make that as easy as possible, (and we have a [step-by-step tutorial](#) that can get you flipping in a few hours and running a full game with modes and scoring and display and sound effects in a few days), but with with all your game modes, rules, light shows, animations, and audio, it's definitely going to be a labor of love!

The Mission Pinball Framework is a work-in-progress. (And probably always will be!)

MPF is far from complete. We're not even to our 1.0 release yet so the code is very "alpha" at this point.

That said, we have a lot done and you can absolutely build a game with it today. The core system is done. You can read in switches, track balls, and fire coils. We have the game flow built, so the machine can track players and ball numbers and drains and turns. You can start and end a game, build light shows, and create game logic. You can create game modes. We have DMD, LCD, color DMD, and audio support. But there's still a long, long way to go.

We are sharing MPF now just so you can get a preview of it to see where we're going, and (if you want) so you can contribute ideas and code.

We don't have specific timeframes in mind for a 1.0 release. (We've been releasing new stable-ish versions every 6 weeks or so with dev versions almost daily.) That said, we have a "to do" list with probably [2 or 3 years' worth of ideas](#), and we're sure that list will continue to grow.

Brian Madden & Gabe Knuth, Mission Pinball

Understanding the MPF Package & Versions

Source code for the MPF project is hosted on [GitHub](#). If you look at the project page, you'll see several branches:

- The *master* branch is the current up-to-date version of MPF and the most stable. It's typically updated every month or two. Versions are in the format x.y.z, (such as 0.17.1). The .z number is the latest patch / bug fix number and changes there do not introduce any new functionality.
- The *dev* branch is the work-in-progress of the next version of MPF. It's typically updated a few times a week. Versions *ahead* of master and end in "-dev" plus a revision number. For example, if the latest master branch is 0.17.1, the latest dev branch might be 0.18.0-dev4.
- The *gh-pages* branch contains the [Sphinx-generated HTML API documentation](#) from the [docstrings](#) in the master branch. It doesn't contain any MPF code, and the other MPF packages do not include any documentation. (Keeping them separate keeps both cleaner.)
- There might also be other branches at various times which contain work-in-progress bug fixes, new features, etc. We work on them there in order to keep the master and dev branches somewhat stable.

You can always download the latest stable version of MPF via the following link: <https://github.com/missionpinball/mpf/releases/latest>

You can think of the MPF package as containing three parts:

- The MPF game engine.
- The MPF media controller.
- Machine-specific configuration files & code.

The [MPF game engine](#) contains all the files MPF uses to run, including the MPF system components, plugins, pinball device drivers, built-in modes, and a system-wide configuration file. These components are contained in the `/mpf` folder in the MPF package that you download.

The [MPF media controller](#) is responsible for all the display, graphics, video, and audio components of a pinball machine, including the DMD and LCD display windows. MPF's

architecture separates the game engine from the media controller, so you can replace the "in-box" MPF media controller with a more advanced one if you like. (For example, there's a team working on a Unity 3D-based media controller.) MPF's built-in media controller files live in the `/mpf/media_controller` folder in the MPF package.

The third part of MPF is your [machine files](#). These are the specific configurations, code, images, sounds, and videos that make up a single pinball machine. MPF includes some sample machine configurations in the `/machine_files` folder in the MPF package, with each specific machine having its own folder under there. (Most likely you'd put your own machine's files in a folder outside of the MPF folder so you can download updates of MPF without overwriting your own machine's files.)

File & Folder Structure

When you download the latest [Mission Pinball Framework release from GitHub](#) (click the *Source code* link in the "Downloads" section), you'll get a file called `mpf-<branch>.zip` or `mpf-<branch>.tar.gz`, (depending which link you click). When you unzip it you'll find a few folders and files:

- `/mpf` (the main game framework)
 - `/devices` (modules for different types of pinball hardware)
 - `/media_controller` (our built-in media controller which handles the DMD, LCD windows, and audio)
 - `/modes` (built-in modes for the game, attract, high score, credits, tilt, etc. You can extend, customize and/or replace these.)
 - `/platform` (the hardware interface modules, for P-ROC, FAST, or virtual hardware)
 - `/plugins` (default plugin modules that we include)
 - `/system` (MPF game engine modules)
 - `mpfconfig.yaml` (holds the default and system configuration settings)
- `/machine_files` (contains sample game configurations. You'll put your own game files in a different location outside of MPF.)
 - `/demo_man` (sample game config & code for our ["Demo Man"](#) game)
 - `/new_machine_template` (contains template files you can use to create your own game)
 - `/tutorial` (contains configuration files from each step in our [getting started tutorial](#))
- `/tools` (contains helpful tools and utilities)
- `mc.py` (the Python file you run to start the media controller which runs the DMD, LCD, and audio)

- *mpf.py* (the python file you run to start your game)

Additional folders, such as */logs* will be automatically created after you run your first game.

Machine Configuration Files

The real magic of the Mission Pinball Framework happens with configuration files. MPF uses a file format called "[YAML](#)" which is text-based and human readable. (YAML is kind of like XML, though easier to read and write. It's kind of like INI files, though more powerful.) We call these the *machine configuration files* (because they specify the configuration of your pinball machine), and if you use the MPF then you're going to get to know them *very well!*

In MPF, just about *everything* your game does can be configured via these config files. (Seriously. Almost everything: switches, coils, lights, light and sound shows, animations, scoring, game logic, modes, display effects, cut scenes...) We have a [detailed configuration 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.

When you create code for a pinball machine with MPF, you create a folder which will hold all the settings and files that machine needs. (This folder contains everything, and in fact will be portable. You could drop your machine's folder into our copy of MPF and it would work fine.)

Inside your machine's folder, you'll have a subfolder called `config`, and that folder will hold your configuration file (or files). So then when you run MPF, you specify which machine (i.e. which folder) you want to run, and then MPF will look for a configuration file called `<your machine>/config/config.yaml`. (You can actually use the command line to specify *which* config file you want to use for your machine, meaning you can easily test out different things or run different configurations for the same machine with the same folder in MPF.)

In addition to holding settings for your machine, a config file can actually point to an additional config file (or files) that should be read in too. This means you can break up your configuration across as many files as you want, and MPF parses all the settings from all the files—in the order they're listed—to create the actual configuration for your machine. (If you look at the *Judge Dredd* configuration files which we include with MPF as an example, you'll see that we broke its config down in a bunch of different files, like `machine.yaml`, `game.yaml`, `devices.yaml`, `text_strings.yaml`, etc.)

How many configuration files you have is totally up to you. You can break the settings into fifty little files if you want, or you can have one huge file that contains all your settings. It really doesn't matter at all. (Seriously, it doesn't matter. The game reads all the settings from all the files into a single configuration dictionary when the game code starts up, so once all the settings are read in it doesn't care whether they came from 100 files or one file.)

The reason we allow the machine config files to be broken into multiple files is because it makes it easier to share files between projects and machines. For example, at the most basic

level, the switches, coils, and lights in a *Funhouse* machine are the same regardless of whether you're writing new game rules from scratch or just building a new version of the classic game. (So for *Funhouse*, the solenoid which moves Rudy's eyes to the right is Driver #25, connected to J122 Pin 1, no matter what.) So in that case you might put all the hardware configuration settings in a file called `funhouse_hardware.yaml`, and then you might have `brian.yaml` which is Brian's version of the game rules, and `gabe.yaml` which is Gabe's version, etc.

What do you configure in these machine config files?

In a word: everything. The machine config files specify what hardware platform you use, how your coils, switches, and lights are mapped, what options they have, and what names these use, how your playfield devices are set up, ball search, ball save, coin and credit settings, text strings, high scores, score values, sound and music effects, DMD animations, game modes, game logic and game flow—the list goes on and on.

You also use these config files also specify what your operator settings are. You literally use them to control *which* options are exposed via the service menu (as well as default options and acceptable ranges of inputs), and they're used to read in stored options that affect how the game is played and how the machine behaves. When an operator navigates through the service menu to configure their game, those configuration settings are written to—you guess it—another YAML configuration file. (The YAML file holding the operator settings is read last, meaning any settings the operator changed overwrite whatever default settings you specified in earlier files.)

For example, you might have the following setting in a configuration file called `game.py`:

```
game:
  balls_per_game: 3
```

And then if the operator changes this to 5, an entry might be written in a file called `operator_settings.yaml` file like this:

```
game:
  balls_per_game: 5
```

If you look in the initial `config.yaml` which is first read in that specifies the names and order of additional files, you'll see that `operator_settings.yaml` is the last file loaded, meaning it will overwrite any default settings that were specified earlier. (This also means that you can play with different sets of settings or sets of operator settings by creating multiple initial config files which each specify different combinations of subsequent files. Then you can just specify different initial files via the command line when you run your game to try out different settings.)

One final note in case it's not clear. The actual names of the YAML configuration files are totally arbitrary. Any setting can be in any file, and which files are read are specified in the list in the initial file. We picked default file names and broke them into logical file groupings, but if you want to call your files `potato.yaml` and `hubcap.yaml`, that's fine with us.

Again, you can check out the [config file reference](#) to see all the settings and options for things you can configure via these files.

Running your game

(Note: This page is a reference for how to start MPF and all the various command-line options. If you're brand-new to MPF and running it for the first time, we recommend that you follow our [step-by-step getting started tutorial](#). The first few steps will walk you through downloading and installing MPF and getting our sample *Demo Man* machine up and running.)

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

There are actually two separate processes that you need to run for MPF: The core **MPF game engine** and the **separate media controller**. The game engine talks to the pinball hardware and runs the game logic, and the media controller drives the LCD/DMD and audio.

The game engine and media controller talk to each other via an open protocol called "[BCP](#)" (for "Backbox Control Protocol") via a TCP socket.

At this point you might be wondering why the game engine and media controller are two separate processes? Two reasons:

- 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 2 would have all the MPF stuff running on one core while the other three cores were wasted doing nothing.
- Having two processes means can replace the built-in Python-based 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.

To run an MPF game, you have to start both the media controller and the MPF game engine. You can do both of these via the command line. However, by default, each component "takes over" the terminal window while it's running, so you actually need to open two terminal windows and run the MPF core engine in one and the media controller in the other. If you don't want to do this we also have a batch file which you can use instead which automatically pops up the two separate windows and runs both pieces. (More on that in the next section.)

Anyway, to run MPF, first start the media controller, like this:

```
python mc.py <path to the machine files you want to run>
```

For example:

```
python mc.py demo_man
```

Then in a second terminal window, start the MPF game engine, like this:

```
python mpf.py <path to the machine you want to run>
```

For example:

```
python mpf.py demo_man
```

Command line parameters & options

There are several command line parameters and options you can add to both the MPF engine and the media controller. The order of these options is not important.

<your machine folder>: The only required command line parameter is to pass the folder location of your machine's folder. The idea is that you can have one copy of MPF which can be used to drive multiple machines, so you use this command line parameter to specify which machine you're running. This command line parameter does not have any dashes or letters that precede it.

The machine folder is the folder that contains your machine's *config*, *modes*, etc. folders.

You can pass a full path to your machine file. For example, if MPF is in the *C:\pinball\mpf* folder, and your machine folder is *c:\pinball\your_machine*, you would switch to the *c:\pinball\mpf* folder and run MPF like this:

```
python mpf.py c:\pinball\your_machine
```

Or if you're using the batch file to launch the MPF core engine and media controller at the same time:

```
mpf c:\pinball\your_machine
```

-c: Specifies the name (and optionally the path) of the initial configuration file MPF will load. If you don't specify this option, it will automatically look for *config/config.yaml* in your machine path folder. (So it looks for a file called *config.yaml* in a folder called *config*.)

Note that you can also specify a comma-separated list of multiple files, and MPF will load them in order, meaning that settings in the latter files overwrite settings in the earlier files. This is great if you want to test different configurations. For example, you might have your default *config.yaml* with all your settings, including the media controller to run in full screen mode, but you might also create a second config file called *small_window.yaml* which sets your display to run in a small window which is nice when you're troubleshooting. So then you could run your machine in full screen mode via:

```
mpf c:\pinball\your_machine
```

And then you could run the small windowed version via:

```
mpf c:\pinball\your_machine -c config,small_window
```

-x: (Uppercase "X") Forces MPF to use the smart virtual platform interface, regardless of what your *hardware: platform:* is set to. Note that the smart virtual platform is the default which is used if you do not specify a platform in your machine config file.

-x: (Lowercase "x") Forces MPF to use the virtual platform interface, regardless of what your *hardware: platform:* is set to.

-b: Sets MPF to not connect to the Media Controller via BCP. This is nice if you're just running the MPF core engine without a media controller so you don't get all those timeout messages about it not being able to connect to a BCP server.

-v: (Lowercase "v") 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 "V") 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.

-c: Sets the location of the default system-wide configuration file. This file has the same format (and can contain the same information) as any machine configuration file, but it's read in first. This parameter is optional. If you do not specify it, MPF will automatically load an MPF configuration file from `/mpf/mpf/mpfconfig.yaml`, and the media controller will automatically load an system-wide configuration file from `/mpf/media_controller/mcconfig.yaml`.

-l: Specifies the name of the log file that will be generated. (This log file contains the same content as the console window output.) By default it creates a file in the `mpf/logs` folder with a filename that's based on the computer host name plus a time and date stamp. For

example, `2014-06-26-13-39-39-mpf-computername.log` represents June 06, 2014 at 1:39:39pm. A separate log file will be created by the MPF game engine and the media controller. Their formats are the same with the example of "mpf" in the game engine log file and "mc" in the media controller log file.

-h: Displays the command-line option help message. (Basically it just describes everything here.)

--version: Prints the version of MPF, the config file version required, and the version of BCP this build of MPF is using. Then it quits. So you use it like this: `python mpf.py --version`.

Using the Windows batch file to launch MPF and the media controller

If you're using Windows, we created a batch file called "mpf.bat" (in the same folder as `mpf.py`) which you can use to easily launch both the MPF game engine and the media controller at the same time. To use it, you type `mpf your_machine`, followed by whatever command line options you want (which will be passed to both). For example:

```
mpf demo_man -x -v -V
```

By default this batch file will launch the game engine and the media controller each in their own separate windows, and then it will leave them open when you exit the game. (It leaves them open so you can see the errors if there's a crash. You can just click the "X" to close each window when you're done.)

If you view the contents of the batch file, you can see some notes about editing the batch file to change this behavior. For example, you can set it so that the MPF game engine launches in the current console window instead of popping open a new window, or you can configure it so that the popup console windows are automatically closed when MPF exits.

Using the Linux shell script to launch MPF and the media controller

The Linux shell script method is nearly the same as the Windows method above. Just run `mpf.sh` rather than `mpf.bat`:

```
./mpf.sh demo_man -x -v -V
```

Current Features

As we mentioned already, the MPF is a work in progress and not yet complete. That said, here's a list of features that are actually implemented so far.

You can also read about the [features we're working](#) on for the next release, as well as our [long-term list of features](#) we want to get done before "v1". (And you can read our [super-long-term list of features](#) that will probably take years to do!)

Major Features of MPF that are real today

- Support for [Multimorphic](#) *P-ROC* and *P3-ROC* pinball controllers (with either Williams WPC, Williams System 11, Data East, Stern Whitestar or S.A.M., or P-ROC driver boards), [FAST Pinball](#) *WPC*, *Core*, and *Nano* controllers (with either FAST or WPC driver boards) and virtual (software only) games without hardware attached.
- Support for Open Pixel Control LED controllers, including the [FadeCandy](#).
- Hardware support for coils, switches, matrix-based lights, RGB & single-color LEDs, flashers, and GI strings.
- Intelligent device support for flippers, auto-firing coils (slingshots, bumpers, etc.), ball devices (things that hold balls like the trough, VUKs, locks, poppers, etc.), diverters, and EM score reels. (We say "intelligent" support because it's more than just being able to fire a coil when a switch is hit. MPF understands what the device is, how it fits into a pinball game flow, what state it's in, etc.)
- Intelligent management of shots and shot groups, including automatic per-player state management, automatic integration with lights, shots based on sequences of switches, and a shots profile manager that lets you apply different behavioral profiles to shots based on game modes.
- Full DMD and LCD support (including traditional DMDs, color DMDs, color RGB DMD matrix displays, and modern HD displays), with multiple types of display elements, slides, transitions, decorators, and support for TrueType fonts. On-screen and physical DMDs are supported.
- Machine and game controller modules which manage the overall machines and game flows. Right out of the box you can play complete games, cycle through balls and players, etc.
- Intelligent ball and playfield tracking, including balls in play, balls on the playfield, and automatic control of ball devices and diverters to move balls to where they need to be.
- A score controller which lets you configure shots, switches, or other game events that add to the player's score and the player's progress in other areas (ramps, loops, aliens destroyed, etc.).
- Game mode support. (Run one or more game modes at once, each with their own priorities, display slides, scoring, shots, sounds, display effects, lighting, and game logic, all which loads when the mode starts and unloads when the mode ends).
- Built-in modes to run the attract mode, the game, and high score, credits, and tilt.
- Ball save, ball lock, and multiball modules.

- A light show controller which lets you build complex coordinated "shows" of lights, LEDs, coils, and GI. You can create playlists, layers, and play multiple shows at once.
- Sound & audio support, including multiple tracks, dynamic loading & unloading of sounds, track-specific volume control, sound priorities, and a sound queue.
- "Logic Block" modules which let you sequence flowchart-style game logic.
- Timers which let you start, stop, pause, and extend all the timers you need (timed modes, shots, countdowns, count ups, hurry ups, etc.).
- Full per-player variable and settings support. Save/restore anything on a per-player bases (shots, objectives, goals collected, targets hit, etc.)
- A game auditor which records and saves game information, high scores, game events, total switch hits, etc. (This is highly configurable.)
- A data manager which reads and writes data from disk, including audits, earnings, machine variables, high scores, etc.
- A keyboard interface which lets you simulate switch actions with your computer keyboard. (Great for testing!)
- A multi-language module which can replace text strings on the fly with alternate versions for other languages.
- An OSC interface which lets you interface, control, and view status of your pinball machine via a tablet or phone.
- An EM-style score reel controller (in case you want to [convert an EM game to solid state](#)).
- A plugin architecture which allows you to write your own plugins to extend baseline functionality.
- A "scriptlet" interface lets you add Python code snippets to extend the functionality you can get with the configuration files.
- A switch "player" which lets you play back timed sequences of switches for automated testing and simulation.
- And the best part: The bulk of your game configuration and logic can be built via text-based configuration files. If you don't want to be a "coder," you don't have to be. (Though the plugin, scriptlet, mode coding, and [API documentation](#) mean that if you want to use MPF for your groundwork and code your game in "real" Python, that's fine too.)

Working with physical pinball machines

If you want to use the Mission Pinball Framework to write software for a real pinball machine, you need some way to get your code onto the machines. Unfortunately you can't just install Python on a pinball machine. "Real" machines don't have normal computers controlling them, rather, they're based on custom-built control systems which run code

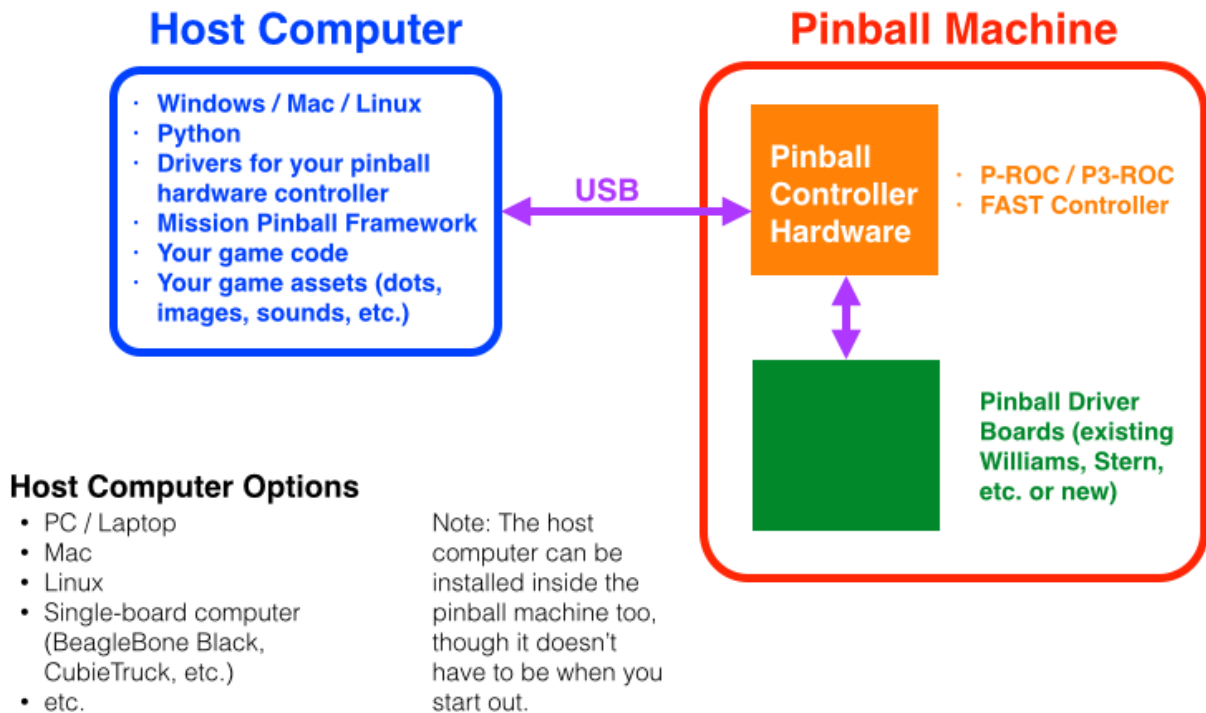
compiled specifically for each platform. Plus, a lot of pinball machines are old, meaning their hardware is weak by today's standards. (For example, all those mid-90s Williams WPC games that we love are powered by a Motorola 6809 CPU running at 2 MHz with 64 KB of RAM. (Yeah, that's 64 *kilobytes*, not megabytes.) So you're not getting the Mission Pinball Framework on there. (If you want to write software which runs on the original 6809 Williams hardware, check out Brian Dominy's [FreeWPC project](#). You can write new game code in C++, compile it for the 6809, burn it to a ROM chip, and replace your existing machine ROMs with your custom ones. Crazy! And awesome!)

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 or FAST Controller. (More on those in a bit.) 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.

If you want to use the MPF to power a new *custom* pinball machine that you build yourself, you can buy new custom driver boards from the same people who make the P-ROC or FAST Pinball controllers. (These P-ROC or FAST driver boards are typically only used with custom machines, because if you're replacing the main CPU board in an existing machine then it's easiest to just use the existing driver boards as they are. i.e. there's no need to rewire anything.) The important takeaway is that hardware controllers from FAST & P-ROC can control the original Williams/Stern driver boards or their own custom driver boards. (More on that in a bit too.)

The most important thing to understand is that the P-ROC and FAST hardware controllers *do not actually run your Python-based code themselves*, rather, they act as the interface between the physical pinball machine and a host computer running your game code (though they are configured to respond to some time-critical events themselves, such as directly firing coils based on switches like the flippers, slingshots, and pop bumpers).

How MPF runs a pinball machine



At this point you might be thinking, "What??? I don't want to have my laptop attached to my pinball machine to run it!" While it's true you *could* just put a laptop in the bottom of the machine to run it, most people end up buying a small form-factor PC board running Linux or something and put that inside the machine to run the actual python game code. There are several options for this today, including the [BeagleBone Black](#), [ODROID](#), [Raspberry Pi 2](#), etc. Or you can use a mini-ITX Intel-based board and run Windows or put a Mac Mini in your cabinet. Really it can be anything as long as it can run Python.

How the Mission Pinball Framework interfaces with hardware controllers

One of the primary goals of MPF is to be hardware-independent. In other words we want MPF to work with any hardware pinball controller out there.

We achieve hardware independence by abstracting all the hardware calls to our [platform interface module](#), and then our platform module talks to a hardware-specific module which is specific to the actual hardware platform you're using. (So in our `/platforms` folder we have a Python module called `p_roc.py` which is used to talk to a P-ROC, `p3_roc.py` talks to a P3-ROC, `fast.py` talks to FAST controllers, etc.)

It's sort of a "we know the low-level details of the different hardware controllers so you don't have to" kind of thing. :)

This also means that a game programmer, you have the flexibility to change your hardware platform 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! If you're using your custom code in an existing Williams WPC game, you can literally switch platforms by changing a single line in a config file. [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.

Choosing a pinball machine

Now that you know the conceptual overview of how MPF works, how do you choose a physical pinball machine to use it with? You have three options:

- Use an existing machine, keep it as it is, and write your own rules using MPF.
- Use an existing machine, strip it down and "retheme" it to your own theme, and use MPF for the software.
- Build a new machine from scratch.

Certainly the easiest way to get started is to use an existing machine. In fact many machines will let you replace the built-in CPU board with a P-ROC or FAST controller in about 5 minutes, and you can be up and running with your own custom code in a few hours!

The easiest existing machines to work with:

- Williams / Bally / Midway "WPC"-era (1990-1998, Funhouse through Cactus Canyon)
- Stern S.A.M (2006-2014, World Poker Tour through The Walking Dead)
- Sega/Stern Whitestar (1995-2006, Apollo 13 through Terminator 3)

These machines are easiest because you can just "drop in" a modern controller. You can use a P-ROC or FAST WPC controller for WPC machines, and you can use a P-ROC with Stern machines. In both of these cases you reuse the existing driver boards, so all you need is the P-ROC or the FAST WPC controller (about \$300 each) and you're all set.

After these, machines, the next-easist batch of machines are:

- Williams / Bally System 11 machines
- Data East (which were clones of the System 11 boards)

To use System 11 / Data East, you need a driver board from Mark Sunnucks (about \$200) that will connect your P-ROC or FAST WPC controller to the existing wiring. (See this [How To guide for more details](#).)

Finally, if you want to use an older solid state machine with MPF (late '70s through early '90s), that's still possible, but you would need to replace the existing driver boards with newer P-ROC or FAST driver boards. You'd also need to build custom wiring adapters to interface the existing wiring to the specific connectors and pinouts the new driver boards

need. This is still pretty straightforward and something you can probably do in a day or so. This will cost you around \$600 (depending on the machine) because you'd have to buy the P-ROC or FAST controller as well as a few hundred dollars in new driver boards.

If you really want to go crazy and use an EM machine with MPF, that's possible too. (In fact [we did this](#) with a 1974 Gottlieb Big Shot machine.) In this case you most likely have to remove all of the existing wiring and solder all new connections, so you're looking at probably 100 hours (again, depending on the machine). This might be a bit more expensive (perhaps \$800-\$1000) because all those score reels have lots of extra switches, lights, and coils you need to hook up, so you'll need more driver boards. (MPF has a score reel module so you can hook right into the mechanical score reels with modern technology. It's very cool!)

3. Installing MPF

Installing MPF is easy, and there are a few ways you can do it. (Note this page is only about the actual installation of the MPF software and hardware drivers. You can use our [step-by-step tutorial](#) to walk you through actually building your game config with MPF.)

OS Platforms supported

- Windows 7, Windows 8, Windows 10 (32-bit)
- Windows 7, Windows 8, Windows 10 (x64 / 64-bit), using 32-bit Python.
- Mac OS X, using the built-in Python 2.7 (some caveats apply, should be fixed in MPF 0.22)
- Linux (on both Intel and ARM hardware)

Installation guides

- [Windows](#)
- [Linux \(Debian, Ubuntu, etc.\)](#)
- [Mac OS X](#)

Windows

This page explains how to get MPF installed on a Windows computer. These instructions work for Windows 7, 8, or 10, and they are valid for both 32-bit and 64-bit Windows.

You have two options for the installation:

- You can use our completely automated all-in-one installer. This will install everything you need from scratch, including Python, all the support libraries you need, your P-ROC/P3-ROC or FAST hardware drivers and libraries, and MPF itself. This should take 3-5 minutes.
- You can do a manual install. This is still very easy, but nice if you want to customize how things are installed or if you already have some components. This should take about 10 minutes.

Let's look at each of these.

Using our all-in-one Windows installer

Our all-in-one Windows installer will literally install everything you need starting from scratch, including Python 2.7, git, Pygame 1.9.2, PyYAML, PyOSC, the drivers for the FAST and/or P-ROC/P3-ROC pinball controllers, and MPF itself. It will work with 32-bit Windows and 64-bit Windows. Use it if you want an automated way to get everything all at once. It should only take about 3 minutes. Steps include:

1. Download the [installer zip file](#).
2. Unzip it
3. Run or double-click `install.cmd`. (Be sure to run this from the location where you unzipped the package to. It will not work if you run it from within the zip file itself.)
4. Answer the few questions... done!

Note that this will install MPF into the `c:\pinball\mpf` folder. There's nothing special about that folder and you can move it to wherever you want after it's installed.

Manually installing on Windows

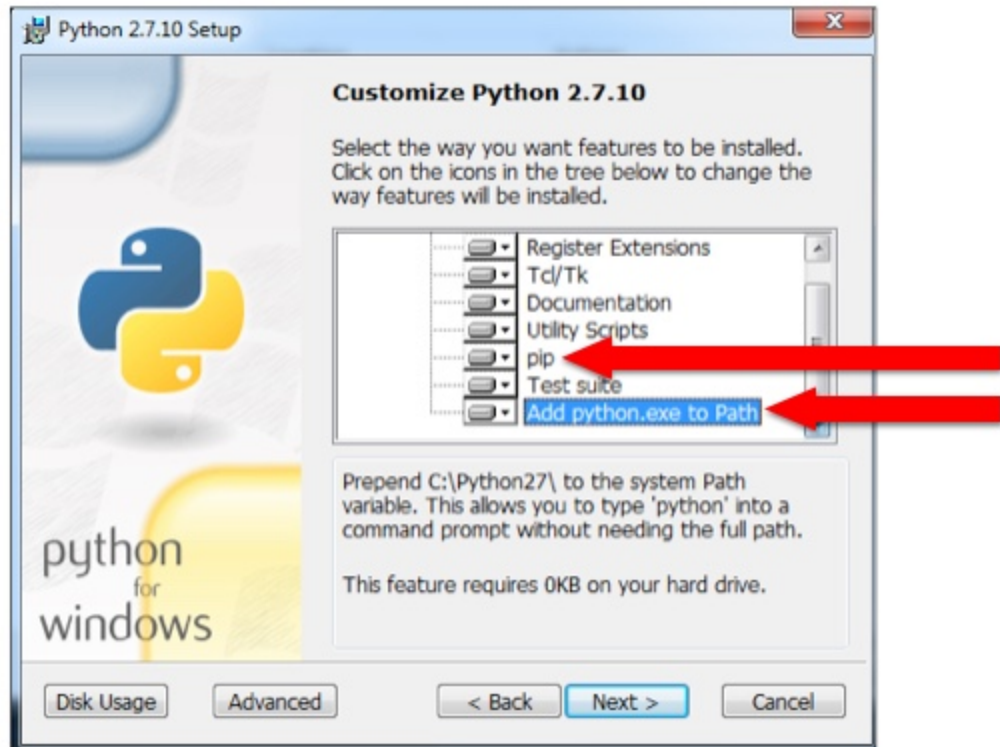
If for some reason you don't want to use the all-in-one Windows installer, you can manually install everything. This is nice if you already have some things installed (like Python) or if you want to use custom options or paths. Even though this is a manual process, it's still very easy and takes less than 10 minutes. In fact here's a video walking through the entire process (which is also detailed in the steps below).

<https://www.youtube.com/watch?v=8EfEaSKceW0>

1. Install Python 2.7, 32-bit

You can download and install Python from <https://www.python.org/downloads/>. Choose the latest 2.7.x 32-bit version. (Note that you need to install the 32-bit Python even if you're using 64-bit Windows.)

When you install it, ensure that the options for "pip" and "Add python.exe to Path" are enabled:



1.1 What if you already have Python installed?

If you already have Python installed, that's fine, but we need to check to make sure it can be used with MPF.

First, make sure that the version of Python you have is 2.7.x. You can do this by running `python -V` from the command prompt. (Note that's an uppercase "V")

Next, verify that you're running the 32-bit version of Python since some of the plug-ins we use require it. (This is fine for now. We may look at supporting 64-bit in the future but it's not a priority at the moment. MPF doesn't require enough memory to require 64-bit.)

The easiest way to know for sure is just to run a quick Python program to test the maximum size item that Python can work with, which will tell us whether it's 32-bit or 64-bit.

To do this, from the command line, run this:

```
python -c "import sys;print(sys.maxsize < 2**32)"
```

If it prints "True" on the screen, then you know you have the 32-bit version of Python and you're fine. If it prints "False" then you have the 64-bit version and you need to install the 32-bit version of Python and run that.

2. Install PyYAML

Next you need to install an add-in module to Python called PyYAML which lets Python read & write YAML files. ([YAML](#) is the file format MPF uses for all sorts of things, including its configuration files, audit logs, high score data, etc.)

The easiest way to do this is to go to a command prompt and run:

```
pip install pyyaml
```

"pip" is a Python package manager. When you run it, it goes online and downloads and installs the package name you gave it. So `pip install pyyaml` is using "pip" to "install" the package called "pyyaml." (You have to be online for this to work.)

If you get an error about pip not being a valid command, that probably means you don't have pip installed, so you can get it from [here](#). (pip was not included in Python before 2.7.9.) Or you can [download and install PyYAML manually](#). (Any version should work. We do not need the version with the LibYAML bindings.)

3. Install Pygame

MPF uses another Python add-on library called *Pygame* which lets Python interact with graphics, sounds, and keyboard libraries.

Unfortunately Pygame is not part of the online Python repository where pip downloads packages from, so you'll have to find and install it manually.

3.1 Check to see if you have Pygame installed already

First, you should check to see if you have Pygame already. (If you just downloaded and installed Python then you don't have it, but if you've had Python for awhile then it's possible that you already have Pygame.) You can do this from the command prompt by typing:

```
python -c "exec('import pygame\nprint pygame.version.ver')"
```

This command is a little crazy looking, but trust us—it works! Basically this is a one-liner which tries to import Pygame and then prints the version.

If this command gives an error which says *ImportError: No module named pygame*, then that means you don't have Pygame and will have to download and install it as outlined below.

If this command just prints a number or some other words, like "1.9.1" or "1.9.2pre" or "1.9.2pre-svn3227", then that means Pygame is installed. For MPF, you need Pygame 1.9.2 or newer.

3.2 Install Pygame 1.9.2a0

The versions of Pygame are kind of confusing at the moment. The Pygame group is supposedly in the process of releasing Pygame 1.9.3 which should have new installers and be really nice, but they've been working on that for

months

years so who knows when they'll be ready? In the meantime, you can install Pygame 1.9.2a0 (32-bit) for Python 2.7 from the [Pygame downloads page](#). (Note that the "a0" in the very means "alpha 0" which sounds scary, but this has been the latest version of Pygame literally since 2009, and so far it seems fine.)

4. Download the latest Mission Pinball Framework from GitHub

If you already have MPF installed, you can check what version it is by opening a command prompt and changing to the folder that has your `mpf.py` file and running `python mpf.py --version`. We use [semantic versioning](#) for MPF, with MPF versions being in the `x.y.z` format. You can see what the latest version is at <https://github.com/missionpinball/mpf/releases/latest>.

There's also a link on that GitHub page to a `.zip` or `.tar.gz` file (they're both the same) which contains MPF. The download file will be named `mpf-<version>.zip` or `mpf-<version>.tar.gz`, depending on which link you clicked. Unzip the download package to whatever location you want. (This can be literally anywhere. It does not need to be in your Python folder.)

Note that there are several branches for MPF in GitHub. The master branch is the most stable and is updated once a month or so. The dev branch represents the next version we're working on. We try to keep it somewhat stable, but there are usually more bugs. The dev branch is updated a lot—probably 10 times a week or more!

5. Install your pinball controller's hardware drivers

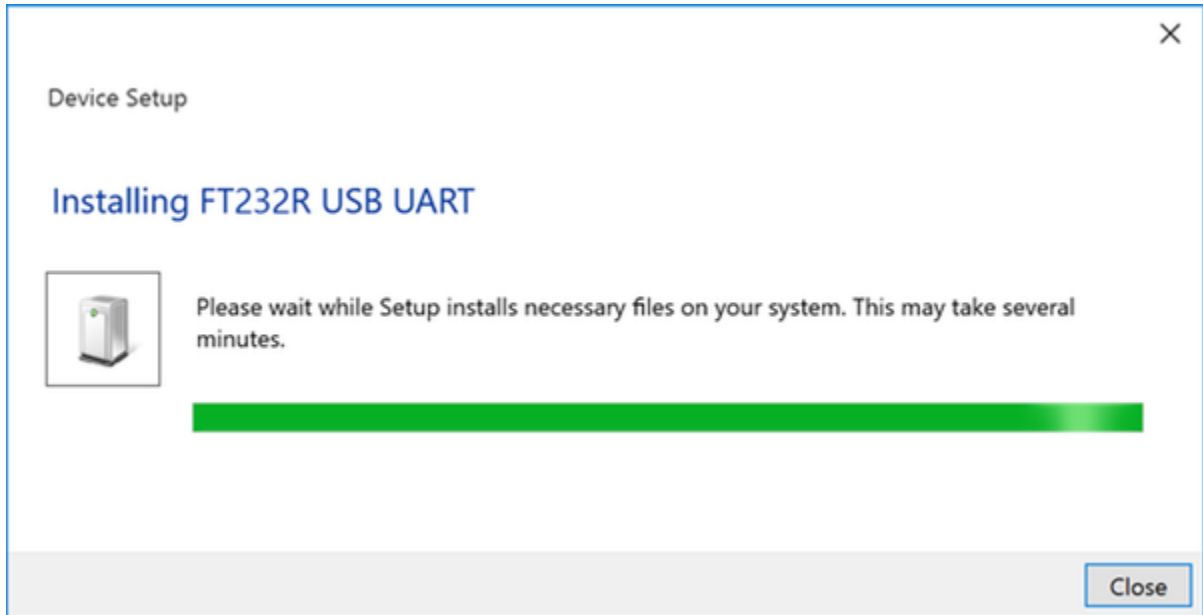
If you're using MPF to control a physical pinball machine via a P-ROC/P3-ROC or FAST Pinball controller, you need to install the software drivers and libraries they need. We have instructions here for both, starting with the P-ROC/P3-ROC:

For P-ROC/P3-ROC:

1. Install the FTDI driver from [this page](#). (You can use the "Setup Executable" link on the right for a single exe that works with 32-bit and 64-bit Windows.)
2. Download our [zip file](#) which has everything else you need.

3. Run the file `pinproc-2.0.win32-py2.7.exe` from that zip file. Just click next next next through the popups. (This is also the file you use for 64-bit Windows, since we're using it with 32-bit Python.)
4. Copy the other three DLL files from the zip file into your `c:\Python27\Lib\site-packages` folder.

That's it! When you first plug in and power-on your P-ROC or P3-ROC, you'll see this screen pop up:



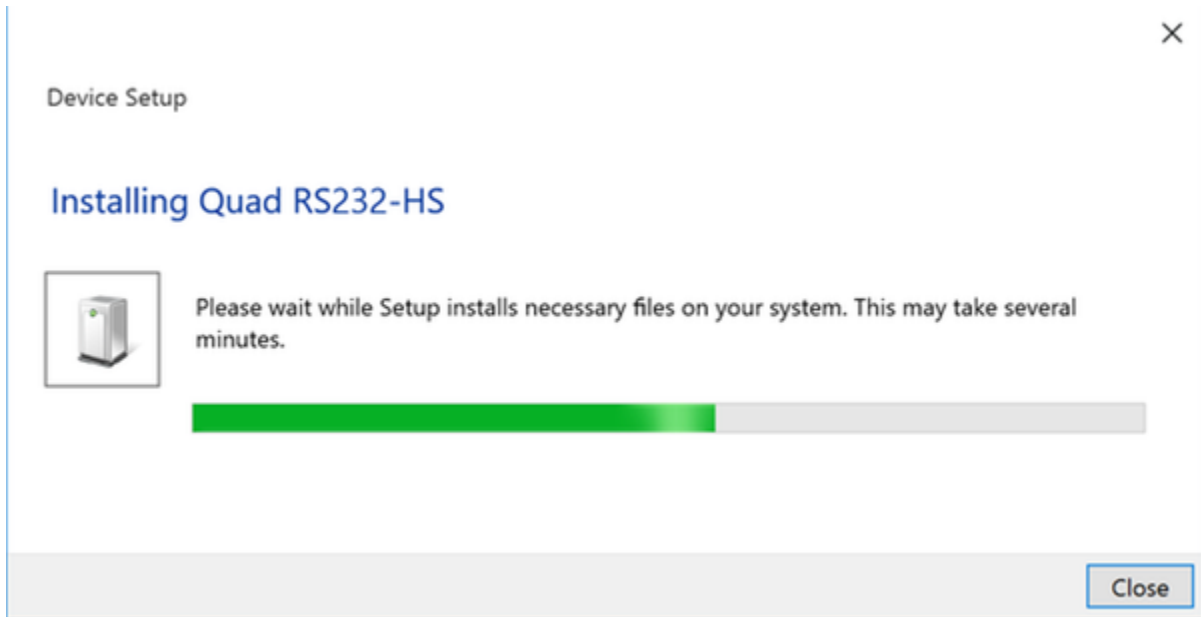
It's just a progress bar which shows Windows configuring the drivers. You don't have to click anything to get it started and it should only take a 5-10 seconds. It will only happen the first time you use the device.

For FAST Pinball:

1. Install the FTDI driver from [this page](#).
 1. Be sure to get the version (32-bit or 64-bit) that matches the version of Windows you're using.
2. Install the PySerial extension for Python.
 1. The easiest way to do this is to open a command prompt and run `pip install pyserial`.
 2. If you don't have pip, you can download an installer for PySerial from [here](#).

Once this is done, when you plug in and power on your FAST controller, you might 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 use the device.)

Then if you go into your device manager, you should see four new COM ports appear. These are "virtual" COM ports that your computer talks to via USB, and these are the ports that MPF uses to communicate with your FAST pinball controller. These ports will disappear when you power off or unplug your FAST controller. Again the exact way you see your COM ports varies depending on OS.

Next Steps

At this point you're all set! Check out our [tutorial](#) which will walk you through running the *Demo Man* sample game that comes with MPF and will show you how to build your own game!

Linux (Debian / Ubuntu)

This page explains how to install MPF (and all the related components) on Linux. Our all-in-one Debian installer can be used for any Debian-based flavor of Linux, including Debian derivatives like Unbuntu, Xubuntu, etc. This installer works with x86, x86-64-bit, and ARM-based systems (Raspberry Pi, Beaglebone Black, ODROID, etc.).

This script will give you the option to also install the drivers for the P-ROC and P3-ROC. (The drivers that FAST Pinball controllers need are built-in to Debian.)

1. Download the [installer zip file](#).
2. Unzip it
3. Run `./install`.
4. Answer the few questions...
5. Reboot

Done!

Next Steps

At this point you're all set! Check out our [tutorial](#) which will walk you through running the *Demo Man* sample game that comes with MPF and will show you how to build your own game!

Mac OS X

Note: Mac support is new for us. We have it working on all hardware platforms (P-ROC, P3-ROC, and FAST), though right now getting it running for P-ROC/P3-ROC requires following a [58-step process](#). We're working on getting the compiled drivers extracted from that so we can build a nice all-in-one installer like we have for Windows and Linux. We expect to have that done soon. (Sept 2015)

Also note that the Movies module is not available on the Mac. This is because Pygame for Mac does not support movies. We will be moving MPF off of Pygame in late 2015, but until then, no movies on the Mac.

Here are some raw notes if you want to try with Mac in the meantime:

- Python 2.7 is built in.
- Install Pygame (1.9.2pre for Apple-supplied Python 2.7)
- Install pip
- Run `sudo pip install pyyaml`
- Install the [FTDI driver](#) for Mac.
- Install the pinball controller interface libraries:
 - FAST, run `sudo pip install pyserial`
 - P-ROC/P3-ROC, follow the 58-step procedure from above
- Download MPF from GitHub

Next Steps

At this point you're all set! Check out our [tutorial](#) which will walk you through running the *Demo Man* sample game that comes with MPF and will show you how to build your own game!

4. Tutorial

So you want to write your own software for a pinball machine? Awesome! (Hopefully you're a little crazy, but seriously, this is awesome. Congrats!)

We're going to attempt to outline everything you need to do to get the Mission Pinball Framework (MPF) up and running on your machine. It's easy, even if you're not a computer programmer! Just keep in mind that MPF is very much a work in progress (as is this tutorial). We're not even to "v1" yet. But we've tried to document how to get up and running as best we can.

Here's how this process works:

1. Follow all the steps in the tutorial, in order. (Each step builds on the previous one, and every machine needs the things from every step.) This will give you a basic, playable game, complete with multi-player support, and attract mode, lights, shots, points, and a basic game mode.
2. When you're done with the tutorial, browse our [How To guides](#). (These are like the "cook book" or "recipe" approach.) The How To guides can be followed in whatever order you want, and you don't have to do all of them.

By the way, this is tutorial outlines the exact process we follow ourselves whenever we walk up to a new machine to get MPF running. We literally open the documentation to this page and follow along!

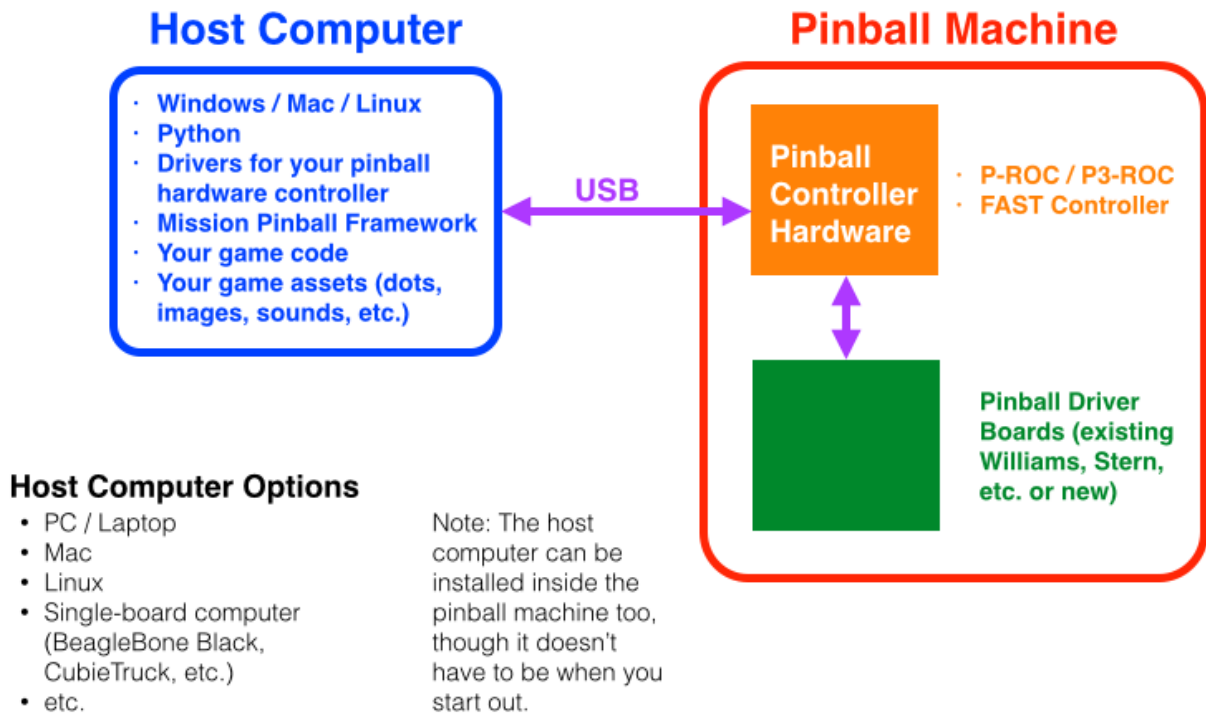


START THE TUTORIAL

Step 1. Prerequisites

If you haven't done so already, it's probably not a bad idea to read the [Overview](#) section of this documentation, including the parts about [understanding the MPF download package](#) and [the file & folder structure](#).

How MPF runs a pinball machine



Once you look at that, there are a few things you need to know about MPF:

The **MPF software runs on a host computer plugged into a pinball hardware controller**. MPF is written in Python 2.7, so anything that can run Python 2.7 should be able to run MPF. (Windows and Linux are both fine today. Mac kind of works with some caveats and should be fully fixed in the next release of MPF in early 2016.) MPF will not run on Python 3.

The MPF software is self-contained. MPF is literally just a folder full of Python code and configuration files. Of course if you want to control a real pinball machine, you'll need a [pinball hardware controller](#). Today we support both the [Multimorphic P-ROC/P3-ROC](#) and the [FAST Pinball](#) controllers, each of which has its own drivers and Python interfaces.

The good news is **you don't need to have a pinball hardware controller to get started**. We also include a ["virtual" \(software only\) interface](#) which you can use to run MPF without any real hardware attached. In fact you can (theoretically) build an entire game without ever connecting it to a real machine. (Though that's not as fun.)

Step 2. Download & Install MPF

Once you understand what you'll need for MPF, you can move on to actually downloading, installing, and getting MPF running.

At this point it really doesn't matter what type of computer you use for this. Most people do all their initial development work on a Mac or Windows laptop (which they connect via USB to their pinball machine for testing), and then when they get closer to actually finishing their project they switch over to a small Linux-based single board computer (like a Raspberry Pi 2 or BeagleBone Black) or a small Windows system to run their final code. The good news is that all the Python files and configuration files you build are 100% identical regardless of the computing platform (and, frankly, regardless of whether you use a P-ROC or FAST controller), so you can safely do all your development on whatever computer you're reading this document from and then easily transfer it to your final computer later.

(A) Install Python, MPF, and related components

We have a [dedicated page about installing MPF](#) that you can follow to get everything installed. If you're using Windows or a Debian-based Linux (Ubuntu, etc.), then we have installers you can download and use which will automatically install everything in under 5 minutes—including the P-ROC, P3-ROC, or FAST Pinball drivers!

Even if you want to manually install MPF and all its related components, it should only take you about 10 minutes.

Note that this tutorial is based on **MPF 0.21**. See [this how to guide](#) for details on how to check what version of MPF you have.

(B) Run *Demo Man*, a sample game that comes with MPF

Now that MPF is installed, let's run a sample game to make sure that everything is working. Then we'll dig into creating your own game.

One of the development machines we have for MPF is a 1994 Williams *Demolition Man*, and we've included sample game configuration files for a game we call *Demo Man* (which runs on a Williams *Demolition Man* machine) in the MPF download package. (You can [track our progress](#) writing the *Demo Man* game code in our blog.) So at this point you can run *Demo Man* (using the software-only "smart virtual" platform with no physical pinball hardware attached since you probably don't have a *Demolition Man* machine) just to make sure that everything is up and running properly.

To do this:

1. Open a command prompt.
2. Switch to the root *mpf* folder where you just installed MPF. This will be `c:\pinball\mpf` by default if you used our all-in-one Windows installer. You should see a bunch of files in that folder like *mpf.py*, *mc.py*, *mpf.bat*, *mph.sh*, etc., as well as various subfolders including */mpf*, */machine_files*, and */tools*.
3. Then the next step varies based on what OS you're using.

If you're on Windows...

For Windows we have a batch file that will launch the MPF core engine and the MPF media controller (which handles the DMD, audio, and on-screen popup window) together at the same time. You can launch *Demo Man* like this:

```
mpf demo_man -v
```

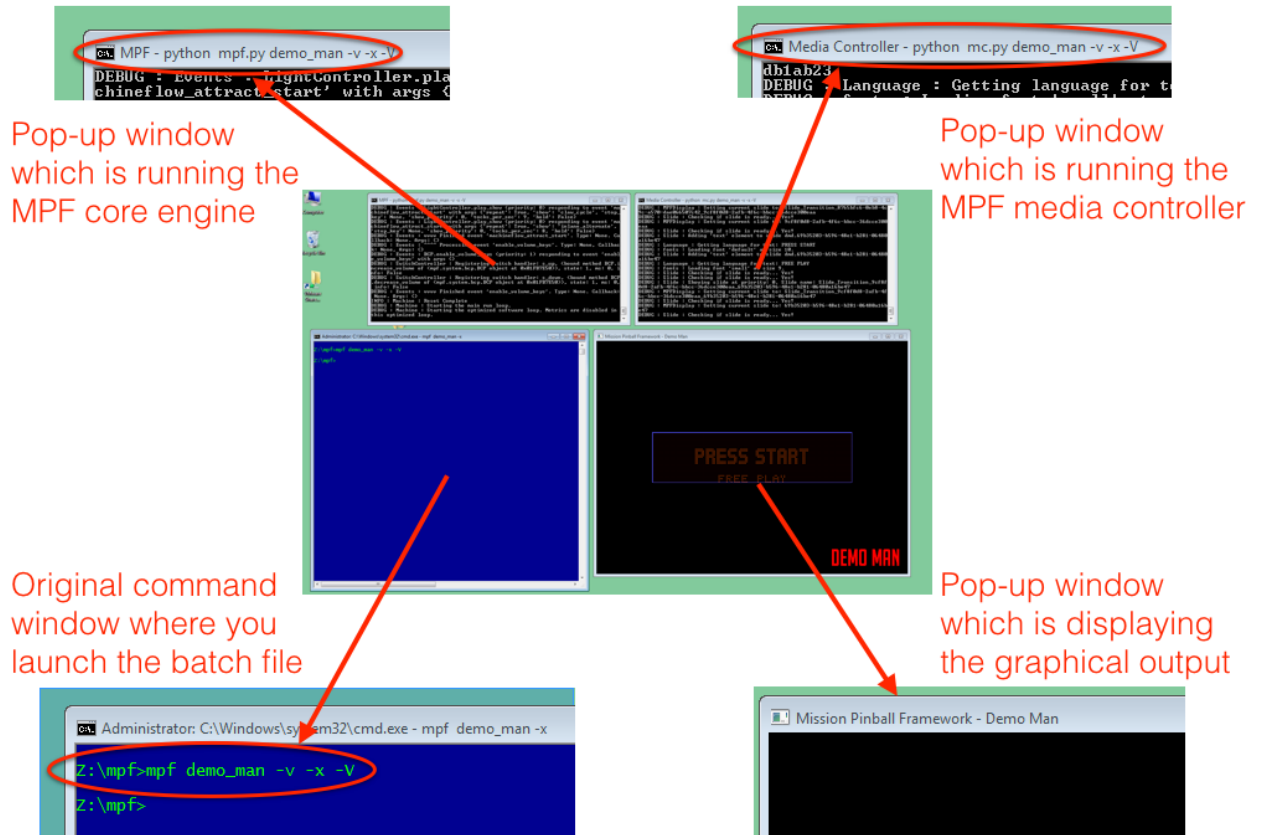
The *mpf* part of that command is running the batch file `mpf.bat`, *demo_man* tells MPF that you're running the *Demo Man* machine code. The `-v` option sets it so the log file is written in "verbose" mode.

You should see three popup windows when you run this:

- A new command window with logging from the core MPF game engine.
- A new command window with logging from the MPF media controller.
- A graphical window which shows you a virtual on-screen DMD of the *Demo Man* game.

If you don't see the two command windows, try dragging the window you do see with the mouse. Sometimes the two windows open on top of each other so it just looks like one window.

As *Demo Man* is running, your desktop should look something like the image below. (Click the image to see it full size.) Also, later in this tutorial we'll show you how to customize this graphical pop-up window to show whatever you want, including text, images, backgrounds, different color dots, change the size of the DMD, etc.



(Note that this screenshot is from an older version of MPF so the exact command line you run is different than what's in the pictures, but the pictures show the gist of what's going on.)

If you're using Linux...

For Linux platforms, we haven't yet created a universal shell script to to automatically launch both the MPF core engine and the media controller. So instead, open up two terminal windows.

In the first window, launch the media controller for *Demo Man*, like this:

```
python mc.py demo_man -v
```

(Note that depending on how you have Python installed, you might need to specify a different Python executable besides just plain `python`.) The `demo_man` parameter tells the media controller to launch with the "demo_man" config, the `-v` tells it to write a log in verbose mode, and the `-V` tells it to also write the verbose messages to the console window.

Then in your other terminal window, launch the MPF core engine for *Demo Man*, like this:

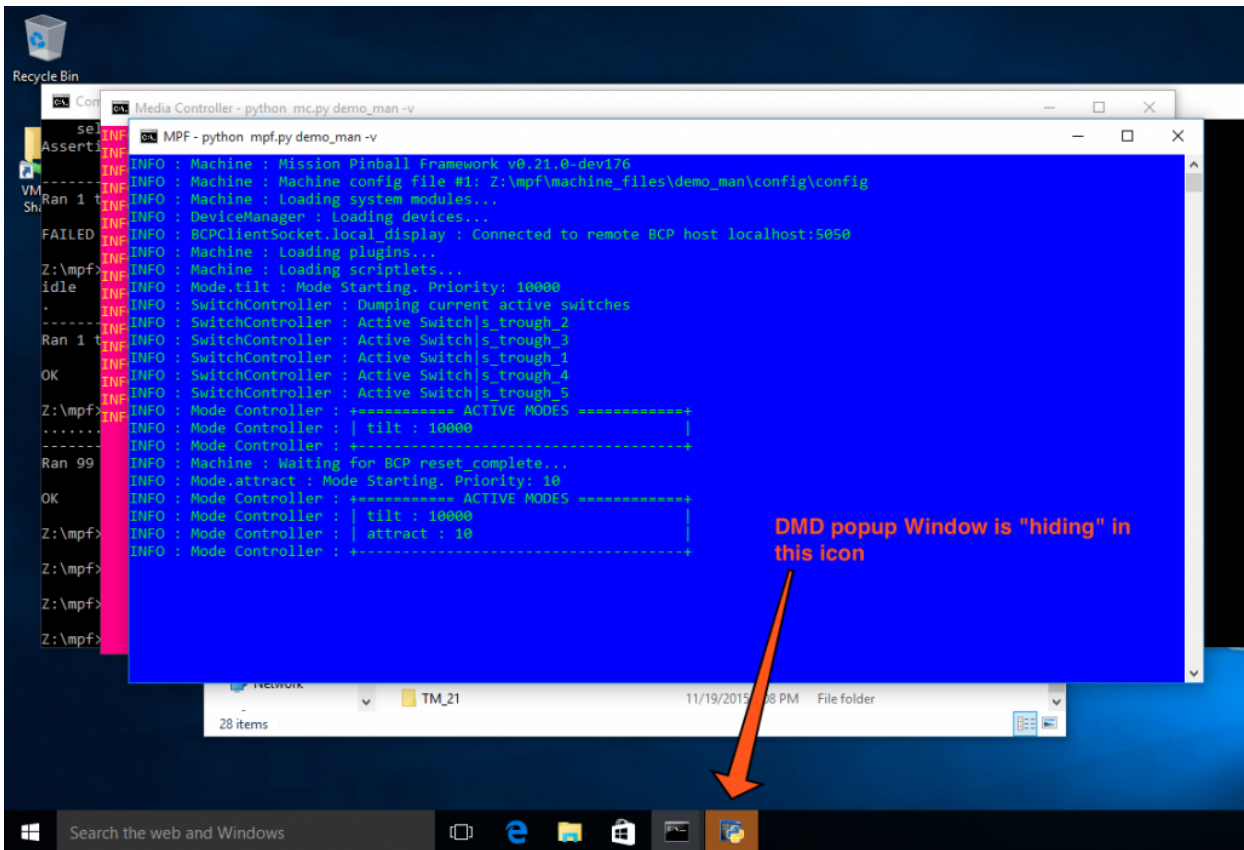
```
python mpf.py demo_man -v
```

The options here are similar to the media controller options, except that the Python script you're running is `mpf.py` instead of `mc.py`.

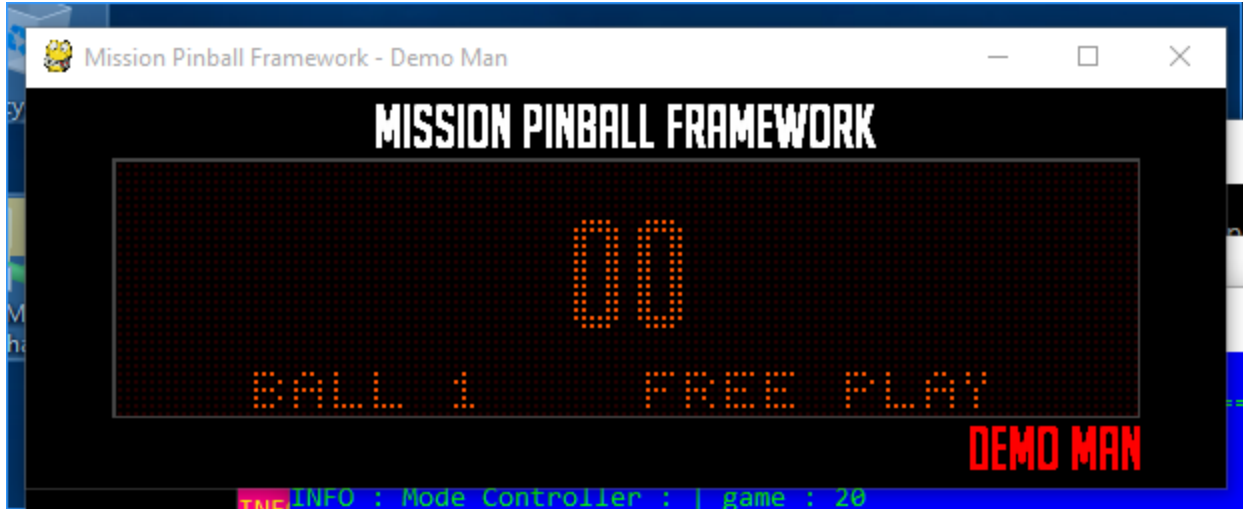
(C) "Play" your first game

Since you don't have physical hardware attached, you can use the keyboard to simulate machine switch changes

If you don't see the DMD window pop up, make sure it isn't hiding in the task bar, like this: (click for full size image)



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 stoop and it change to a score screen showing a score of *00* and *BALL 1 FREE PLAY*, like this:



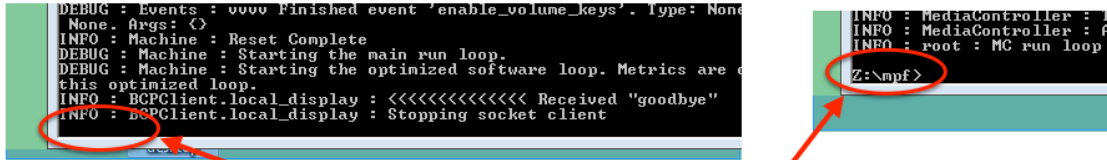
If your speakers are on you should also hear a music loop playing. (Depending on your system, you might not hear the music when the DMD window doesn't have focus.)

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.

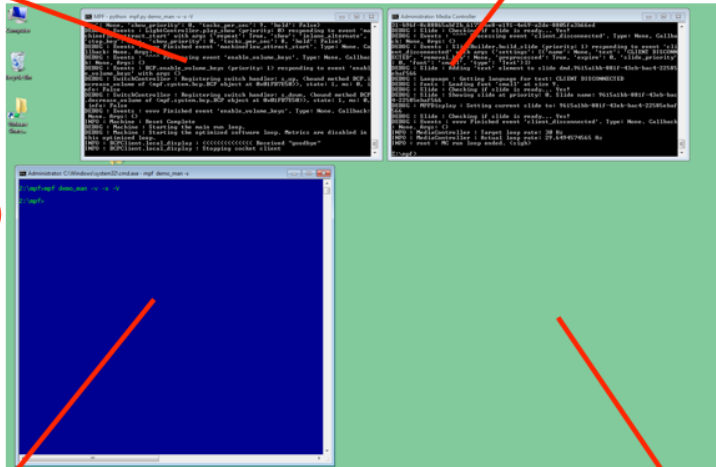
You can hit the "1" key to simulate the ball draining and entering the trough. Then you can hit the "L" key again to launch the ball into play again. You can also press the "S" key additional times during Ball 1 to add additional players. When you play through a complete game (3 balls per player), the machine should go back into attract mode.

You can quit the game by making sure the *Demo Man* popup window is in focus and hitting the *Esc* key. If you ran MPF from the Windows batch file, then you can just close the two popup command windows by clicking the "X" button in the corner of each. If you're on Mac or Linux, the media controller process will stop when you hit *Esc* in the graphical window, and you can stop the MPF core process by clicking in that terminal window and hitting *CTRL+C*. (You can use the MPF config files to automatically launch and stop each process as the other starts and stops. More on that later.)



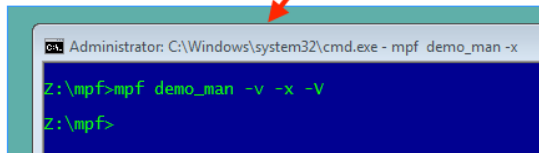
MPF game engine is still running. (You can configure it to auto stop when the media controller stops in the options.)

Media controller has stopped



Original command window where you launch the batch file

Pop-up window is gone since the media controller stopped



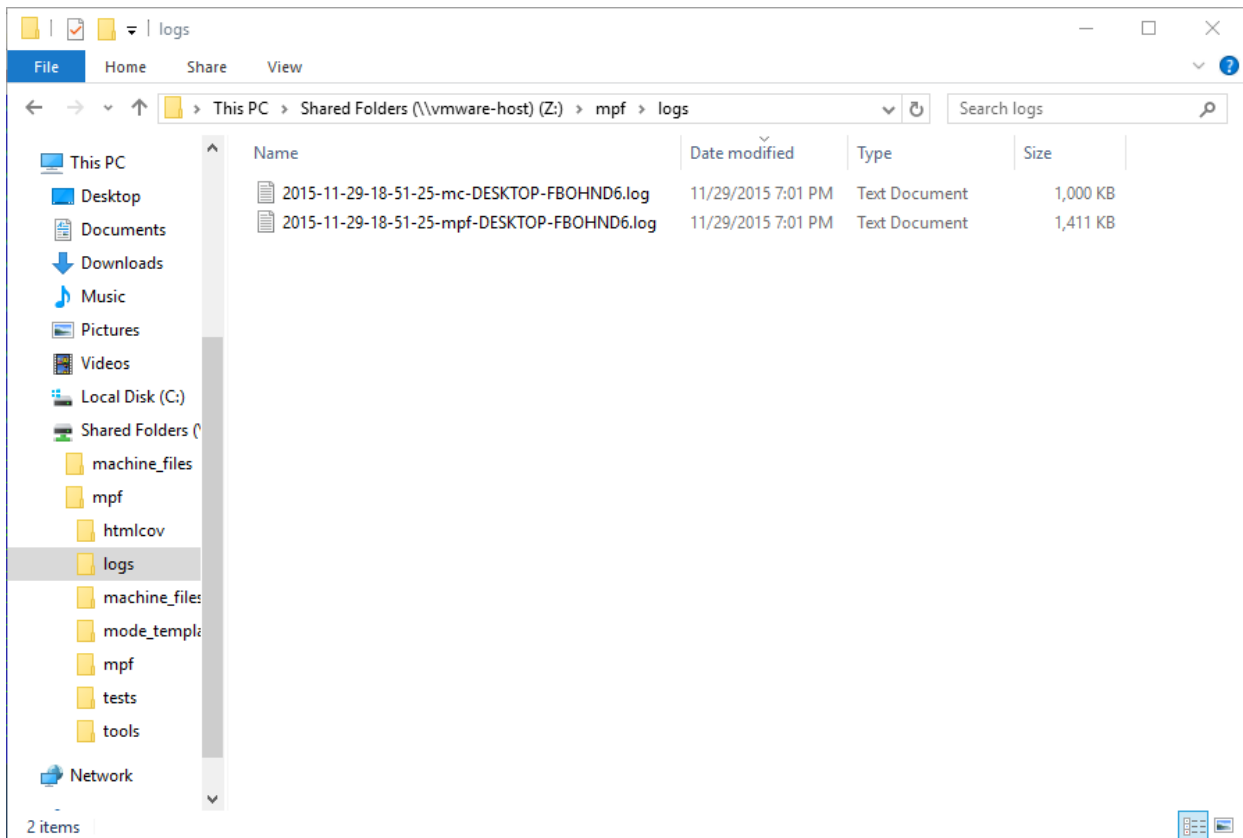
To summarize the instructions for "playing" a game from the paragraphs above:

1. Launch both the MPF core engine and the media controller and make sure you see the two processes running along with 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 hear 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 close the media controller.
10. Click in the window with the MPF game engine and press *CTRL+C* to exit from it.

(D) Take a look at the log files to see what just happened!

The MPF core engine and media controller always create log files. Since we ran these with the `-v` command line option, we'll have "verbose" log files to look at.

A new folder called `/logs` should have been created in your root `mpf` folder. Inside that folder you should see two new log files—one from the MPF core engine and one from the media controller. The files are named with a date & time stamp, then either `mpf` or `mc` (for MPF core or media controller), then the host name of the machine they were run on. Here's an example from playing a single-player complete game (3 balls) and entering initials for a high score:



Note that since we did the log files in verbose mode they are huge! About 2.5mb from a single game where basically nothing happened. :) So verbose mode is really verbose, and only something you'd use while troubleshooting.

You can double-click on them to see all the crazy things that MPF does behind the scenes. The files have a `.log` file extension, but they're just regular text files. On Windows if you double-click them, they'll open with Notepad. On a Mac they'll open with an app called Console which is a log file viewer.

If you're just reading this documentation without following along and you'd like to see log files, here are direct links to the two files mentioned above:

- [MPF core engine](#) (1.5MB .log file)
- [MPF media player](#) (1MB .log file)

Step 3. 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.

(A) Make sure you're running the right version of MPF

This tutorial is written for MPF version 0.21. So before we go any further, let's make sure that you have that version of MPF. Instructions for checking the version of MPF you have are [here](#). Make sure you have v0.21.x.

(B) Understand the "machine folder" concept

In MPF, we use the term *machine folder* to describe all the configuration files, code, images, videos, sounds, audits, and everything else you need for a pinball machine in one single folder. These folders are portable, so 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 that we call these "machines" and not "games" because in MPF language, a "game" is an actual game-in-progress—what you're really creating a pinball machine config, not a pinball game config.)

You can have multiple different pinball machine projects (and multiple versions of each) all living side-by-side on your computer. (We have dozens of them on ours!) Each is in its own folder.

Your machine folder can be anywhere you want. It does not have to be inside the MPF folder. In fact we actually recommend that you *do not put your machine folder inside the MPF folder*, because that way when we update MPF you can just cleanly delete the old MPF folder and replace it with the new one without overwriting the files in your machine folder.

That said, there is a folder in the MPF package called `/machine_files` that has some sample machine configurations you can look at. For example, the *Demo Man* game from the previous step is in the `<your_mpf_root>/machine_files/demo_man` folder. But those are just examples that are part of the MPF package and you shouldn't put your own machine folder there.

Before we dive into creating your machine, take a few minutes to look at all the various files and folders in the `machine_files/demo_man` folder. This just gives you a taste of what you'll ultimately be creating. Note that *Demo Man* is *not* a complete game. (Again, you can follow our progress [here](#).) In fact it's not really a goal of ours to ever build a complete *Demo Man*

game. It just so happens that we have a *Demolition Man* machine handy, so it's the guinea pig where we test out all the MPF functionality. But it's still worth looking through the folder to see the types of things we're doing. You'll see folders for configurations, modes, assets, etc... Just poke around and see what things look like.

(C) Create your machine folder

Okay, so let's get started with your own game's machine folder. The first step is to create an empty folder somewhere. (Again, do *not* put this in your MPF folder.) Maybe something like `c:\pinball\your_machine`.

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, e.g. `c:\pinball\your_machine\config`.

Next, create a file called `config.yaml` in your `config` folder. This will be your main configuration file which will ultimately be thousands of lines long and which will contain all the configuration and settings for your machine. This file should be at `<your_machine_folder>/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`.

Now let's try running your machine! Make sure you're in your MPF root folder (the one with `mpf.py` in it, not your machine folder) and run the following command: (Replace `c:\pinball\your_machine` with the path to your actual machine folder.)

```
python mpf.py c:\pinball\your_machine -v
```

Note that we're running Python directly to just start the MPF core engine rather than running the batch file which launches the core engine and the media controller.

We use `-v` (lowercase) to tell it we want verbose logging to the log file.

Your result should look something like this. (This example is from Windows, but it's similar for Linux):

```
C:\pinball\mpf>python mpf.py c:\pinball\your_machine -v
INFO : Machine : Mission Pinball Framework v0.21.0
INFO : Machine : Machine config file #1: z:\machine_files\your_machine\
config\config
ERROR : ConfigProcessor : Config file z:\machine_files\your_machine\config\
config.yaml is version 0. MPF 0.21.0 requires version 3
ERROR : ConfigProcessor : Use the Config File Migrator to automatically
```

```

migrate your config file to the latest version.
ERROR : ConfigProcessor : Migration tool: https://missionpinball.com/docs/
tools/config-file-migrator/
ERROR : ConfigProcessor : More info on config version 3:
https://missionpinball.com/docs/configuration-file-reference/
config-version-3/
ERROR : root : Config file version mismatch: z:\machine_files\your_machine\
config\config.yaml
Traceback (most recent call last):
  File "mpf.py", line 119, in main
    machine = MachineController(options_dict)
  File "Z:\mpf\mpf\system\machine.py", line 85, in __init__
    self._load_machine_config()
  File "Z:\mpf\mpf\system\machine.py", line 221, in _load_machine_config
    Config.load_config_file(config_file)
  File "Z:\mpf\mpf\system\config.py", line 61, in load_config_file
    config = FileManager.load(filename, verify_version)
  File "Z:\mpf\mpf\system\file_manager.py", line 130, in load
    halt_on_error)
  File "Z:\mpf\mpf\file_interfaces\yaml_interface.py", line 51, in load
    format(filename))
Exception: Config file version mismatch: z:\machine_files\your_machine\
config\config.yaml
C:\pinball\mpf>

```

Of course all this does is cause MPF to crash saying that there's a "Config file version mismatch," but hey, MPF ran and found your machine folder! (The errors are saying that your *config.yaml* is not a valid MPF configuration file, which makes sense, because it's totally empty. We'll add stuff to it in the next step.)

MPF will automatically look for a file called */config/config.yaml* in the folder you pass in the command line. So running `python mpf.py c:\pinball\your_machine` means MPF is looking for a file at `c:\pinball\your_machine\config\config.yaml`.

What if it doesn't work?

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

If you get an `IOError: Couldn't find file`, that means MPF couldn't find your *config.yaml* file, like this:

```

C:\pinball\mpf>python mpf.py c:\pinball\wrong_folder
INFO : Machine : Mission Pinball Framework v0.21.0
INFO : Machine : Machine config file #1: c:\pinball\wrong_folder\config\
config
ERROR : root : Could not find file c:\pinball\wrong_folder\config\config
Traceback (most recent call last):
  File "mpf.py", line 119, in main
    machine = MachineController(options_dict)
  File "c:\pinball\mpf\system\machine.py", line 85, in __init__
    self._load_machine_config()
  File "c:\pinball\mpf\system\machine.py", line 221, in _load_machine_config

```

```

Config.load_config_file(config_file)
File "c:\pinball\mpf\system\config.py", line 61, in load_config_file
  config = FileManager.load(filename, verify_version, halt_on_error)
File "c:\pinball\mpf\system\file_manager.py", line 140, in load
  raise IOError("Could not find file {}".format(filename))
IOError: Could not find file c:\pinball\wrong_folder\config\config

C:\pinball\mpf>

```

In that case make sure you typed the proper path to the machine root in the command line, and make sure your config file is in the `<machine_root>/config/config.yaml`.

If you get some type of error about **python not being a valid command**, that means you either don't have Python installed or it's not in your path. You can read the documentation on python.org about [Python setup and usage](#) to get that cleared up.

If you get an error about **mpf.py not found**, make sure you're running Python from the root folder you unzipped the MPF package to. (There should be a file called *mpf.py* in that folder.) You'll notice that the MPF download package has two *mpf* folders. There's the overall package name called *mpf*, and then within there you'll find additional folders called *machine_files*, *mpf*, *tests*, and *tools*. (So the parent *mpf* folder is your main MPF folder, and the child *mpf* folder holds the actual MPF python files.) You want to be running this command from within the parent *mpf* folder.

Once you've confirmed that MPF is trying to load your config file (with the error about the config version mismatch), then you can move on to the next step of actually adding things into your config!

Step 4. Get flipping!

There's something exciting about seeing the first flips of your own pinball machine (whether it's a machine you built from scratch or an existing machine you're writing custom code for), so in this step we're going to focus on getting your machine flipping as fast as possible.

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 into some flipper devices.

So go ahead and open your newly-created `config.yaml` file in the `/your_machine/config` folder. This file should be totally blank. You can edit it in whatever text editor you want. [Atom](#) or [Sublime](#) are nice free ones that understand the formatting required for YAML files. [PyCharm](#) is a fully featured Python IDE & debugger which is what we use to develop MPF. (And thanks to [JetBrains](#), the makers of PyCharm who make their Professional version available for free developers of Open Source projects with it.)

(A) Add `#config_version=3` 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.

```
#config_version=3
```

The reason we do this is because one of the challenges we had with all the frequent updates to MPF is that sometimes new versions of MPF change certain settings in the config files. So we need a way to track which set of config file settings a particular YAML file uses. That way when MPF loads the config, it can make sure the actual contents of the config file match up with what MPF is expecting.

That said, not every new version of MPF has changes to the YAML file, so that's why the YAML file `config_version` and the MPF version aren't the same.

Adding versioning to YAML files also means it's easy for people to see what they have to change in their existing YAML files. For example, if a future version requires YAML files with `config_version 4` and your files are `config_version 3`, we can simply create a page explaining what needs to change between version 3 and 4. (We also have a [config file migration tool](#) you can use to automatically update your config files if we change it in the future.)

The current version of the config files is 3 which is what's used with MPF 0.20.0 and newer, so that's what we're adding here.

(B) Add your flipper buttons

On the next line after `#config_version=3`, 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:

```
switches:
  s_left_flipper:
    number:
  s_right_flipper:
    number:
```

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 popup 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`.)

Also, note that the settings in MPF config files are case-insensitive. This is a change we made with MPF 0.17 because lots of people were running into issues with things not working because they didn't enter the name of a setting exactly right. (For example they'd type the section name as `switches:` instead of `Switches:`. So what happens internally is MPF converts everything to lowercase. (Well, not everything. Certain things like labels and text strings and stuff will be in whatever case you enter them as. But in general stuff is case insensitive.) The reason we mention this is because you can *not* have two things configured with the same name that only vary based on case sensitivity. For example, the switch names `s_lane_trEk` and `s_lane_treK` are not allowed since they'd both be converted internally to `s_lane_trek`.

Anyway, let's look at a few notes about YAML (which is the format of the file we're creating here):

First, **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 doesn't strictly 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.

Next you have to enter the hardware numbers for your two switches. (This is what tells MPF which hardware switch numbers map to these switch names you're creating.) If you have a physical pinball controller attached, you need to enter the numbers that correspond to your flipper button switches on your hardware. *Note that you must have no space between "number" and the colon, and you must have a space between the colon and your switch number.*

The exact format will depend on whether you're using a FAST, P-ROC, or P3-ROC pinball controller and whether these are matrix or direct switches. Refer to the [switches: section of our Configuration File Reference](#) for the details of exactly what number you need to use for these. (By the way there are many more settings and options for switches, but for now we just need the numbers.)

If you don't have physical hardware right now, then you can just make up whatever numbers you want. Keep it simple, like 0 and 1. So your file now should look something like this:

```
switches:
  s_left_flipper:
    number: 0
  s_right_flipper:
    number: 1
```

Make sure (now and forever) that you've formatted the YAML file properly, like this:

```
Switches:
  leftFlipperButton:
    number: 0
  rightFlipperButton:
    number: 1
```

The diagram illustrates the correct formatting for the switches section. Red arrows point to the following details:

- No space:** Points to the colon at the end of the key name (e.g., `leftFlipperButton:`).
- Space:** Points to the space character before the value (e.g., `number: 0`).
- spaces (not tabs):** Points to the spaces used for indentation of the values.

(C) 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 the [Flippers section of the MPE 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.

If you have dual-wound coils, your `coils:` section of the documentation should look like this:

```
coils:
  c_flipper_left_main:
    number: 0
  c_flipper_left_hold:
    number: 1
  c_flipper_right_main:
    number: 2
```



```
c_flipper_right_hold:
    number: 3
```

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 P-ROC or FAST and whether you're using their driver boards and standard Williams boards. (Refer to the [coils: section of our configuration file reference](#) for more details about coil numbers for your specific hardware.)

Also, again, if you're only using virtual hardware at this point, you can enter whatever you want for your numbers. (It's okay if some of your flipper coil numbers are the same as your switch numbers, since MPF keeps track of coil numbers and switch numbers separately.)

(D) Add your flipper "devices"

Okay, so now 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 [devices](#), which, broadly-speaking, and be broken down into two classes:

- There are low level raw hardware devices which you actually connect to your pinball controller. These are coils, switches, matrix lights, RGB LEDs, flashers, motors, and servos.
- There are higher-level logical devices which are familiar pinball devices, like flippers, pop bumpers, troughs, drop targets, shots, etc. All these higher-level devices are logical groupings of the lower level devices: a flipper is *this* switch plus *that* coil, a drop target is *this* switch and *that* knockdown coil and *this* reset coil, etc.

So getting back to the flippers, you create your logical flipper devices by adding a `flippers:` section to your config file, and then specifying the switch and coil(s) for each flipper. 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
```

What if your flippers coils only have one winding?

The example in the tutorial above uses dual-wound flipper coils where MPF literally sees each flipper coil as two separate coils (with two separate names and two separate drivers). When you push the flipper button, MPF energizes both coils initially, but cuts the power to the main coil after a few milliseconds so only the lower power hold coil remains active. This prevents the flipper coil from burning up. As an alternative, some flippers just use normal (single winding) coils and then the hardware controller controls the flow of electricity through it to prevent it from burning up. In that case the hardware will send an initial constant pulse for a few milliseconds to give the flipper its strong initial pulse, and then it will flip the current on & off really fast (really fast, like hundreds of times per second) to keep the flipper in the 'up' position without overheating it.

If you have single-wound flipper coils (or if you have traditional dual-wound coils but you don't want to waste two drivers per flipper and you just want to use a single winding), make sure you've read [our documentation on flipper devices](#) for all the details about how that works.

If you'd like to use single-wound flipper coils, you need to do two things in your config file:

- First, you can remove the `hold_coil:` entries from your two flippers since you don't have hold coils.
- Second, you need to add a `hold_power:` entry to each of your two coils in the `coils:` section of your config file. This is how you tell MPF what timing it should use to quickly pulse the current to that coil when its being held on.

Here's an example of what the `coils:` and `flippers:` sections of your config file would look like if you're using **single wound coils**. (The `switches:` section would be the same in both cases):

```
coils: #P-ROC / P3-ROC only
  c_flipper_left_main:
    number: 0
    pulse_ms: 20
    hold_power: 2
  c_flipper_right_main:
    number: 2
    pulse_ms: 20
    hold_power: 2
```

```
flippers:
  left_flipper:
    main_coil: c_flipper_left_main
```

```

    activation_switch: s_left_flipper
right_flipper:
    main_coil: c_flipper_right_main
    activation_switch: s_right_flipper

```

Note that we used a values of 2 for the *hold_power*. The *hold_power* setting is a whole number from 0-8 which represent a percentage of power that's applied when that coil is held on. (0 = 0%, 4=50%, 8=100%, etc.)

At this point we have no idea if `hold_power: 2` is the correct setting or not. We can fine-tune that later. (And again, *hold_power* is only used with single-wound coils. Dual-wound coils fire both windows at full power all the time.)

(E) 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 MPF project folder, and run this:

```
python mpf.py your_machine -v -b
```

Notice that we have the familiar `-v` option to write a verbose log file, but we also have a new `-b` option. The `-b` option (which means "no BCP") tells MPF that it should not try to connect to a media controller to run a display. We're adding that option for now because we haven't configured a media controller yet as that's something we'll get to in a few more steps.

When you run this, you'll see some things loading and a message that your attract mode has started. If you see this, then congrats! Your config file is okay and your game is running. It will kind of look like it's hung, but it's not—it's actually running.

```

C:\pinball\mpf>python mpf.py c:\pinball\your_machine -v -b
INFO : Machine : Mission Pinball Framework v0.21.0
INFO : Machine : Machine config file #1: C:\pinball\your_machine\config\
step4
INFO : Machine : Loading system modules...
INFO : DeviceManager : Loading devices...
INFO : Machine : Loading plugins...
INFO : SwitchController : Dumping current active switches
INFO : Mode.attract : Mode Starting. Priority: 10
INFO : Mode Controller : +===== ACTIVE MODES =====+
INFO : Mode Controller : | attract : 10 |
INFO : Mode Controller : +-----+

```

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

When you stop it, you'll see a few more lines appear on the console which have information about the "target" and "actual" game loop rates. By default MPF is configured to run at 30

loops (or "ticks") per second, and hopefully you should see your actual loop rate somewhere in that neighborhood, like this:

```
INFO : Machine : Target MPF loop rate: 30 Hz
INFO : Machine : Actual MPF loop rate: 30.0 Hz
INFO : Machine : Hardware loop rate: 63.98 Hz
INFO : root : MPF run loop ended.
```

Potential errors and how to fix them

If your game ran fine, then you can skip down to Section (F) below. If something didn't work then there are a few things to try depending on what your error was.

If the last line in your console output is *AssertionError: Device 'x' does not have a valid config*, that means that device entry in your config file isn't right. Probably this is caused by incorrect indentation errors.

If the last line in your console output is *AssertionError: Device 'x' has an empty config*, that means the device entry in your config file doesn't have any sub-sections under it (like you're missing the *number:* setting, for example).

If the last line in your console output is *CRITICAL: YAML File Interface: Error found in config file 'x'. Line x, Position x*, that means you have a formatting problem with your YAML file. The line and position numbers will get you close to finding where the problem is, but they're never exactly right because most formatting errors in YAML files actually affect how the YAML processor sees the file, so it's reporting what it saw based on your error.

The big "gotchas" with YAML files are:

- 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=3` as the first line in your file.

(F) 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 basic config file 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 once the initial initialization phase is done, rather than them waiting for a ball to start.)

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
```

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 [Step 6: Add a display](#).

(G) Configure MPF to use your FAST, P-ROC, or P3-ROC 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, or a P3-ROC Pinball 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:`. Some hardware platforms require additional settings for ports and stuff too. Let's look at the specifics depending on your hardware platform.

If you're using a P-ROC:

If you're using MPF with a P-ROC, simply add the entry `platform: p_roc` in your `hardware:` section. (This is for a P-ROC only. Instructions for the P3-ROC are in the next section.)

For the driver boards, you have the option to use either the P-ROC driver boards (like the PD-16), or existing WPC or Stern driver boards (like if you're plugging your P-ROC into an existing machine). For this tutorial you can use either.

If you want to use P-ROC driver boards, you add an entry **driverboards: pdb**. If you want to use WPC driver boards with an existing machine, you add the entry **driverboards: wpc**. And if you want to use MPF on a Stern S.A.M. machine, you add the entry **driverboards: sternSAM**.

So the `hardware:` section you add to your config file will look like this:

```
hardware:
  platform: p_roc
  driverboards: pdb
```

Or like this:

```
hardware:
  platform: p_roc
  driverboards: wpc
```

Or like this:

```
hardware:
  platform: p_roc
  driverboards: sternSAM
```

If you're using a P3-ROC:

For the P3-ROC, everything is the same as the P-ROC above, except you use `p3_roc` for your platform. (And of course you'd use `pdb` for the driver boards since the P3-ROC doesn't support other types.)

If you're using a FAST Pinball controller:

To use MPF with a FAST Pinball controller, you add an entry **platform: fast** to the **hardware:** section of your config file. FAST Pinball controllers also have the option of either working with either WPC or FAST driver boards, so you need to add the configuration entry **driverboards: fast** or **driverboards: wpc**, depending on what you have.

When using a FAST Controller, the `hardware:` section of your config file will either look like this:

```
hardware:
  platform: fast
  driverboards: fast
```

Or like this:

```
hardware:
  platform: fast
  driverboards: wpc
```

FAST Controllers also require that MPF is configured for the serial port and baud rate. This is done via a section in the config file called `fast:` which will look like this:

```
fast:
  ports: com3, com4, com5
```

If you're using a FAST controller, the above section will be 100% accurate for you except for the names of the ports. You'll have to change those to the actual port names that the FAST controller uses on your system. If you don't know the name of the ports, read the `ports:` section of [our configuration file reference for FAST](#) for instructions on how to figure out which port it's using. (Basically just plug in your FAST controller and look for the four COM ports that pop up, and then add the first three for a FAST Core or WPC controller, and the middle two for a FAST Core controller.)

If you're using a FAST controller, you'll end up adding both `hardware:` and `fast:` sections to your config file, like this:

```
hardware:
  platform: fast
  driverboards: wpc
fast:
  ports: COM3, COM4, COM5
```

(H) Make sure you have your hardware drivers installed

Just like any peripheral you plug into a computer, you need install drivers and the interface software before your computer can talk to a hardware pinball controller. You should have gotten the drivers installed when you originally setup MPF, but if you started with no hardware and you're adding it now, go back to the [installation documentation](#) and read the section for your platform to get the drivers installed.

(I) 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 P-ROC plugged into an existing *Demolition Man* machine.)

```
#config_version=3

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
  c_flipper_right_main:
    number: FLRM
  c_flipper_right_hold:
    number: FLRH

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.

(J) 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:

```
python mpf.py your_machine -v -b
```

Watch the console log for the following entry:


```
INFO : Mode Controller : +===== ACTIVE MODES =====+
INFO : Mode Controller : | attract : 10 |
INFO : Mode Controller : +-----+
```

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*, but we're using *Demolition Man* as our sample machine in this tutorial. That's okay since everything is basically the same. The only difference is the actual outputs that are configured for the switch and coil connections.)

<https://www.youtube.com/watch?v=SkxZxkHHmXw>

What if it doesn't work?

If your game doesn't flip while you're running this code, 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` command line option, since that tells MPF to run in 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 "O/4" or "SD12". Again look at our configuration file reference for both [coils](#) and [switches](#) for explanations of all the different options for the `number:` setting depending on what type of hardware, driver boards, and connections you have in your physical machine.

- Make sure you added `enable_events: machine_reset_phase_3` to each of your flipper configurations.
- Make sure you don't have a typo in your config file. (For example, `flipper:` instead of `flippers:`, etc.) Search through the log file (in verbose mode) for your flipper names to make sure they're being created and activated.
- 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 the game software 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 a P-ROC and you get an error `ImportError: No module named pinproc`, that means you either (1) don't have the P-ROC drivers installed, or (2) you have multiple instances of Python on your computer and you installed the drivers into one and you're running MPF from the other. [Post to the forum](#) and we can help sort it out.
- If you get an with the name of something not being valid from your config file, that probably means that you mistyped something. For example if you mistype one of your switch or coil names in your `flippers:` section, then there will be an error when the game tries to enable the flippers since one of those names doesn't point back to a real switch or coil in your machine.

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, and 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 our [MPF users forum](#). We'll incorporate your issues into this tutorial to make it easier for everyone in the future!

Step 5. Adjust your flipper power

As we mentioned already, MPF uses a fairly 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.)

(A) Adjust coil pulse times

Modern pinball controllers like the P-ROC and FAST controllers 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 the default pulse time that's specified in the `<your mpf root folder>/mpf/mpfconfig.yaml` defaults file which is set to 10ms out of the box.

(The `mpfconfig.yaml` file is full of all the default settings that MPF uses for anything you don't specify in your own machine config file.)

So in the last few steps, 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, 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 `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
    pulse_ms: 20
  c_flipper_left_hold:
    number: 01
  c_flipper_right_main:
    number: 02
    pulse_ms: 20
  c_flipper_right_hold:
    number: 03
```

Notice that we only added `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 `pulse_ms:` values up or down until they your flippers feel right. In the future we'll create a coil test mode 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 `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.

(B) Adjusting coil "hold" strength

If you're using single-wound flipper coils, you should also take a look at the `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 `hold_power:` values should be. Really you can just start with a value of 1 or 2 and then keep increasing it (whole numbers only) until your flipper holds are strong enough not to break their hold when a ball hits them.

Each hardware platform has additional options for fine-tuning the hold power if you find the values of 1-8 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.

(C) 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 MPF package at `<your_mpf_root>/machine_files/tutorial/config/step5.yaml`. And remember if you're using physical hardware, your coil and switch numbers will be different than the ones in the sample file, since you'll need to configure them based on your driver boards' actual inputs and outputs.

You can run the sample config file directly (in software-only virtual mode) via the following command:

```
python mpf.py tutorial -c step5 -b -v
```

Step 6. Add a display

The next step is to add an on-screen display to your game (and, optionally, to configure the DMD if you're working with a physical machine that includes a DMD). To do this, we're going to start using a new component of MPF called the "Media Controller."

(A) Understand what the Media Controller is

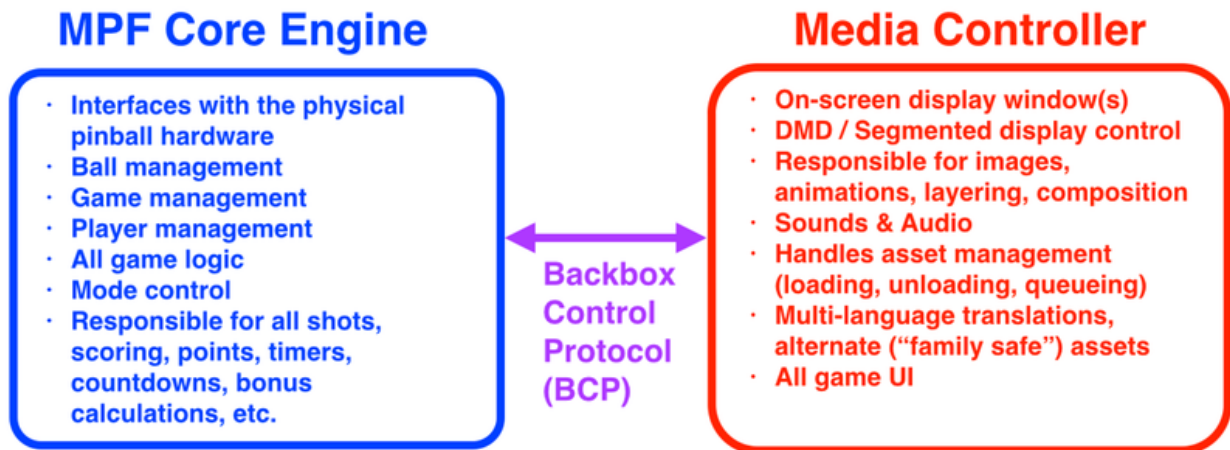
One of the most important things to understand about the architecture of MPF is that the core MPF game engine is completely separate from the process that controls graphics and audio. We call the thing that handles graphics and audio a "media controller."

So far we've only been dealing with the MPF core engine (which is launched via the Python file `mpf.py`). But when you run a complete pinball machine with MPF, you actually run two processes—the MPF core and a separate standalone media controller. The two processes communicate via a TCP socket via a protocol called "BCP" (which stands for "Backbox Control Protocol").

The MPF package you downloaded from GitHub includes a Media Controller (which we call the "MPF Media Controller"), and you launch it via the Python file `mc.py`.) The BCP protocol which is used to communicate via the MPF core engine and the MPF media controller is an open protocol, meaning that anyone is welcome to build either a pinball controller or a media controller that uses it. So it's possible that you might choose to use MPF as your core pinball engine while using some other product as your media controller. (More on that in a minute.)

Here's a diagram which shows what's handled by the MPF core engine and what's handled by the media controller. (The details of this are not important right now. But take a minute to

read through the bullets of each piece so you can start to understand how the roles break down.)



At this point you are probably wondering why we broke MPF into two pieces. After all, doesn't this seem more complex than having everything in the same place?

The answer is yes, this is more complex, but there are two important reasons why we did this:

- Breaking MPF into two processes means that we can get much better performance on multi-core processors.
- Separating the media controller into its own standalone component means that you can swap out MPF's built-in media controller for a more advanced one if you have advanced needs.

Let's look at each of these more in depth, starting with the performance thing.

Running the MPF core engine and the media controller as two separate processes means they can each run on their own core on a multi-core system. This doesn't really matter if you're running MPF on a fast desktop or laptop computer with a full-speed Intel processor. But many people don't want to put full computers in their pinball machines, rather, they want to use small single-board computers like the Beaglebone Black, Raspberry Pi, ODOIRD, CubieBoard, etc. Our experience running MPF on these tiny systems has been mixed. We found that they were work fine for basic things, but when once you started adding in lots of graphics, videos, animations, and multiple tracks of audio, things start to slow down.

What made it worse with earlier versions of MPF is that while many of these small computers are moving to multicore processors, MPF was a single process. So we'd have MPF running on this little quad-core computer with MPF pegging one core to the max while the other cores were sitting there doing nothing. Breaking MPF into two processes means that we can run each process on a separate core, essentially doubling the performance we can get on one of these tiny computers. (Plus as an added bonus, since the video and audio

processing is on a separate core from the main MPF engine, if we do run into any performance issues playing video or anything then it won't actually slow down the core game engine.)

The second advantage to having a standalone media controller is that you can replace MPF's built-in media controller with another one that's more advanced. This is something that will become more important in the future as more pinball machines adopt full-screen LCD-based backboxes. This addresses a challenge we found in MPF once we started building the DMD and LCD window support. We literally had to write every single display function for MPF ourselves. We had to handle building elements (text and images and animations and video), as well as positioning those elements, handling layering and z-ordering, transparencies, blends, movement, scaling, etc. Over time it seemed like we were spending more time writing a graphical video game creation engine instead of pinball machine software!

The problem is that graphical game engines already exist in the world! So why should we, a bunch of pinheads, try to rewrite what essentially already existed? Of course the existing game engines are geared towards video games—they don't have the ability to control pinball hardware, and they're not nearly as easy to use as MPF is.

Around the same time that we were thinking about ways to address this problem, some users of MPF posted in the forum asking about whether it was possible to integrate MPF with a "real" gaming engine such as the [Unreal Engine](#) or [Unity 3D](#). After several weeks of conversations in the forum, we decided to take the plunge and to break MPF into the two pieces while also defining a standard for communication (the BCP protocol) so that MPF could be used with any media controller.

Again, the MPF package you download from GitHub has a media controller bundled in with it. (It's what we call the MPF Media Controller.) It can handle multiple display elements, animations, videos, and multiple tracks of audio. It's more than capable of powering a DMD, a color DMD, and even the full LCD display of a machine like *The Wizard of Oz*. (And the MPF Media Controller can be configured with the same YAML-based configuration files as the MPF core engine.)

But if you want to do something very advanced with your backbox display—perhaps something like 3D graphics or surround sound—then you can still use MPF for your core and then use another media controller in place of MPF's basic media controller. There's a group of folks working on an open source advanced media controller based on Unity 5 (which is also open source). We have some more information about it [here](#) and a [forum](#) dedicated to it, but it's very early and not ready. Stay tuned though if you're interested!

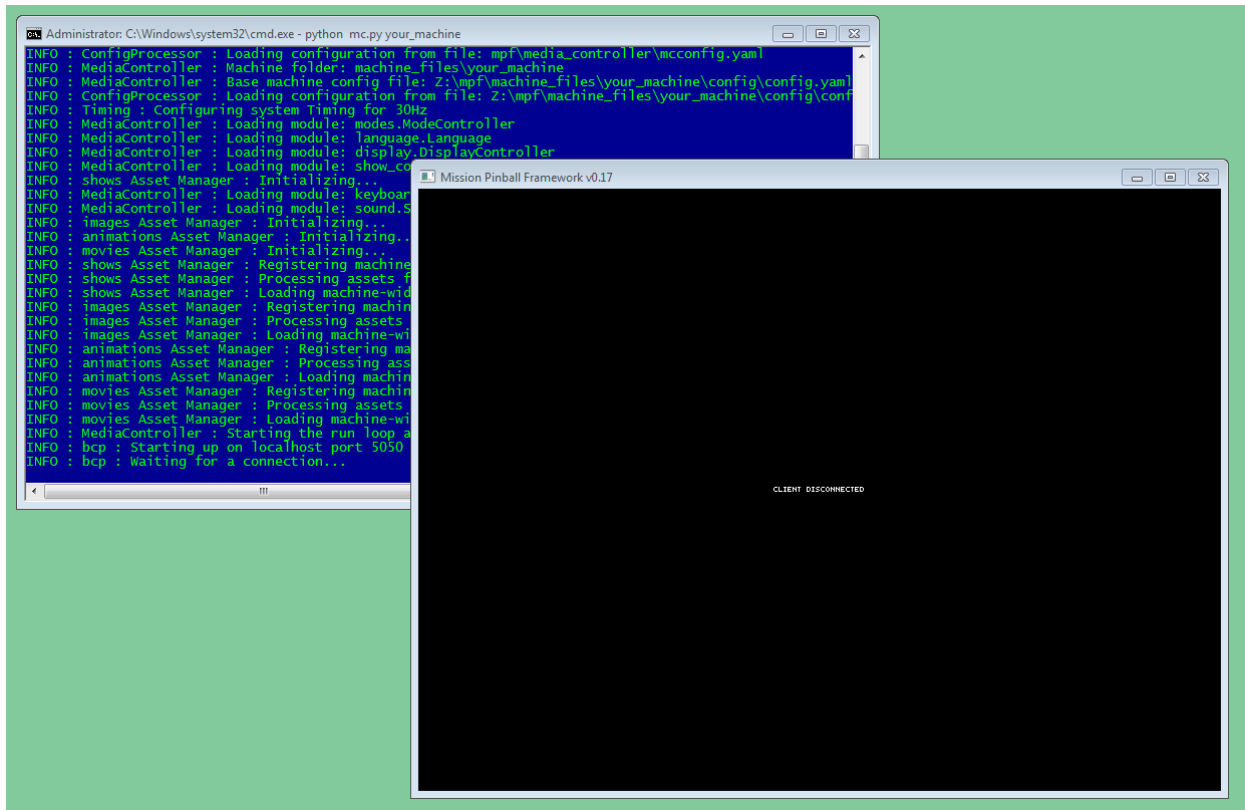
(B) Run the media controller to see how it works

As we mentioned already, MPF's built-in media controller is launched via a Python script `mc.py`. It takes command line options similar to `mpf.py`, including the path to your machine folder and `-v` to write verbose logging to the log file.

Go ahead and run the Media Controller by pointing it to your machine's config, like this:

```
python mc.py your_machine
```

When you do this, you should see an 800x600 popup window with teeny-tiny text that reads "WAITING FOR CLIENT CONNECTION...", similar to this: (This example is from a prior version of MPF so the text is not the same as the current version, but you get the idea.)



At this point you can click your mouse back into the command window where you ran the Python command to launch the media controller and press `CTRL+C` to kill it. Before we work on connecting the media controller to the MPF core engine, let's spend a few minutes adding some elements to our media controller pop-up window so we can actually read what's going on.

(C) Add a DMD

Note: This tutorial shows how to setup a single-color traditional DMD like what you'd find in a 1990s Williams or current Stern machine. If you want to use a Color DMD (either rendered on an LCD screen or a physical color LED-based DMD), we have [How To guides for those](#), but it's probably easiest for now just to follow along here with the mono DMD and then switch the settings to color once you finish the tutorial.)

The first thing we're going to do is to add a virtual DMD to our popup window. If you have a physical machine with a physical DMD, that's great, because we'll also be able to control the physical DMD with this step. But if you don't have a physical machine, or you have a physical machine but it doesn't have a DMD, that's fine too. Let's add the DMD anyway since it's really easy, and then you can remove it later.

The first thing to do is to tell MPF that we want to use a DMD. To do this, add a `dmd:` section to your configuration file, like this:

```
dmd:
  physical: no
  width: 128
  height: 32
```

The settings for the this section are detailed in the [dmd: section](#) of the config file reference, so you can read about them there if you want to know about what these settings mean or you want to see what other options are available.

The only thing we'll point out here is the `physical:` setting which specifies whether you have a physical DMD connected or not. (This is referring to the actual physical DMD with the 14-pin ribbon cable.) Set it to `yes` (or `true`) if you do, and `no` (or `false`) if not. It's perfectly fine to use `physical: no` as a permanent solution if you want a virtual DMD in your window but you have no intentions of ever hooking on up.

At this point if you have a physical machine with a physical DMD, you can rerun the MPF core engine and you should see the physical DMD go blank:

```
python mpf.py your_machine
```

Your flippers should still work (just like they did in the last step), except this time the P-ROC or FAST logo that's on the screen when you power on your hardware should go blank. (In order for this to work then you need to use `physical: yes` in your config. But you won't see a virtual DMD in the on screen window since we haven't added that yet, so let's do that now.)

(D) Add a "virtual" DMD element to your on-screen window

Now that you've defined a DMD device (even if it's not a physical DMD), you can add the Virtual DMD [display element](#) to your on-screen window.

Why is this a separate step? Because with MPF, the on-screen window is just another display you can use for anything you want. While some people might only want to use it to display a virtual DMD, you can also use the window to show light, switch, and coil states, animation effects, troubleshooting information, multi-player scores, or just about anything else you want. So MPF can't automatically assume that you want to show a virtual DMD in your

window just because you have a physical DMD, so that's why you have to manually set it up here.

All the on-screen window settings are configured in the `window:` section of your config file. Since we haven't specified any other window settings so far in this tutorial, go ahead and add a `window:` entry now.

Then you configure all the display elements (the things that show up on an MPF display) in the `elements:` subsection of the `window:` section, so add that now too.

Finally you can add a display element for the DMD. Your `window:` section should now look like this:

```

window:
  elements:
    - type: virtualdmd
      width: 512
      height: 128
      h_pos: center
      v_pos: center
      pixel_color: ff6600
      dark_color: 220000
      pixel_spacing: 1

```

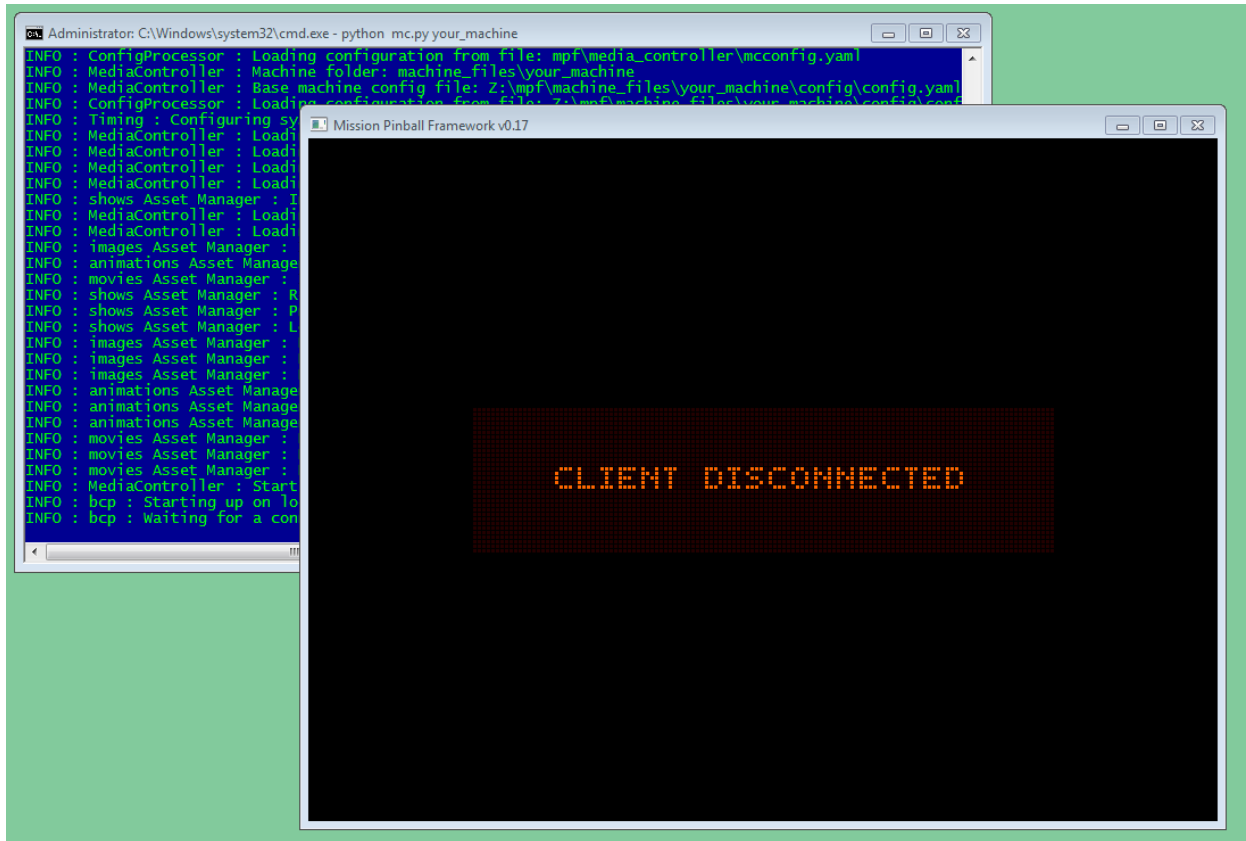
Like everything else you've been adding to your config file, the `window:` entry should be at the beginning of a line with no spaces in front of it, and then each child element should be indented by four spaces. (So `elements:` is indented by four spaces, `type:`, `width:`, etc. all have eight spaces.) Note that there's a dash (a minus sign) plus a space in front of the `type:` setting. This is a YAML formatting thing and how MPF knows where one display element ends and the next starts.

Before we explain what all these settings mean, let's run it again real quick to make sure it works.

To do this, run the *media controller* (since that's what controls the on-screen windows, not the MPF core), like this:

```
python mc.py your_machine
```

You should see something like this:



Pretty cool! Again the "CLIENT DISCONNECTED" message is there because the media controller is not connected to the MPF core engine (which makes sense because MPF isn't running). (And again, the current version of MPF will actually say "WAITING FOR CLIENT CONNECTION...")

So back to what all those virtual DMD settings mean. You can read all about them in the [Virtual DMD display element in the MPF documentation](#), but here are the basics:

The entries for **width:** and **height:** are where you specify the size (in pixels) of your on-screen virtual DMD. Notice that these are different values than the `width:` and `height:` you configured in the `dmd:` section of your config. The settings in the `dmd:` section told MPF what size of the DMD was in terms of display pixels. The settings in the `window:` section control how big you'd like to draw the DMD in your on-screen window. The on-screen width and height can be anything you want, though you should make sure you keep them at the same aspect ratio as your physical DMD or else the contents of the window will be stretched weird. Also it's probably best if they're an exact multiple of your DMD pixels.

The **h_pos:** and **v_pos:** settings specify where within your window the virtual DMD element will be drawn, with each representing the upper and left sides of the window. You can use the value `center` for each one to center it, or integer values from the upper left corner of your main window. So `h_pos: 0` and `v_pos: 0` will put your DMD element in the upper left corner of the on screen window, `h_pos: center` and `v_pos: 450` will center it in the lower

portion of your window. etc. You can read more about positioning and placing elements [here](#).

The **pixel_color**: setting controls what color a fully "on" pixel will be in your on-screen window. The example of `ff6600` is an orange-ish color which looks pretty close to classic Williams DMDs. You can change it to red `ff0000` or green `00ff00` or whatever you want. (This color specification, like all colors in the config file, is a 6-character HTML-style color representation. `ff` is full on (255), and `00` is full off (0). White would be `ffffff`, red would be `ff0000`, yellow is `ffff00`, etc. There's an online color picker [here](#). Note that you do *not* add a preceding hash sign (#) to your color entries in these config files since the hash sign is used to comment out a line in YAML.

By the way, you might be wondering why the pixel color setting is a property of the virtual DMD window display element, rather than a property of the actual DMD itself you just set up? That's because the physical DMD's color is dictated by the actual DMD you have—MPF doesn't know (or care) what color it is, rather, it just knows that it needs to turn pixels on or off. But when it comes to displaying a virtual DMD in an on-screen window, you can make it whatever you want. (And of course the on-screen pixel color setting doesn't have to match what your actual physical DMD color is.)

The **dark_color**: setting controls what color is used when the pixels are "off" (fully dark). A value of `220000` (dark red/brown) gives it a nice DMD-ish look.

MPF will automatically calculate all the in-between color shades which range from `off_color` to `pixel_color` based on the number of shades you specified in your DMD configuration back in Step (1).

Finally you can set a value for the **pixel_spacing**: which affects how much visual space there is between pixels in the on screen DMD. The exact number you pick is a matter of personal preference and how big your on screen DMD is. We have some examples on our [Virtual DMD documentation page](#).

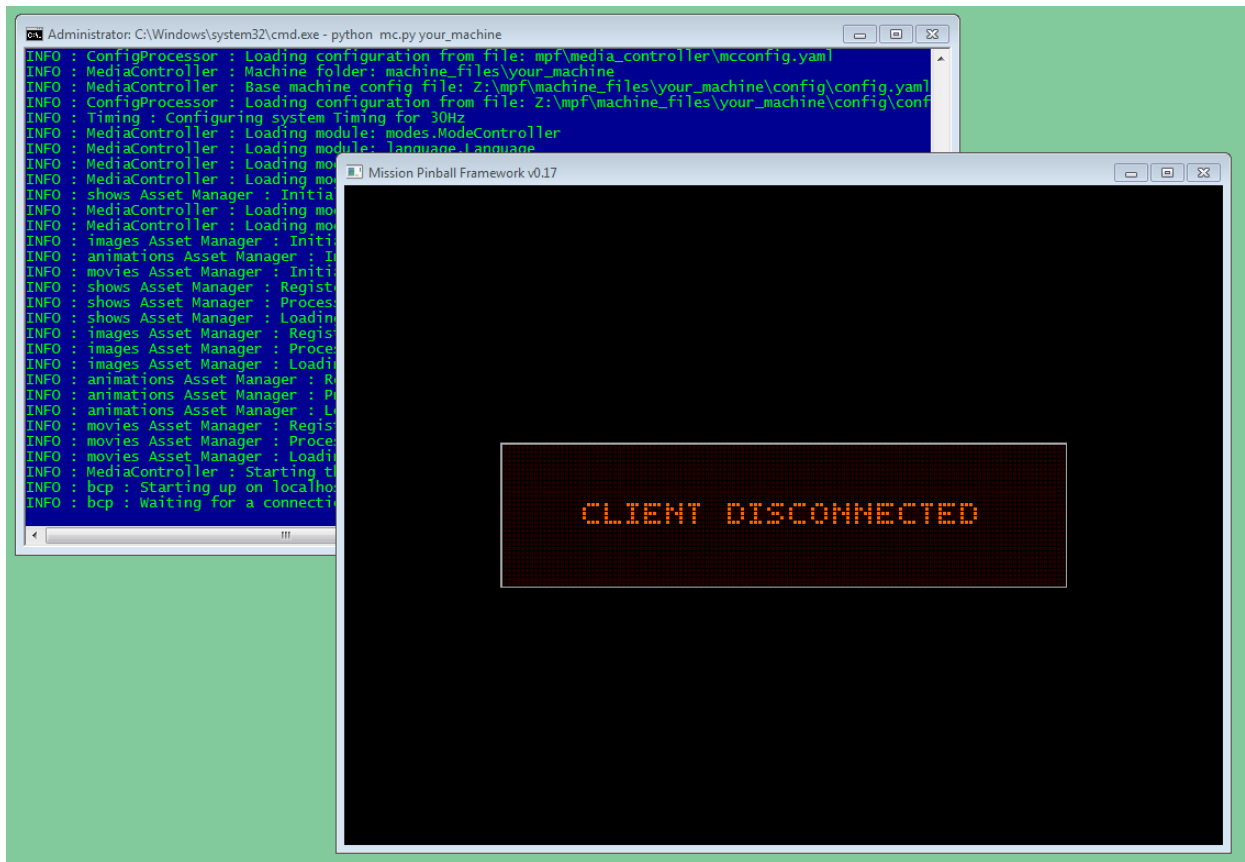
(E) Draw a box around the DMD

At this point we have a blank DMD on the screen, but it's kind of hard to see. So let's draw a box around it. We'll do this by adding another display element to the **window**: section of your config. This time we'll use a display element called "shape," and we can add it like this:

```
- type: shape
  shape: box
  width: 516
  height: 132
  color: aaaaaa
  thickness: 2
```

Notice that we didn't add `v_pos:` and `h_pos:` entries. That's because MPF uses "center" as the default for both, so we don't have to add them here. (Technically we didn't have to add them in the previous step either, but we just wanted to include them there so you could learn about them.)

Now launch the media controller again (via `python mc.py c:\your_machine`) and your window should look like this: (Be sure to save your `config.yaml` file first!)



(F) Run your media controller and the MPF core at the same time

Ok, so now we're able to run the media controller and to get a basic DMD to show up in the on-screen window. The next thing to do is to run them both at the same time and to make sure that they're able to talk to each other.

To do this, first launch the media controller as you did in the previous steps.

```
python mc.py c:\pinball\your_machine -v
```

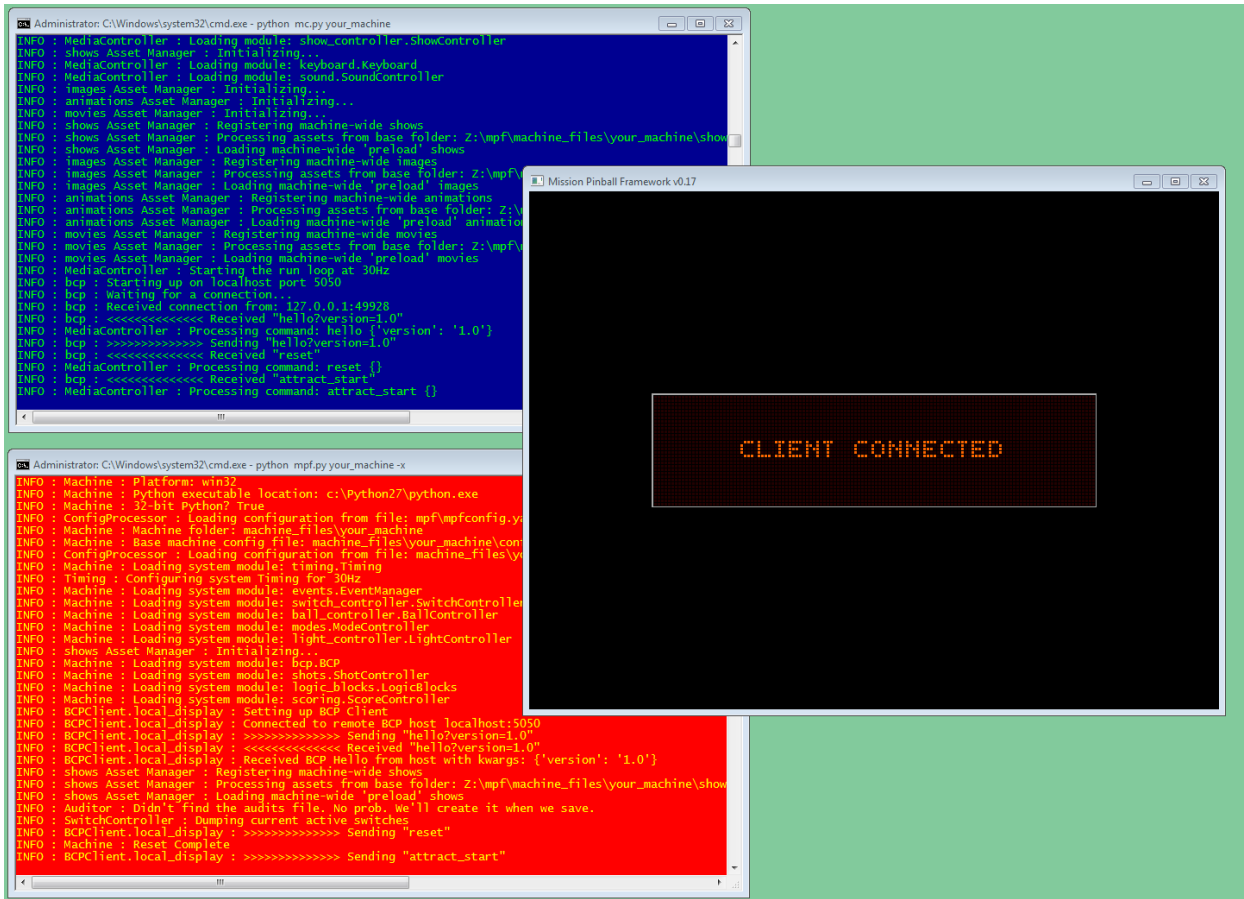
Then open up a second command window so you can launch MPF while the media controller is still running. (You have to open a new window because the media controller "takes over"

the first window which means you can't launch anything else from it until the media controller stops.)

So instead, open up a second window and launch the MPF core (using `mpf.py` instead of `mc.py`), except this time do *not* use the `-b` option since we want the MPF core engine to connect to the Media Controller::

```
python mpf.py c:\pinball\your_machine -v
```

When you initially launch the media controller, it should open the window with your virtual DMD (just like last time) with the "WAITING FOR CLIENT CONNECTION..." message on the DMD. Then once you launch the MPF core engine, it will automatically connect to the media controller and you should see the message "CLIENT CONNECTED" flash briefly in the DMD. The image below shows what this should look like. You might miss the CLIENT CONNECTED message because once the attract mode starts, the DMD will go blank. (In this screenshot we set the color of the console window that launches the MPF core to be red just so we can tell the two apart.)



If you have a physical machine with a DMD, you should see the "CLIENT CONNECTED" message flash on the physical DMD. (If you don't see this message on the DMD but your

physical machine's flippers work, make sure you have `physical: yes` in your `dmd:` section.)



(G) Using the batch file to launch MPF & the Media Controller at the same time

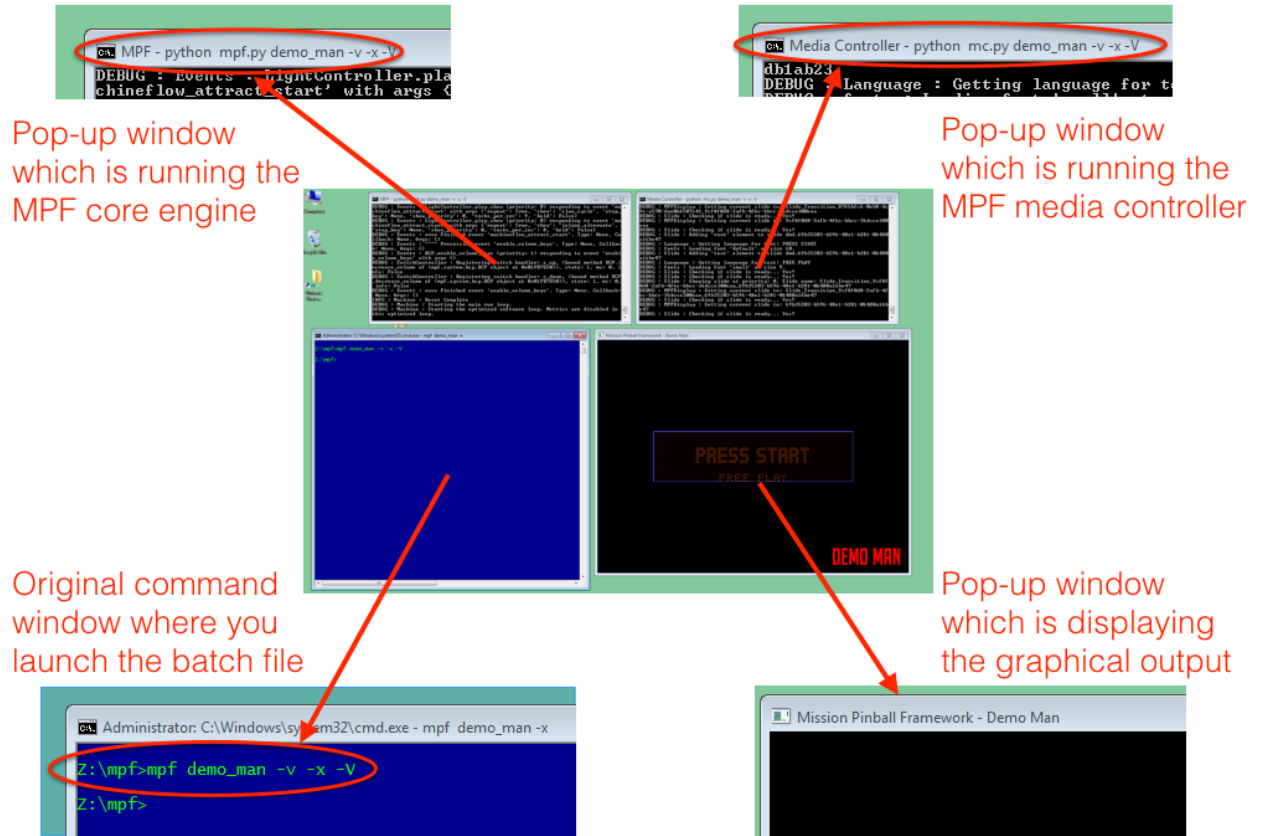
If you're using Windows, at this point you can probably switch over to using our batch file called `mpf.bat` which you can use to launch both the MPF core engine and the media controller at the same time. Using this batch file is completely optional. To use it, simply run "mpf" from the command line (which will run `mpf.bat`) and then pass it the same parameters and options as when you launch `mpf.py`. For example:

```
mpf c:\pinball\your_machine -v
```

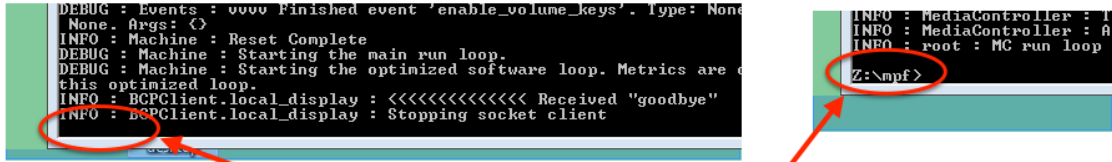
What happens in the batch file is it takes whatever options you pass it then first runs `python mc.py` with your options, and then it runs `python mpf.py` with your options. The default behavior of the batch file is to pop up two new console windows—one for the media controller and one for the MPF core, and then when MPF exits it keeps the windows open so you can see any errors or crashes that may have occurred. If you edit `mpf.bat`, you'll see that there comments in there which explain how to change this behavior (for example, you

might want to have the pop-up windows automatically close when MPF ends, or you might want to launch MPF in the current window instead of a new window).

Here's what your desktop would look like if you use the Windows batch file to launch both windows: (click on the image to see it full size if it's too small on your screen)



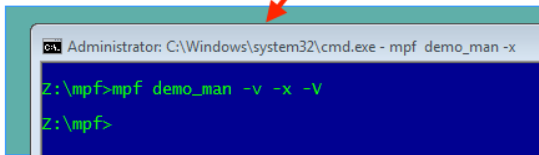
You can click in the pop-up window with the DMD and then hit `ESC` to shut down the media controller, but you'll still have to click into the MPF window and hit `CTRL+C` (or just click the "X" in the corner of the window) to stop the MPF core process. The following image shows what your desktop will look like after you stop the media player but while the MPF core is still running.



MPF game engine is still running. (You can configure it to auto stop when the media controller stops in the options.)

Media controller has stopped

Original command window where you launch the batch file



Pop-up window is gone since the media controller stopped

(H) Getting some useful information to show up in your DMD window

Okay, now we're finally ready to get some useful things to show up on the DMD. There are a lot of different ways to do this, and as you get deeper into your game development you'll end up mixing-and-matching techniques, but at this point in the tutorial the easiest way is to use the "Slide Player." MPF's Slide Player lets you group together one or more display elements ([text](#), [images](#), [animations](#), [drawing shapes](#), etc.) into [slides](#) which are then shown on a display when certain events happen.

We haven't really talked about *events* yet. MPF is an event-driven framework, which means that MPF posts "events" when certain things happen, and then other parts of MPF watch for those events and act on them. (You can read more about MPF's event system [here](#).)

Events are really, really important in MPF. They happen fast, like hundreds per second. Think of events like MPF's running commentary on everything that's happening at all times. *The trough just got a ball. The player just hit the flipper button. This switch just became active. The DMD needs a new frame. The ball save timer just expired. A ball just entered the VUK. etc.*

Anyway, it's these events you use to trigger the Slide Player to show a slide on the display. Let's step through this using a plain-English example first, and then we'll look at how you can set these up.

Let's say you want a the DMD to show the word "Jackpot" when a jackpot shot was made. First you'd enter a configuration for your jackpot slide. That might include a text element with the word "Jackpot" in a certain font at a certain size. Then you use the Slide Player to link the displaying of that slide to the "jackpot" game event, so then whenever the Slide Player sees the jackpot event it will show the slide you configured for it.

The first step is to add a new top-level (i.e. no spaces before it) section to your config file called `slide_player:`. Then underneath it you create entries for the various events you want to display slides for, and then under each event you add settings for what you want to display.

(1) Adding "Press Start" text during attract mode

For example, let's say you want to show the text "PRESS START" on the DMD while the machine is running its attract mode. To do that, you can create a `slide_player:` entry which shows a slide when the event `mode_attract_started` is posted. (How did we know that? We looked at the list of events that MPF generates. We'll show you how to do that later. For now just know that `mode_attract_started` is posted when the attract mode starts. Add an entry to your `slide_player` like this:

```
slide_player:
  mode_attract_started:
    type: text
    text: PRESS START
    slide_priority: 10
```

You can make all sorts of adjustments to this text element, including the [font](#), [size](#), [positioning](#), [decorations](#) (blinking, etc.), [transitions](#) (sliding in), running it through the [alternate language module](#), etc, etc. But for now if you just enter like above, you'll get the default font, at the default size, in the default position.

If you run your game again, your popup window should look like the image on the left, and if you have a physical DMD (with `physical: yes` in your `dmd:` section), then your physical DMD should look like the picture on the right:



Hey! This is a big step. You have a working DMD now!!!

(I) Configure other window settings

While we're still working with the window, it's probably worth mentioning that there are a few other settings you can use to control the look and feel of the on screen window. These are all covered in the [window: section of the config file reference](#), so you can take a look there and see if you want to configure anything else. (These control things like the window title, the size in pixels, whether it's full screen or has a frame, etc.)

Taking some of these settings and adding them into your existing `Window:` configuration section would result in something that looks like this:

```

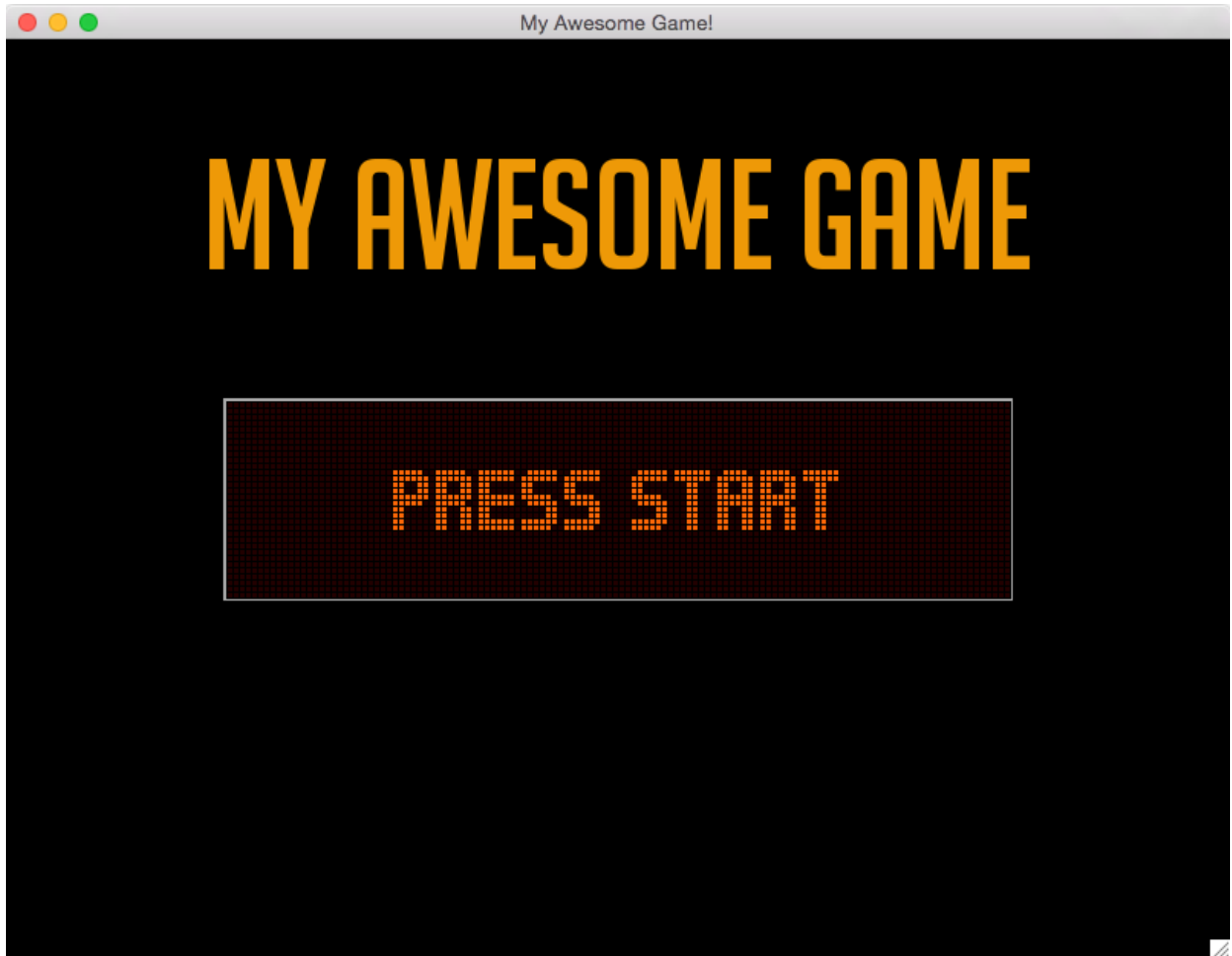
window:
  title: My Awesome Game!
  elements:
    - type: virtualdmd
      width: 512
      height: 128
      h_pos: center
      v_pos: center

```

```
pixel_color: ff6600
dark_color: 220000
pixel_spacing: 1
- type: shape
  shape: box
  width: 516
  height: 132
  color: aaaaaa
  thickness: 2
- type: text
  font: tall title
  text: MY AWESOME GAME
  h_pos: center
  v_pos: top
  y: 60
  size: 100
  antialias: yes
  layer: 1
  color: ee9900
```

Notice that we added another display element to our window, this time adding text in the game font "tall title", size 60, color yellow (ee9900), positioned 60 pixels down from the top center of the window, with text that says "MY AWESOME GAME". We learned about these settings from the [Text display element](#), with instructions for positioning from the [positioning & placement documentation](#).

If you run your game again, your on screen window should look like this:



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 MPF package at `<your_mpf_root>/machine_files/tutorial/config/step6.yaml`. (Again remember you need to rename this file to `config.yaml` to use it.) And remember if you're using physical hardware, your coil and switch numbers will be different than the ones in the sample file, since you'll need to configure them based on your driver boards' actual inputs and outputs.

You can run this tutorial example config via the following command. (This assumes you're on Windows.)

```
mpf tutorial -c step6 -v
```

Step 7. Add keyboard control

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

In this next step, we're going to add some keyboard settings to your 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 the machine. :)

(A) 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 to your config file, 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
```

Again make sure that you have proper YAML formatting. The "z" and "/" entries should be 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.

(B) Test your new keyboard interface!

At this point we're ready to test this out. 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 (each one in a separate console window)

```
python mc.py c:\pinball\your_machine
```

```
python mpf.py c:\pinball\your_machine
```

Or if you're on Windows and you're using the batch file launcher, via:

```
mpf c:\pinball\your_machine
```

Note that if you have a physical machine connected, *your physical flippers will not flip with the keyboard keys*. (We'll cover why not in Step (C) below.)

In order for the keys to work, the catch is that the graphical popup window (the one with the virtual DMD 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—it's the Pygame logo which is a yellow cartoon alligator holding a Nintendo controller in his mouth) 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 core 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 core engine via the TCP socket the two processes use to communicate. If you launched the media controller with the `-v -v` command line options (the uppercase `-V` enables verbose logging to the console), you'll actually see the BCP (that's "Backbox Control Protocol"—the protocol the media controller and MPF core engine use to communicate) messages being sent. They'll look something like this:

```
DEBUG : BCP : Sending "switch?state=1&name=s_left_flipper"
DEBUG : BCP : Sending "switch?state=0&name=s_left_flipper"
```

And if you enabled verbose logging of the MPF core engine, you'll also see BCP messages showing it receiving the switch messages from the media controller, like this:

```
DEBUG : BCPCClient.local_display : Received
"switch?state=1&name=s_left_flipper"
```

```

DEBUG : SwitchController : <<<<< switch: s_left_flipper, State:1 >>>>>
DEBUG : BCPCClient.local_display : Received
"switch?state=0&name=s_left_flipper"
DEBUG : SwitchController : <<<<< switch: s_left_flipper, State:0 >>>>>

```

Notice that there are actually state changes each time you hit 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, and MPF (thanks to Pygame) keeps it all straight.

(C) Why can't "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?!?

To understand why this happens you have to understand how the MPF handles "quick response" switches which are used for things like flippers, slingshots, and pop bumpers. As you can imagine, those types of devices require near-instant response. When you hit a flipper button, you want that flipper to fire *instantly*. So for those types of devices, MPF actually writes a out a configure to the physical pinball controller that tells it "Hey, if you see this switch, fire this coil." (This happens with both P-ROC and FAST controllers.) That keeps the entire cycle within the pinball controller itself, and the pinball controller can respond by firing that coil with sub-millisecond level response times.

Compare that to the alternative where the player would push a button on the pinball machine, the pinball controller would receive it, then it would have to be sent via USB to the host computer, then the host computer would have to process it and figure out that a coil needed to be fired, then it would have to send that coil firing command back across the USB bus to the pinball controller, and then finally the pinball controller could fire that coil. Even though computers are really fast, that whole process would still take a few milliseconds which would be horrible in a pinball scenario.

To deal with this, the pinball controllers allow rules to be written to their hardware where we can set up which coils we'd like to be fired (and with which settings) when switches change state. So part of what the thousands of lines of code of MPF do behind the scenes is when you set up your flippers in the `flippers:` section of your config file, it actually writes those rules out to the hardware controller so the hardware controller can handle them. These rules are dynamic and updated often. For example they're deactivated when a game is not in progress, when it tilts, and (optionally) between balls, and you can change them to do all sorts of novelty things like inverted flippers, no hold flippers, weak flippers, etc.

By the way, even when you write hardware rules to your pinball controller, the MPF software still receives notification when those switches change state. After all, you might want to play a sound effect or update a score even if the hardware controller fired the actual coil, and in the case of flipper buttons you need to know when they're activated for lane changes and to cancel video modes and stuff. In this case you still have one physical switch in your machine and one switch configured in your config files—it's just that if you have a hardware rule configured for a switch then when that switch changes state, the pinball controller fires the associated coil *in addition* to sending the switch state change to MPF as usual.

You might think there's a potential issue around timing for this situation. After all, if the hardware fires the coil instantly but it takes awhile for the host computer to receive notification and play a sound, doesn't that mean that the coil action and sound will be out of sync? In theory, yes, but in practicality we're only talking about a few milliseconds—much faster than any human can notice. (In fact if you think about it, sound only travels at about 1 foot per millisecond, so a typical player standing 4 feet away from the speakers in the backbox is already hearing everything on a 4ms delay, and of course no human can notice that, so in this case we're fine.)

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+CRTL+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 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 MPF package at `<your_mpf_root>/machine_files/tutorial/config/step7.yaml`. (You need to rename this file to `config.yaml` to use it.) And remember if you're using physical hardware, your coil and switch numbers will be different than the ones in the sample file, since you'll need to configure them based on your driver boards' actual inputs and outputs.

Step 8. Create your trough

At this point you have a flipping machine with a DMD, but you don't have a "working" pinball machine since you can't start or play games. All you can do is flip. So the next step in this tutorial is to get your first two *ball devices* set up—your trough and plunger lane. That will give MPF a basic understanding of where your balls are and allow it to start moving through the game, player, and ball cycles. (Once you have ball devices defined, an MPF core component called the [Ball Controller](#) is able to start tracking where your balls are at all times.)

(A) Understanding what "devices" are

The first step to setting up your ball devices is that you have to understand what a ball device is. :) Broadly-speaking, a device in the MPF world is a physical thing that's in your pinball machine.

Back in Step 4 we showed you how to add your flipper *devices* and we briefly outlined what devices are. We also said there are low level, *physical* devices (switches, coils, lights, LEDs, etc.) as well as higher-level, *logical* devices that intelligently group together the lower-level physical devices. A ball device is one of the logical devices (like the flippers).

Not every low-level hardware device will be part of a higher level logical device—sometimes a switch is just a switch—but in many cases your low level devices will end up being part of something bigger.

So to configure your first two ball devices (the trough and the plunger lane), you'll first add the coils and switches to their respective sections, then you'll group those building block devices into higher level ball devices.

(B) Understand what "ball devices" are

A [ball device](#) quite literally anything in a pinball machine that holds a pinball (even for a moment), including your trough, the plunger lane, playfield locks, VUKs, the gumball machines in *Twilight Zone*, a kickout hole, etc. (Actually behind the scenes the playfield is a ball device too, because when a ball is rolling around on it, it's "in" the playfield device.)

Your machine will most likely have lots of ball devices. Turning around and looking at the *Judge Dredd* machine behind me, I count eight(!) ball devices: the trough, the right plunger lane, the left plunger lane, the Sniper VUK, the Hall of Justice VUK, the Deadworld orbit thingy, the crane, and the playfield.

MPF keeps track of how many balls are in each ball device at all times, and it assumes that any balls *not* in ball devices are either in transit from one device to another, or they're stuck somewhere. Most ball devices have a one-to-one ratio of ball switches to ball capacity, so MPF can simply count how many switches are active to see how many balls each device has

at any given time. (Not all ball devices have ball switches for every ball—the gumball machine in *Twilight Zone* is a good example of this—but we'll get to that later.)

Ball devices support all sorts of settings and commands. They can post events when balls enter or exit, you can 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, if you find a ball device that you can't figure out how to configure, [contact us](#) and we'll build the support for it.)

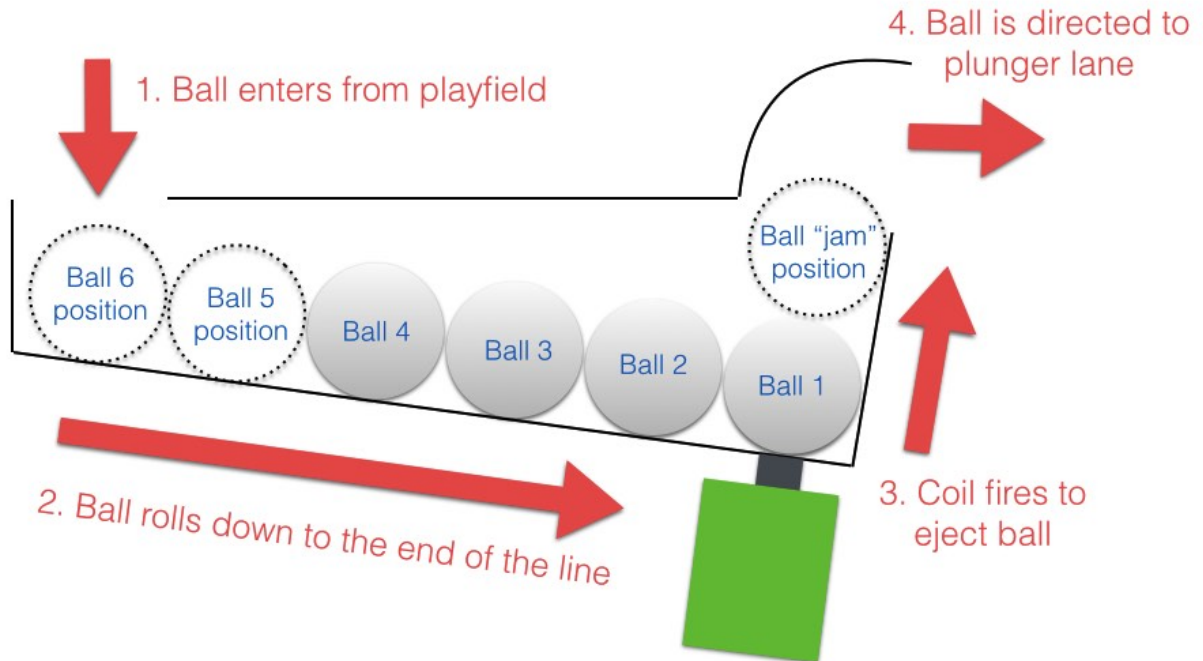
(C) Understanding "trough" ball devices

In modern pinball machines, the trough is the ball device that holds balls after they've drained off the playfield. The act of a ball entering a trough device usually triggers a *ball drain* event, and when the game wants to launch a ball into play, it ejects a ball from the trough (typically into another ball device like the plunger lane or some sort of launch catapult). So when we're building the MPF config for a new machine, the first ball device we create is the trough.

First, though, we should talk about what exactly a trough is. A modern Stern- or Williams-style trough usually holds between 4 and 6 balls. The trough sits underneath the playfield so that a ball entering it is gravity-fed and rolls down to the end of the line. There's a switch (either a physical leaf switch or an opto switch) for each ball position which lets the machine know how many balls are in the trough, and there's a solenoid at the end that pulses to kick a ball out of the trough and into the shooter lane.

Most modern troughs also have a switch in the upper position near the exit that's used to detect if a ball falls back into the trough from the "exit" side—something that tends to happen if your eject coil pulse is too strong or too weak. (Too weak means the ball falls back in because it didn't have enough oomph to make it out, and too strong means the ball flies out too fast, bounces off the right edge rail of the plunger lane, and lands back in the trough after the other balls have rolled down into their new positions.) In MPF we refer to this as a "jam" switch though it's also called a "ball stacked" or "up ball" switch depending on whose manual you're reading.

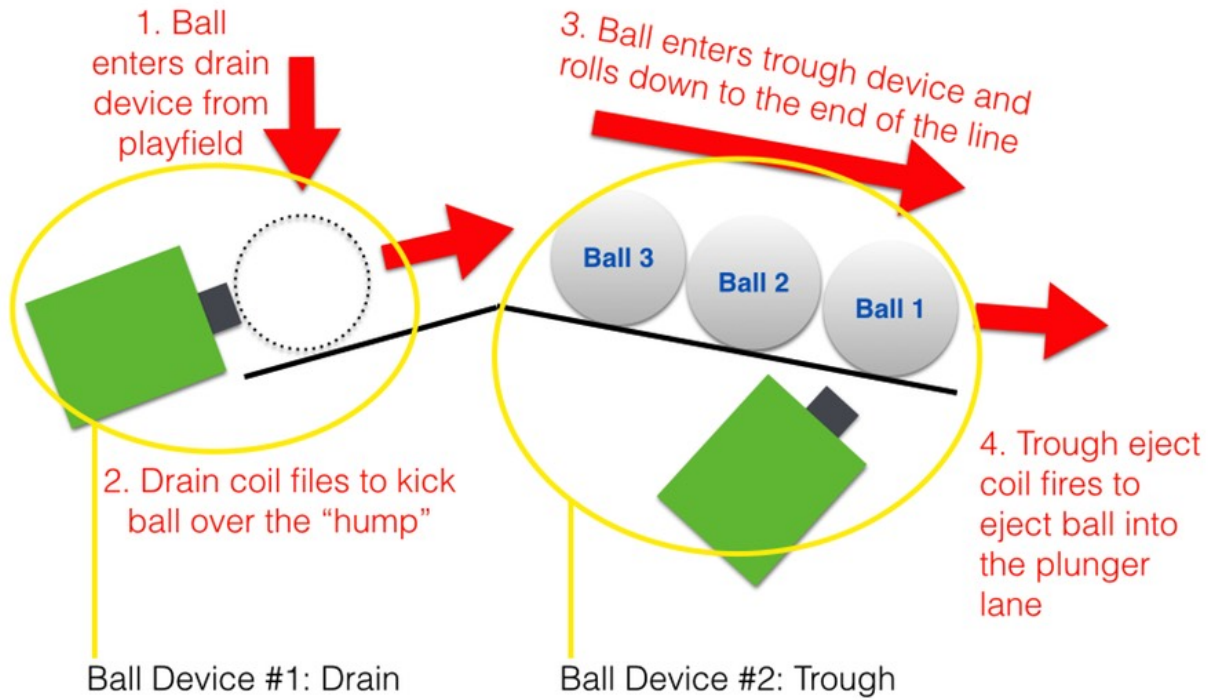
Here's a side-view diagram of a modern style trough:



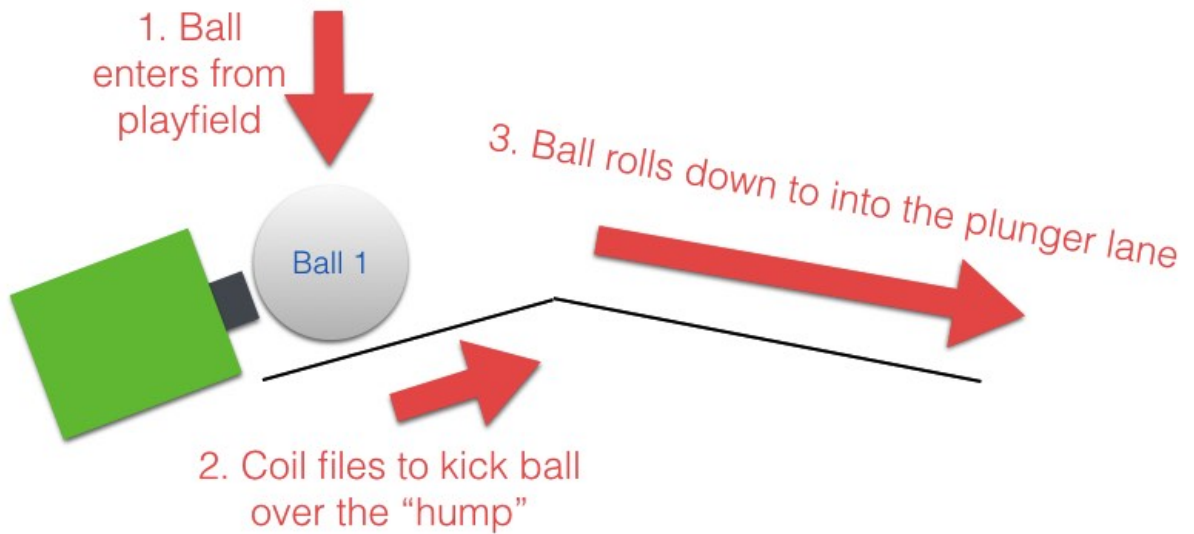
The diagram above does not show the ball switches, but you get the idea.

If this trough diagram does not look like your trough...

Of course not all troughs are the same. In older machines (most 1980s machines, Williams System 11, and early WPC machines), the trough was entirely above the playfield, and a second ball device (called the "outhole") was responsible for receiving drained balls from the playfield and ejecting them into the trough. If you have a classic style outhole + trough combination that looks something like the diagram below, you can see how to configure it via [this "How To" guide for 1980s-style troughs](#).



If you have a very old machine single ball machine (almost all EMs and most early solid state machines) like the diagram below, then you can typically configure your trough just like a modern trough. (It's just that there's only one switch and one coil, but everything in Steps (D) and (E) below still apply.)



(D) Adding your trough switch(es)

Now that we've covered the background information of low-level devices, logical devices, ball devices, and looked at some different types of troughs, we're finally ready to build the actual configuration for your trough. In this tutorial we're going to assume you have a modern style trough device similar to the one shown above.

The first step is to add your trough's switches to the `switches:` section of your config file and the trough's eject coil to the `coils:` section. Let's start with the switches.

Create an entry in your `switches:` section for each switch in your trough, like this:

```
switches:
  s_left_flipper:
    number: 00
  s_right_flipper:
    number: 01
  s_trough1:
    number: 02
  s_trough2:
    number: 03
  s_trough3:
    number: 04
  s_trough4:
    number: 05
  s_trough5:
    number: 06
  s_trough6:
    number: 07
  s_trough_jam:
    number: 08
```

Like the switches we added before, this tutorial just uses generic numbers, but the actual numbers you enter need to reflect that actual hardware numbers you have in your particular machine. They might be something like SD12 or S62 or 2/5. Refer to [switches: section of the configuration file reference](#) to see the exact number pattern for your particular hardware controller and machine combination.

If your trough uses opto switches instead of mechanical switches, then their states will be inverted. In other words they will report "active" when there is no ball present and "inactive" when there is a ball, so you need to add a further entry `type: NC` (normally closed) to tell MPF that this is a normally-closed switch (meaning it's closed when there's no ball and open when there's a ball.) Doing this means that MPF will automatically flip every reading of that switch it gets, so in MPF a state of "1" (active) truly will mean that the switch is active and there's a ball there. An example of this will look like this:

```
s_trough1:
  number: 02
  type: NC
s_trough2:
```

```
number: 03
type: NC
```

It makes no difference which switch is which (in terms of whether Switch 1 is on the left side or the right side). 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 `s_trough_ball_1` or `s_mr_potatohead`. (Just remember (1) you can only use letters, numbers, and underscores, and your name can't start with a number, (2) device names are not case-sensitive, and (3) we recommend starting your switch names with "s_" to make it easier for your editor to autocomplete them later.) Also, we should note, that these names 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.

If you don't have a trough jam switch that's fine, you would just enter your other switches.

(E) 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_flipper_left_main:
    number: 00
    pulse_ms: 20
  c_flipper_left_hold:
    number: 01
  c_flipper_right_main:
    number: 02
    pulse_ms: 20
  c_flipper_right_hold:
    number: 03
  c_trough_eject:
    number: 04
    pulse_ms: 20
```

Again the exact number you enter will be dependent on how your coil is physically connected to your pinball controller. Refer to the [coils: section of the config file reference](#) for details.

You'll also note that we went ahead and entered a `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.

(F) Add your "trough" ball device

Next create a new top-level section (i.e. an entry with no spaces in front of it) in your config file called `ball_devices:`. Then on the next line, enter four spaces and create an entry called `trough:`, like this:

```
ball_devices:
    bd_trough:
```

This means that you're creating a ball device called *bd_trough* (or whatever word you use for the setting with four spaces before it). Again 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, type eight spaces and start entering the configuration settings for your trough ball device:

(1) Add your trough switches

Create an entry called `ball_switches:` and then add a comma-separated list of all the switches in your trough, like this: (At this point if you have an editor that supports autocomplete then you'll really start to appreciate why we preface our switch names with "s_")

```
    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...

Note that 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. (That's why we also include the trough jam switch here, since if that jam switch is active, that means there's a ball sitting on top of another ball right near the exit of your 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 your 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.

(2) Add your entrance count delay

Next add a setting called `entrance_count_delay:` followed by a time entry. (This time entry can be in seconds or milliseconds and is entered with the same formatting you would use to enter any time duration setting in your config files as we explain [here](#).)

This setting tells MPF how long a ball switch has to be in a steady state before it checks all the switches to see how many balls are in this device. If we didn't have this delay then a ball entering a trough would be chaos. (Imagine the physical switch changes when a ball enters a trough. First the switch *trough6* would activate, then *trough6* would become inactive a few milliseconds later, then *trough5* activates, then *trough5* becomes inactive, then switch *trough4* activates, etc.

In our example we're going to use *300ms* (which is probably fine for most cases). This means that whenever any trough switches change state, MPF will wait until they've all been in a steady state for 300 ms before it actually counts them. If it tries to count them before all the switches haven't changed in 300 ms, it will abort the count and try again once they're all settled.

```
entrance_count_delay: 300ms
```

MPF actually uses a default *entrance_count_delay*: time of 500ms, so if you don't enter a setting here, it will use 500ms. So really you can ignore this setting, but we wanted to mention it so you knew about it.

(3) Add your eject coil

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 this ball device. This should be the name of the coil you just added above, *c_trough_eject* in our case:

```
eject_coil: c_trough_eject
```

(4) 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. You can add any tags to any device you want in MPF, and tags make grouping and programming for certain devices easier later on. MPF also uses some special tag names to tell it how it should treat certain devices.

For your trough, we're going to use a few special tags.

First, we'll add a tag called ***trough*** which tells MPF that this device wants to hold as many balls as it can. (You need this tag even if you have one of the System 11 or EM-style troughs.) 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 you'll 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." (So if you're programming a machine like Star Trek: The Next Generation which holds a ball in the upper playfield lock when a game starts, you'd add a tag of `home` to that ball device too.)

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:

```
tags: trough, home, drain
```

(5) Enable debugging so you can see cool stuff in the log

Finally, add an entry `debug: yes` to your trough which will cause MPF to write detailed debugging information about this 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.

At this point your trough configuration should be complete! If you followed along exactly, the `ball_devices:` section of your config file 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
    entrance_count_delay: 300ms
    eject_coil: c_trough_eject
    tags: trough, home, drain
    debug: yes
```

(G) 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. (If you don't have a physical machine attached then you can skip this step.)

To do so, launch the MPF core engine with the `-v` command line options so it shows the verbose information in the log file, like this:

```
python mpf.py your_machine -v
```

You don't have to launch the media controller this time since we're just looking at the console output of the MPF core engine, though if you want to run both MPF and the media controller than that's fine too.

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).

Now quit MPF and open the MPF log file (the one with *mpf* in the name, not *mc*, since you want the log file from the MPF core engine) and scroll to the bottom.

You should see all sorts of messages and events about the ball entering the trough, including it updating its ball count, processing the newly-entered ball, messages about the playfield ball count changing, etc. You don't have to know what any of this means, but it's kind of cool to see things happening!

Here's an example of everything that happens after a single ball switch is activated in the trough.

```
2015-11-29 22:01:14,911 : INFO : SwitchController : <<<<< switch:
s_trough1, State:1 >>>>>
2015-11-29 22:01:14,913 : DEBUG : SwitchController : Found timed switch
handler for k/v 1448863275.21 / {'callback': <bound method
BallDevice._switch_changed of <ball_device.bd_trough>>, 'state': 1,
'switch_action': 's_trough1-1', 'ms': 300, 'callback_kwargs': {},
'switch_name': 's_trough1', 'return_info': False}
2015-11-29 22:01:14,914 : DEBUG : Events : ^^^^ Posted event
's_trough1_active'. Type: None, Callback: None, Args: {}
2015-11-29 22:01:14,920 : DEBUG : Events : ===== EVENTS QUEUE
=====
2015-11-29 22:01:14,923 : DEBUG : Events : s_trough1_active, None, None, {}
2015-11-29 22:01:14,926 : DEBUG : Events :
=====
2015-11-29 22:01:14,927 : DEBUG : Events : ^^^^ Processing event
's_trough1_active'. Type: None, Callback: None, Args: {}
2015-11-29 22:01:14,927 : DEBUG : Events : vvvv Finished event
's_trough1_active'. Type: None, Callback: None, Args: {}
2015-11-29 22:01:15,243 : DEBUG : SwitchController : Processing timed
switch handler. Switch: s_trough1 State: 1, ms: 300
2015-11-29 22:01:15,244 : DEBUG : ball_device.bd_trough : Counting balls
2015-11-29 22:01:15,246 : DEBUG : ball_device.bd_trough : Confirmed active
switch: s_trough1
2015-11-29 22:01:15,246 : DEBUG : ball_device.bd_trough : Confirmed
inactive switch: s_trough2
```

```

2015-11-29 22:01:15,246 : DEBUG : ball_device.bd_trough : Confirmed
inactive switch: s_trough3
2015-11-29 22:01:15,247 : DEBUG : ball_device.bd_trough : Confirmed
inactive switch: s_trough4
2015-11-29 22:01:15,249 : DEBUG : ball_device.bd_trough : Confirmed
inactive switch: s_trough5
2015-11-29 22:01:15,250 : DEBUG : ball_device.bd_trough : Confirmed
inactive switch: s_trough_jam
2015-11-29 22:01:15,250 : DEBUG : ball_device.bd_trough : Counted 1 balls
2015-11-29 22:01:15,253 : DEBUG : ball_device.bd_trough : Received 1
unexpected balls
2015-11-29 22:01:15,256 : DEBUG : Events : ^^^^ Posted event
'balldevice_captured_from_playfield'. Type: None, Callback: None, Args:
{'balls': 1}
2015-11-29 22:01:15,257 : DEBUG : Events : ===== EVENTS QUEUE
=====
2015-11-29 22:01:15,257 : DEBUG : Events :
balldevice_captured_from_playfield, None, None, {'balls': 1}
2015-11-29 22:01:15,259 : DEBUG : Events :
=====
2015-11-29 22:01:15,262 : DEBUG : Events : ^^^^ Posted event
'balldevice_balls_available'. Type: boolean, Callback: None, Args: {}
2015-11-29 22:01:15,263 : DEBUG : Events : ===== EVENTS QUEUE
=====
2015-11-29 22:01:15,263 : DEBUG : Events :
balldevice_captured_from_playfield, None, None, {'balls': 1}
2015-11-29 22:01:15,265 : DEBUG : Events : balldevice_balls_available,
boolean, None, {}
2015-11-29 22:01:15,272 : DEBUG : Events :
=====
2015-11-29 22:01:15,273 : DEBUG : ball_device.bd_trough : Processing 1 new
balls
2015-11-29 22:01:15,273 : DEBUG : Events : ^^^^ Posted event
'balldevice_bd_trough_ball_enter'. Type: relay, Callback: <bound method
BallDevice._balls_added_callback of <ball_device.bd_trough>>, Args:
{'device': <ball_device.bd_trough>, 'new_balls': 1, 'unclaimed_balls': 1}
2015-11-29 22:01:15,275 : DEBUG : Events : ===== EVENTS QUEUE
=====
2015-11-29 22:01:15,276 : DEBUG : Events :
balldevice_captured_from_playfield, None, None, {'balls': 1}
2015-11-29 22:01:15,276 : DEBUG : Events : balldevice_balls_available,
boolean, None, {}
2015-11-29 22:01:15,278 : DEBUG : Events : balldevice_bd_trough_ball_enter,
relay, <bound method BallDevice._balls_added_callback of
<ball_device.bd_trough>>, {'device': <ball_device.bd_trough>,
'unclaimed_balls': 1, 'new_balls': 1}
2015-11-29 22:01:15,279 : DEBUG : Events :
=====
2015-11-29 22:01:15,283 : DEBUG : Events : ^^^^ Posted event
'balldevice_bd_trough_ok_to_receive'. Type: None, Callback: None, Args:
{'balls': 5}
2015-11-29 22:01:15,289 : DEBUG : Events : ===== EVENTS QUEUE
=====
2015-11-29 22:01:15,289 : DEBUG : Events :
balldevice_captured_from_playfield, None, None, {'balls': 1}
2015-11-29 22:01:15,290 : DEBUG : Events : balldevice_balls_available,
boolean, None, {}
2015-11-29 22:01:15,292 : DEBUG : Events : balldevice_bd_trough_ball_enter,
relay, <bound method BallDevice._balls_added_callback of

```

```

<ball_device.bd_trough>>, {'device': <ball_device.bd_trough>,
'unclaimed_balls': 1, 'new_balls': 1}
2015-11-29 22:01:15,296 : DEBUG : Events :
balldevice_bd_trough_ok_to_receive, None, None, {'balls': 5}
2015-11-29 22:01:15,298 : DEBUG : Events :
=====
2015-11-29 22:01:15,299 : DEBUG : Events : ^^^^ Processing event
'balldevice_captured_from_playfield'. Type: None, Callback: None, Args:
{'balls': 1}
2015-11-29 22:01:15,305 : DEBUG : Events : Playfield._ball_removed_handler
(priority: 1) responding to event 'balldevice_captured_from_playfield' with
args {'balls': 1}
2015-11-29 22:01:15,309 : DEBUG : Events : ^^^^ Posted event
'sw_playfield_active'. Type: None, Callback: <bound method
Playfield._ball_removed_handler2 of <playfield.playfield>>, Args: {'balls':
1}
2015-11-29 22:01:15,313 : DEBUG : Events : ===== EVENTS QUEUE
=====
2015-11-29 22:01:15,315 : DEBUG : Events : balldevice_balls_available,
boolean, None, {}
2015-11-29 22:01:15,315 : DEBUG : Events : balldevice_bd_trough_ball_enter,
relay, <bound method BallDevice._balls_added_callback of
<ball_device.bd_trough>>, {'device': <ball_device.bd_trough>,
'unclaimed_balls': 1, 'new_balls': 1}
2015-11-29 22:01:15,316 : DEBUG : Events :
balldevice_bd_trough_ok_to_receive, None, None, {'balls': 5}
2015-11-29 22:01:15,318 : DEBUG : Events : sw_playfield_active, None,
<bound method Playfield._ball_removed_handler2 of <playfield.playfield>>,
{'balls': 1}
2015-11-29 22:01:15,319 : DEBUG : Events :
=====
2015-11-29 22:01:15,325 : DEBUG : Events : vvvv Finished event
'balldevice_captured_from_playfield'. Type: None. Callback: None. Args:
{'balls': 1}
2015-11-29 22:01:15,328 : DEBUG : Events : ^^^^ Processing event
'balldevice_balls_available'. Type: boolean, Callback: None, Args: {}
2015-11-29 22:01:15,329 : DEBUG : Events : vvvv Finished event
'balldevice_balls_available'. Type: boolean. Callback: None. Args: {}
2015-11-29 22:01:15,331 : DEBUG : Events : ^^^^ Processing event
'balldevice_bd_trough_ball_enter'. Type: relay, Callback: <bound method
BallDevice._balls_added_callback of <ball_device.bd_trough>>, Args:
{'device': <ball_device.bd_trough>, 'unclaimed_balls': 1, 'new_balls': 1}
2015-11-29 22:01:15,332 : DEBUG : Events :
BallController._ball_drained_handler (priority: 1) responding to event
'balldevice_bd_trough_ball_enter' with args {'device':
<ball_device.bd_trough>, 'unclaimed_balls': 1, 'new_balls': 1}
2015-11-29 22:01:15,334 : DEBUG : Events : ^^^^ Posted event 'ball_drain'.
Type: relay, Callback: <bound method BallController._process_ball_drained
of <mpf.system.ball_controller.BallController object at 0x020A6CF0>>, Args:
{'device': <ball_device.bd_trough>, 'balls': 1}
2015-11-29 22:01:15,344 : DEBUG : Events : ===== EVENTS QUEUE
=====
2015-11-29 22:01:15,345 : DEBUG : Events :
balldevice_bd_trough_ok_to_receive, None, None, {'balls': 5}
2015-11-29 22:01:15,346 : DEBUG : Events : sw_playfield_active, None,
<bound method Playfield._ball_removed_handler2 of <playfield.playfield>>,
{'balls': 1}
2015-11-29 22:01:15,348 : DEBUG : Events : ball_drain, relay, <bound method
BallController._process_ball_drained of

```

```

<mpf.system.ball_controller.BallController object at 0x020A6CF0>>,
{'device': <ball_device.bd_trough>, 'balls': 1}
2015-11-29 22:01:15,352 : DEBUG : Events :
=====
2015-11-29 22:01:15,355 : DEBUG : Events : vvvv Finished event
'balldevice_bd_trough_ball_enter'. Type: relay. Callback: <bound method
BallDevice._balls_added_callback of <ball_device.bd_trough>>. Args:
{'device': <ball_device.bd_trough>, 'new_balls': 1, 'unclaimed_balls': 1}
2015-11-29 22:01:15,358 : DEBUG : Events : ^^^^ Processing event
'balldevice_bd_trough_ok_to_receive'. Type: None, Callback: None, Args:
{'balls': 5}
2015-11-29 22:01:15,358 : DEBUG : Events : vvvv Finished event
'balldevice_bd_trough_ok_to_receive'. Type: None. Callback: None. Args:
{'balls': 5}
2015-11-29 22:01:15,362 : DEBUG : Events : ^^^^ Processing event
'sw_playfield_active'. Type: None, Callback: <bound method
Playfield._ball_removed_handler2 of <playfield.playfield>>, Args: {'balls':
1}
2015-11-29 22:01:15,371 : DEBUG : Events : Playfield.playfield_switch_hit
(priority: 1) responding to event 'sw_playfield_active' with args {'balls':
1}
2015-11-29 22:01:15,371 : DEBUG : Events : ^^^^ Posted event
'playfield_active'. Type: boolean, Callback: None, Args: {}
2015-11-29 22:01:15,372 : DEBUG : Events : ===== EVENTS QUEUE
=====
2015-11-29 22:01:15,374 : DEBUG : Events : ball_drain, relay, <bound method
BallController._process_ball_drained of
<mpf.system.ball_controller.BallController object at 0x020A6CF0>>,
{'device': <ball_device.bd_trough>, 'balls': 1}
2015-11-29 22:01:15,375 : DEBUG : Events : playfield_active, boolean, None,
{}
2015-11-29 22:01:15,375 : DEBUG : Events :
=====
2015-11-29 22:01:15,375 : DEBUG : playfield : Playfield_active switch hit
with no balls expected. glass_off_mode is enabled, so this will be ignored.
2015-11-29 22:01:15,377 : DEBUG : Events : vvvv Finished event
'sw_playfield_active'. Type: None. Callback: <bound method
Playfield._ball_removed_handler2 of <playfield.playfield>>. Args: {'balls':
1}
2015-11-29 22:01:15,385 : DEBUG : Events : ^^^^ Processing event
'ball_drain'. Type: relay, Callback: <bound method
BallController._process_ball_drained of
<mpf.system.ball_controller.BallController object at 0x020A6CF0>>, Args:
{'device': <ball_device.bd_trough>, 'balls': 1}
2015-11-29 22:01:15,388 : DEBUG : Events : vvvv Finished event
'ball_drain'. Type: relay. Callback: <bound method
BallController._process_ball_drained of
<mpf.system.ball_controller.BallController object at 0x020A6CF0>>. Args:
{'device': <ball_device.bd_trough>, 'balls': 1}
2015-11-29 22:01:15,391 : DEBUG : Events : ^^^^ Processing event
'playfield_active'. Type: boolean, Callback: None, Args: {}
2015-11-29 22:01:15,392 : DEBUG : Events : vvvv Finished event
'playfield_active'. Type: boolean. Callback: None. Args: {}
2015-11-29 22:01:15,395 : DEBUG : playfield : 1 ball(s) removed from the
playfield
2015-11-29 22:01:15,397 : DEBUG : playfield : Ball count change. Prior: 0,
Current: -1, Change: -1
2015-11-29 22:01:15,398 : WARNING : playfield : Playfield balls went to -1.
Resetting to 0, but FYI that something's weird

```

```

2015-11-29 22:01:15,398 : DEBUG : playfield : New Ball Count: 0. (Prior
count: 0)
2015-11-29 22:01:15,404 : DEBUG : Events : ^^^^ Posted event
'playfield_ball_count_change'. Type: None, Callback: None, Args: {'balls':
-1, 'change': -1}
2015-11-29 22:01:15,411 : DEBUG : Events : ===== EVENTS QUEUE
=====
2015-11-29 22:01:15,413 : DEBUG : Events : playfield_ball_count_change,
None, None, {'balls': -1, 'change': -1}
2015-11-29 22:01:15,414 : DEBUG : Events :
=====
2015-11-29 22:01:15,414 : DEBUG : Events : ^^^^ Processing event
'playfield_ball_count_change'. Type: None, Callback: None, Args: {'balls':
-1, 'change': -1}
2015-11-29 22:01:15,415 : DEBUG : Events : vvvv Finished event
'playfield_ball_count_change'. Type: None. Callback: None. Args: {'balls':
-1, 'change': -1}

```

(H) Add keyboard entries for your trough switches

While we're working with the trough config, let's create some keyboard-to-switch entries in your config file for your trough switches. You do this just like how you created the entries you added in the Step 7. For example (in your *keyboard:* section):

```

1:
    switch: s_trough1
    toggle: true

```

The *toggle: true* setting for this keyboard entry sets up this key as a "toggle" key meaning that it functions in a push on / push off kind of way. (Without *toggle: true*, you'd have press and hold the key to represent the ball activating the switch. With *toggle: true*, you tap the key once to activate the switch, and tap it a second time to deactivate it.)

By the way, all of these true/false settings in the config file give you a lot of leeway. You can enter the values as *true*, *True*, *on*, or even *yes*, and you can use *No*, *no*, *off*, *false*, etc. for "no" values.

On important note about the "toggle" function: The toggle function applies to keyboard keys, *not* to switches. In other words in your physical machine, there is no concept of a "toggle" style switch. The switch is open when it's open and closed when it's closed. The toggle function only affects how keyboard key behavior maps to switches in your machine. (If you really want some fun, try using the toggle keys when you have a live machine connected with balls in the trough. MPF will get really confused. :)

Now re-run MPF with `-v -v` for verbose screen logging and tap the 1 key. (This time since we're using the keyboard interface and that requires the graphical on-screen window from

the media controller, you'll need to launch both the MPF core engine and the media controller.)

After 300ms, (your entrance delay), you should see a whole bunch of messages about a ball entering the trough.

Now make some more key entries for the trough and set them all to toggle. In this example we'll set up one key for each regular ball switch in the trough using the number keys. So now the *keyboard:* section of your config file might look like this:

```
keyboard:
  z:
    switch: s_left_flipper
  /:
    switch: s_right_flipper
  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
  6:
    switch: s_trough6
    toggle: true
```

You don't have to enter keyboard shortcuts for all the switches if you don't want to. One should be fine for now.

(I) Configure your virtual hardware to start with balls in the trough

If you're following along with virtual hardware, at some point you're going to get annoyed because you'll have to press the 1 key every time you run MPF to make it think there's a ball in the trough. (As you'll soon learn, if you don't do this then MPF will think there are no balls, so it won't start a game.)

To get around that, you can add the a new section to your config file called **virtual_platform_start_active_switches:**. (Sorry this entry name is hilariously long. We tried couldn't really think of any other name that accurately described what it does.)

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 or smart 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
```

Note that you don't actually have to have *keyboard:* entries for any of these *virtual_platform_start_active_switches* for switches. If you do, though, and if those keyboard entries are set to *toggle: true*, then that just means the switches will start out in the active state, and the first time you hit the key associated with a switch, the switch will change from active to inactive.

(J) Troubleshooting if something didn't work

If you've gotten this far and your trough isn't working right, there are a few things you can try (depending on what your problem is).

If your log file shows a number of balls contained in your trough that doesn't match how many balls you actually have, that either means that (1) you didn't add all the ball switches to the *ball_switches:* section of the trough configuration, or (2) your trough uses opto switches but you didn't add *type: NC* to each switch's configuration, or (3) 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.

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 trough configuration exactly match the switch and coil names in the `switches:` and `coils:` sections of your config file (including the same capitalization).

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, like this:



```
Switches:
  leftFlipperButton:
    number: 0
  rightFlipperButton:
    number: 1
```

At this point your trough is ready to go! Next we have to configure your plunger lane.

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 MPF package at `<your_mpf_root>/machine_files/tutorial/config/step8.yaml`. (You need to rename this file to `config.yaml` to use it.) And remember if you're using physical hardware, your coil and switch numbers will be different than the ones in the sample file, since you'll need to configure them based on your driver boards' actual inputs and outputs.

Step 9. Create your plunger lane

The previous step was really long since we had to go through all the basics of what ball devices are and how they're configured. In this step we're going to create a second ball device—your *plunger lane* (or *shooter lane* or *ball launcher* or *catapult* or whatever you want to call it). We'll just call it the *plunger lane* in this tutorial, but again, feel free to call it whatever makes you happy.

Remember that a ball device is anything in a pinball machine that can hold a ball, so the plunger lane is no exception. The exact settings you use for the plunger lane will vary depending on the type of plunger you have. Options include:

- A manual plunger (where the player pulls on the spring knob to launch the ball).
- A coil-powered ball launcher where the player hits a "launch" button which fires a coil to launch the ball.
- A combination of both (where the player has the option to use a manual spring loaded plunger but where the machine can also use a coil to launch the ball).

Regardless of the type of plunger, all modern machines have one thing in common: there's a switch in the plunger lane/catapult/whatever which is active when a ball is sitting in the plunger waiting to be fired. So we'll get the plunger ball device set up first, and then if you have a coil-powered plunger we'll walk through the additional steps you need to configure that.

That said, what if your plunger lane doesn't have a switch in it which is activated when a ball is sitting at the plunger? (This is common in single ball machines, like EMs and older solid state machines.) In this case you can't follow this step in the tutorial since you don't have the switch. Instead check our guide [How to configure plunger lanes that don't have plunger switches](#). Then come back here and pick up with the next step in the tutorial.

Modern Plunger Lane



Switch which is active when a ball is resting in the lane.

Old Plunger Lane



This switch doesn't count. It's higher up and used in this machine to tell the machine a ball has entered the plunger lane from the playfield via a gate in the outlane. But it is not activated when a ball comes from the trough, so we ignore it when configuring the plunger.

No switch. Machine does not know when a ball is here.

(A) Add your plunger lane switch

The first thing to do is to add the switch in your plunger lane to your list of switches in the `switches:` section of your config file. By now this should be pretty easy for you:

```
switches:
  s_left_flipper:
    number: 00
  s_right_flipper:
    number: 01
  s_trough1:
    number: 02
    type: NC
  s_trough2:
    number: 03
    type: NC
  s_trough3:
    number: 04
    type: NC
  s_trough4:
    number: 05
    type: NC
  s_trough5:
    number: 06
    type: NC
  s_trough6:
    number: 07
    type: NC
  s_trough_jam:
    number: 08
    type: NC
  s_plunger_lane:
    number: 09
```

Again, we're just using the number value of 09 as an example. If you're following this tutorial with a physical machine then you'll need to enter the number that corresponds to your physical switch. See the "number" section of the [Switch settings in the machine config file reference](#) for details.

(B) Create your plunger ball device

Next, go to the `ball_devices:` section of your config file and create a new entry called `bd_plunger:` under your `bd_trough:` section. This plunger entry should be at the same indent level as the trough (i.e. 4 spaces).

(1) Add your ball switch

Under the line that defines your plunger, create an entry called *ball_switches*: and add the name of the switch that you created in Step (A). Even though this setting mentions "switches" (plural), your plunger lane probably only has one switch since it only holds one ball. (Though some have two switches for two balls. If that's your scenario then add them both here.)

(2) Set up your entrance count delay

Next, create the *entrance_count_delay*: entry. Remember this is how long the ball switch has to stay in the active state to allow for ball settling. Using 300ms is probably fine for now.

(3) Configure the eject timeout

One of the things we touched on but haven't really talked about is the concept of eject confirmations. In MPF, a ball device needs to know (to the best of its ability) that a ball that was supposed to eject actually ejected properly.

When one device ejects a ball into another device, that eject confirmation is pretty easy—it just looks to see when the ball made it into the target device and it knows that everything went ok. But with devices that eject to the playfield (such as the plunger), it can get a bit tricky.

If there are no balls on the playfield, it's pretty easy to know when a ball made it out because as soon as a playfield switch is hit, MPF knows there's a ball on the playfield.

But if there's already a ball on a playfield and then an additional ball is ejected to the playfield, then when a playfield switch is hit, MPF has no way of knowing whether that switch was hit by the new ball or the existing ball.

Therefore when you configure ball devices that eject to the playfield, you need to configure an *eject_timeout* setting which basically says, "if you eject a ball and the ball leaves (because the ball switch deactivated), and then the ball doesn't come back after this amount of time, then MPF assumes the ball made it out of the device."

The exact amount of time you should use for your eject timeout depends on your machine. For a plunger lane, you want to try to figure out what the longest possible time is that a failed plunge could still end up with the ball coming back to the plunger lane. If you have a manual spring plunger and a plunger lane that wraps all the way up the side, the eject timeout could be 3 or 4 seconds.

Even if you have a coil-fired eject plunger, you have to set your timeout in case your coil gets weak and can't eject the ball all the way. So get out your stopwatch and put a ball in your plunger lane and time how long it takes for the ball to go from the plunger to the very end of the plunger lane, stop, and then roll back down to the plunger.

You'll have to play with this setting to get it right.

If it's set too short, then you could wind up with a scenario where you have two balls in the plunger lane. (This could happen if you had an eject timeout set for 2 seconds and your machine was adding a bunch of balls into play for multiball. In this case 2 seconds after launch, MPF would think the ball made it out and kick the next ball into the plunger lane to eject it, but if the first ball didn't make it out and was rolling back, then you'll have two balls stuck.)

It's probably best to err on the side of caution, since if your eject timeout is too long that will just mean that you can't add lots of balls into play as fast, but really if you're adding them at a 2 second pace or a 3 second pace, that shouldn't matter.

Anyway, once you decide what you want your timeout to be, then create a setting in your plunger lane for it, like `eject_timeouts: 3s`. (Note that you can enter time values in config files in seconds or milliseconds. [More details on that are here.](#))

By the way, if you're wondering why that setting is called `eject_timeouts:` (plural) instead of `eject_timeout:` (singular), that's because MPF's ball devices are integrated with diverter devices that are used to automatically route balls to different locations, and each location can have its own timeout. But for now you just need to enter the one and if you have diverters and stuff you can configure those once you're done with the tutorial.

(4) Add the tags

Like the trough, there's a special tag we need to add to our plunger lane: `ball_add_live`. The `ball_add_live` tag is used to tell MPF that this ball device should be used to add a live ball into play.

The way it works is when MPF's game controller wants to add a ball into play (typically at the start of a ball), it looks for a device tagged with `ball_add_live` and makes sure that device has a ball that can be ejected.

You add this tag by adding a line `tags: ball_add_live`.

At this point your plunger lane ball device configuration should be all set, looking something like this:

```
bd_plunger:
  ball_switches: s_plunger_lane
  entrance_count_delay: 300ms
  eject_timeouts: 3s
  tags: ball_add_live
```

(C1) Configure your human-power spring plunger

If your plunger is the traditional spring-driven human-powered plunger, then you need to add another configuration option which is *mechanical_eject: true*. The reason for this is that MPF likes to know what's going on with all the balls at all times. If you have a spring plunger, when the player plunges the ball, from MPF's perspective it's like the ball just vanished! So setting the *mechanical_eject: true* lets MPF know that if the ball just disappears then that means the player ejected it and MPF needs to look for the ball to end up in the target device.

So if this applies to your plunger, then your plunger device config should look like this:

```
bd_plunger:
  ball_switches: s_plunger_lane
  entrance_count_delay: 300ms
  eject_timeouts: 3s
  tags: ball_add_live
  mechanical_eject: true
```

If your plunger has both a human-powered eject and a coil-fired eject, then go ahead and add *mechanical_eject: true* here.

(C2) Add your coil for coil-fired plungers

If you have a coil-fired ball launcher or plunger, you can configure that now too. To do this:

(1) Add the coil to your coils: configuration

First, add an entry for your plunger lane eject coil to the *coils:* section of your config file. Your complete section will probably now look something like this:

```
coils:
  c_flipper_left_main:
    number: 00
    pulse_ms: 25
  c_flipper_left_hold:
    number: 01
  c_flipper_right_main:
    number: 02
    pulse_ms: 25
  c_flipper_right_hold:
    number: 03
  c_trough_eject:
    number: 04
    pulse_ms: 25
  c_plunger_eject:
    number: 05
    pulse_ms: 25
```

Again, if you have physical hardware then make sure your new coil's *number*: is accurate, and remember you can adjust the *pulse_ms*: setting here if your plunger eject ends up being too strong or too weak.

(2) Add your eject coil to your plunger

Next add your newly-entered coil name to your plunger ball device configuration so MPF knows that's the coil that should be used to eject a ball from that device. Based on the entry from Step (1) above, that would be *eject_coil*: *c_plunger_eject*.

(3) Add your plunger eject switch

If your plunger device is coil-fired, and if you want the player to hit a button to launch a ball into play, then you can setup that switch now. To do this, add that switch to the switches: section of your config. You also need to add a tag to that switch entry which is how MPF will know that switch is the one that will be used to launch the ball from the plunger. We typically call that tag "launch".

So you would add the following to the switches: section of your config:

```
s_launch_button:
  number: 09
  tags: launch
```

Note that if you have a plunger lane with both a spring-powered plunger and a coil-fired eject, it's possible that you don't actually have a launch button. (Many Stern games are like this.) In those cases the coil is only used for ball save and to auto-launch balls for multiball, so it's possible that you will still add the *eject_coil* to your plunger but you won't actually wire up a switch to it in this step and the next one.

(4) Configure your plunger to eject based on the launch button

If you configured a switch to launch the ball in the previous step, now go back to your plunger ball device and add a setting so that the plunger knows it should eject a ball based on the switch you just setup. To do that, create an entry called *player_controlled_eject_event*: and then set the value to *sw_* followed by the name of the tag you just added to your launch button. (For example, *sw_launch*.)

The reason this works is because by default, when you add tags to switches, whenever that switch is activated then MPF posts an event with the name *sw_<tag_name>*. So every time you hit a switch tagged with *launch*, MPF will post an event called *sw_launch*. (Don't worry—this event won't actually launch a ball from the plunger every time that switch is hit. It's just used when a player-controlled eject is setup from that device which is what MPF does with the ball device tagged with *ball_add_live* whenever a new ball starts.)

So now your plunger ball device config will look something like this:

```
bd_plunger:
  ball_switches: s_plunger_lane
  entrance_count_delay: 300ms
  eject_timeouts: 3s
  tags: ball_add_live
  eject_coil: c_plunger_eject
  player_controlled_eject_event: sw_launch
```

If you have a dual spring/coil fired plunger, you'll also have the *mechanical_eject: true* setting in there.

(D) Go back to your trough device and reconfigure its eject settings

We talked a little bit about how MPF is able to confirm ball ejects because it "knows" which ball devices eject into other devices. In other words when the trough attempts to eject a ball, it will watch the plunger device, and when the plunger device receives a ball, the trough will mark its eject as successful.

Now that you have a plunger device setup, you can go back to the trough settings and configure its eject target.

To do this, in the `bd_trough: ball device settings`, create a new entry called *eject_targets:* with a value of *bd_plunger*. This tells the trough that the *bd_plunger* ball device is the target of its ejects. (The *eject_targets:* setting can actually be a list of more than one device, but in this case the trough only ejects to one place—the plunger—so we only need one entry here.)

This *eject_targets:* entry is used for a few things. First, as we already mentioned, configuring a target device is how the trough knows which device to watch to know that an eject was successful.

A ball device configured with *eject_targets:* setting will also monitor the target devices to see if any of them ever wants a ball. For example, remember before we added the tag *ball_add_live* to the plunger device. This means that when MPF wants to launch a ball into play, it will go to the device tagged with *ball_add_live* and ask that device make sure it has a ball. What happens if that device doesn't have any balls to eject? In that case the plunger device would post an event that says, "I want a ball!" And the trough device, since its target device is the plunger device, would say, "Hey! I have a ball and I can give it to you." So by linking your devices together via the *eject_targets:* settings you can set up a ball path which ensures that any device that needs a ball can get it.

(By the way, every ball device needs to have at least one eject target since the balls have to go somewhere. If you don't explicitly add *eject_targets:* to a ball device config, then MPF assumes that device ejects to the playfield. This is why we don't have to add an *eject_targets:* setting to the plunger.)

Now your trough device should look like this:

```
ball_devices:
  bd_trough:
    ball_switches: s_trough1, s_trough2, s_trough3, s_trough4,
s_trough5, s_trough6, s_trough_jam
    entrance_count_delay: 300ms
    eject_coil: c_trough_eject
    tags: trough, home, drain
    debug: yes
    eject_targets: bd_plunger
```

(E) Verify your trough and plunger ball device settings

At this point you can go back and look at both your trough and plunger ball device settings to make sure everything looks good. Something like this:

```
ball_devices:
  bd_trough:
    tags: trough, home, drain
    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: yes
  bd_plunger:
    ball_switches: s_plunger_lane
    entrance_count_delay: 300ms
    eject_timeouts: 3s
    tags: ball_add_live
    eject_coil: c_plunger_eject
    player_controlled_eject_event: sw_launch
```

Notice that the above configuration only uses two spaces to indent each section instead of four. We did this to show you that YAML files don't always have to be indented by four spaces. It can be two or even one—it really doesn't matter. So just use whatever feels best for you.

At this point we like to run the game in software only mode just to make sure everything starts properly and that we don't have any typos. You don't even need to launch the media controller for this, so you can just launch the MPF core engine like this:

```
python mpf.py your_machine -v -x
```

You can quit (via the Esc key) once everything is settled. If you scroll through the log files you should see information about both your trough and plunger, as well as a bunch of other things going on that we don't have to worry about yet.

(F) Add a keyboard entries

If you're keeping your keyboard shortcuts up to date, you might also want to create a keyboard entry for your plunger lane switch. Like the entry for your trough switches, you'll want to include `toggle: true` so you don't have to hold down the key constantly to simulate a ball being in the plunger.

At this point the keyboard layout is getting confusing, so who knows what key is best for the plunger lane. Maybe P? So you could make an entry like this to the `keyboard:` section of your config file:

```
p:
  switch: s_plunger_lane
  toggle: true
```

If you have a launch button for a coil-fired plunger, add that too:

```
L:
  switch: s_launch_button
```

Note that the launch button switch is not a toggle switch, and also notice that we add an uppercase letter "L". The case of letters for keys doesn't matter, but since a lowercase L and the number 1 look similar, we decided to add "L" in uppercase.

At this point we're really close to being able to play a game! Next is to create a start button (and a launch button if you have a coil-fired plunger), add a few more ball options, and we're off and running 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 available in the MPF package at `<your_mpf_root>/machine_files/tutorial/config/step9.yaml`. (You need to rename this file to `config.yaml` to use it.) And remember if you're using physical hardware, your coil and switch numbers will be different than the ones in the sample file, since you'll need to configure them based on your driver boards' actual inputs and outputs.

Step 10. Add a 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.

(A) 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:

```
s_start:
  number: 10
```

(B) Add a "start" tag to your Start button

Just like the special-purpose tags we used when configuring our 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:

```
s_start:
  number: 10
  tags: start
```

(C) 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:

```
s:
  switch: s_start
```

(D) 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 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:

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

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

```
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, drop targets, 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 available in the MPF package at `<your_mpf_root>/machine_files/tutorial/config/step10.yaml`. (You need to rename this file to `config.yaml` to use it.) And remember if you're using physical hardware, your coil and switch numbers will be different than the ones in the sample file, since you'll need to configure them based on your driver boards' actual inputs and outputs.

Step 11. 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.

(A) 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 DMD will continue to display "PRESS START" 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:
  mode_attract_started:
    type: text
    text: PRESS START
    slide_priority: 10
  ball_started:
    type: text
    text: PLAYER %number% BALL %ball%
    slide_priority: 10
```

(B) 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 do that, 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.)

```

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

```

(C) Running your game with physical hardware

If you have a physical machine attached, go ahead and run your game without the `-x` command line option. If you're using Windows, we *highly recommend that you do not do verbose logging to the console window*, since Windows' console screen updates are *very* slow and this will screw up your game. (It's crazy. Console logging on Windows is so slow that you might find your hardware controller's watchdog timer expires and your whole machine turns off while Windows chugs along spitting things out to the screen!)

So if you're on Windows, and you're using the batch file, launch your game like this:

```
mpf your_machine -v
```

The `-v` (lowercase 'v') option still writes verbose logging information to the log file while just putting basic logging information on the screen.

If you're on Mac or Linux, you can run it with verbose console logging too, like this (both lowercase 'v' and uppercase 'V'), but of course you'll need to have two windows open and will have to launch both the MPF core engine and the media controller separately.

Launch the media controller first:

```
python mc.py your_machine -v -V
```

Then launch the MPF core engine:

```
python mpf.py your_machine -v -V
```

Make sure you have at least one ball in the trough and then run your game. The DMD should display "PRESS START." Hit the start button. A ball should be kicked out of the trough and into the plunger lane, and the DMD 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 DMD 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 *pulse_ms*: setting in the config file.
- If you launch your game code 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 your config file.

(D) "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.

Before you do this

Doing so is somewhat complex, but it is technically possible. Here's how you'd do it:

1. Once you launch MPF and the media controller and all the stuff stops scrolling by in the logs and you see the "PRESS START" message on the DMD, push the "S" key to start a game, and wait for all the stuff to scroll by. At this point MPF will eject a ball from the trough to the plunger
2. If you have a coil-fired plunger, push the "L" key (or whatever key you mapped to your launch button) to launch the ball. You should see the plunger lane switch deactivate based on the coil firing thanks to the smart virtual platform.
3. 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.
4. Now you can "flip" with the "Z" and "/" keys.
5. 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.
6. Repeat until you're bored.

7. After Ball 3 is over you can push "S" again to start another game.
8. Congrats! You just played your first virtual pinball game. Yeah, it's boring right now and kind of hard to understand, but it worked! (Later we'll show you how to write automated scripts to "play" the game for you. :)

(E) Look at your log files

Assuming everything went correctly, now let's look at the log files to see what actually happened. Remember that you'll actually have two log files—one from the MPF core and one from the media controller. These will be the two newest files in your `<mpf project root>/logs` folder. (One of them has "mpf" in the name and the other has "mc" in the name.) Just take a look through it to start to get a feel for everything that MPF is doing behind the scenes.

(F) 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:

- No ball in the trough.
- 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 *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? You can check out a complete config file in the next section.

If you're still having problems, feel free to post to the [MPF Users forum](#) on this site. If you do post, please run MPF and the media controller with verbose logging enabled (via the `-v` lowercase "v" option), and post your log files as well as your config file to the forum. (Again, your log files will be automatically created in `<your mpf project root>/logs` folder.)

Sample config file which should look similar to yours

The config file below is the real one we use for the *Demolition Man* machine connected to a P-ROC, and if you've been following all the steps in this tutorial, it should look pretty close to yours. However, unless you're following this tutorial with an actual *Demolition Man* and a P-ROC, you'll have some differences in your config file, including:

- Your hardware and driver boards might not be P-ROC, and your driver boards might not be `wpc`.
- If you're using FAST hardware, you'll have a `fast:` section in your config.
- Your `number:` settings for all your switches and coils will be your actual hardware numbers and not the numbers for *Demolition Man* from this file.
- Your flippers might be configured for single-wound coils instead of dual-wound (main + hold) like in this file.
- Your trough might have fewer switches.
- Your plunger lane might not have a coil-fired eject, which also means you might not have a launch button or `player_controlled_eject_tag`.
- Your plunger lane might not have a switch which is activated when a ball is in it, meaning it won't be configured as a ball device.
- Your trough might be a Williams System 11 or early Williams WPC style which would be configured as two separate ball devices.

```
#config_version=3

# Config file for Step 11 of our step-by-step tutorial.
# https://missionpinball.com/docs/tutorial/

# WARNING: The switch and coil numbers in this configuration file are for a
Demolition Man machine.
# Do not use this file with your own hardware unless you change the coil
and switch numbers to match your actual
# hardware!

switches:
  s_left_flipper:
    number: SF4
  s_right_flipper:
    number: SF2
  s_trough1:
    number: s31
    label:
    tags:
    type: NC
  s_trough2:
    number: s32
    label:
    tags:
    type: NC
  s_trough3:
    number: s33
    label:
    tags:
    type: NC
  s_trough4:
    number: s34
    label:
    tags:
```

```

    type: NC
s_trough5:
  number: s35
  label:
  tags:
  type: NC
s_trough_jam:
  number: s36
  label:
  tags:
  type: NC
s_plunger_lane:
  number: s27
s_start:
  number: s13
  tags: start
s_launch:
  number: s11
  tags: launch
s_right_inlane:
  number: s17
  tags: playfield_active

coils:
  c_flipper_left_main:
    number: FLLM
    pulse_ms: 25
  c_flipper_left_hold:
    number: FLLH
  c_flipper_right_main:
    number: FLRM
    pulse_ms: 25
  c_flipper_right_hold:
    number: FLRH
  c_trough_eject:
    number: c01
    pulse_ms: 25
  c_plunger_eject:
    number: c03
    pulse_ms: 25

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

dmd:
  physical: yes
  width: 128
  height: 32

window:
  elements:
    - type: virtualdmd

```

```

width: 512
height: 128
h_pos: center
v_pos: center
pixel_color: ff6600
dark_color: 220000
pixel_spacing: 1
- type: shape
  shape: box
  width: 516
  height: 132
  color: aaaaaa
  thickness: 2
- type: text
  font: tall title
  text: MY AWESOME GAME
  h_pos: center
  v_pos: top
  y: 60
  size: 100
  antialias: yes
  layer: 1
  color: ee9900

slide_player:
  mode_attract_started:
    type: text
    text: PRESS START
    slide_priority: 10
  ball_started:
    type: text
    text: PLAYER %number% BALL %ball%
    slide_priority: 10

keyboard:
  z:
    switch: s_left_flipper
  /:
    switch: s_right_flipper
  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_lane
    toggle: true
  s:
    switch: s_start

```

```

L:
  switch: s_launch
q:
  switch: s_right_inlane

ball_devices:
  bd_trough:
    tags: trough, home, drain
    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: yes
  bd_plunger:
    ball_switches: s_plunger_lane
    entrance_count_delay: 300ms
    eject_timeouts: 3s
    tags: ball_add_live
    eject_coil: c_plunger_eject
    player_controlled_eject_event: sw_launch

virtual_platform_start_active_switches:
  s_trough1
  s_trough2
  s_trough3
  s_trough4
  s_trough5

```

Once you've got everything working, congrats!

Step 12. Add the rest of your coils & 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.

(A) Add the rest of your switches

This step is pretty simple. 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, drop 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. 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.

(B) 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 for a *Demolition Man* machine, it's available in the MPF package at `<your_mpf_root>/machine_files/tutorial/config/step12.yaml`. (You need to rename this file to `config.yaml` to use it.) And remember if you're using physical hardware, your coil and switch numbers will be different than the ones in the sample file, since you'll need to configure them based on your driver boards' actual inputs and outputs.

Step 13. 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
    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: yes

  bd_plunger:
    ball_switches: s_plunger_lane
    entrance_count_delay: 300ms
    eject_timeouts: 3s
    tags: ball_add_live
    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
```

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.

Step 14: Add slingshots & pop bumpers

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 `pulse_ms:` setting in the [coils: section](#) of your config.

As in previous steps, if you want to see a complete `config.yaml` file up to this point, it's available in the MPF package at `<your_mpf_root>/machine_files/tutorial/config/step14.yaml`. (You need to rename this file to `config.yaml` to use it.) And remember if you're using physical hardware, your coil and switch numbers will be different than the ones in the sample file, since you'll need to configure them based on your driver boards' actual inputs and outputs.

Step 15: 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 we've been doing has been machine-wide configuration which was stored in the `machine_files/your_machine/config` folder. Now we're going to add a `modes` folder and then a sub-folder for each mode. Then in each mode's folder, we can create mode-specific configurations.

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.

In MPF you 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.)

(A) 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.

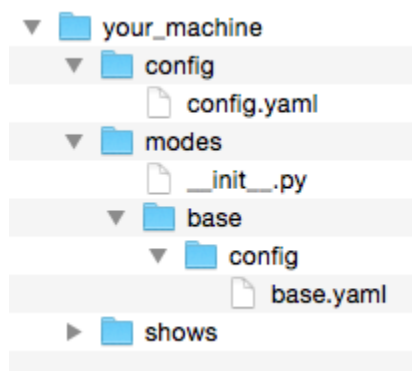
(B) Set up the folders & files for your "base" mode

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 game 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 DMD 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 an empty file in that folder called `__init__.py`. That's two underscores, then `init`, then two more underscores, then `.py`. That file can literally be blank, but it just has to have that name. (What is `__init__.py`? It's a marker file to tell Python that a folder might contain code it has to run. MPF needs it to allow you to add custom code to your modes folder. Just add it now and forget about it and all is well. [More info here.](#))
3. Create a subfolder 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.)
4. Inside your `base` folder, create a folder called `config`. (This folder will hold your mode-specific config files.)
5. 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:



(C) Add your base game mode's settings to its config file

The settings that control a mode are configured 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.

To do this, open your new mode's `base.yaml` in your code editor. Add your `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:

```
#config_version=3
mode:
  start_events: ball_starting
  priority: 100
```

The various settings options for your `mode:` section are detailed in the [configuration file reference](#). 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.

(D) 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 it 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 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.)

(E) 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:

```
INFO : Mode.base : Mode Started. Priority: 100
INFO : ModeController : +===== ACTIVE MODES =====+
INFO : ModeController : | base : 100 |
INFO : ModeController : | game : 20 |
INFO : ModeController : +-----+

```

As you can probably imagine, the *ACTIVE MODES* section lists the game that are currently active, as well as their priorities. This list will automatically reprint in the log any time a mode starts or stops.

(F) 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 DMD so that it shows the player's score. 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:
  start_events: ball_starting
  priority: 100

slide_player:
  mode_base_started:
    - type: text
      text: "%score%"
      number_grouping: true
      min_digits: 2
      v_pos: center
      transition:
        type: move_in
    - type: text
      text: PLAYER %number%
      v_pos: bottom
      h_pos: left
      font: small
    - type: text
      text: BALL %ball%
      v_pos: bottom

```

```
h_pos: right
font: small
```

We briefly touched on the `slide_player:` functionality earlier in this tutorial. (Remember that each sub-entry here will listen for an MPF event with that name and then show its content on the display.) So what's happening here is two things:

MPF's mode controller posts an event called `mode_<mode name>_started` and `mode_<mode name>_stopped` whenever a mode starts or stops. 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 text will print whenever you start a ball.)

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:` entries under `mode_base_started:`, you'll see that it shows the player's score, the player number, and the ball number. Note that those entries have words between percentage signs. Words between percentage 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. (We'll dig into this more later.)

Also note that the text: `"%score%"` entry has quotes around it. That's a YAML thing because YAML values can't start with a percent sign. So we wrap it in quotes.

Finally, notice that there's a [move in transition](#) which will move this slide in from the top of the display. (Check out the configuration file reference for [full details about how the slide_player: configuration works.](#))

Now run your game and press start and look at what happens:

<https://www.youtube.com/watch?v=v1DGd8exQIs>

(G) 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:
  mode_attract_started:
    type: text
    text: PRESS START
    slide_priority: 10
```

(H) Troubleshooting 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 (B). (A common mistake is to put `base.yaml` in the `base` folder rather than the `base/config` folder.)

Step 16: Add scoring to your base mode

By now you have a "playable" game with a base game mode, and you've added a score display to the DMD, 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.

(A) Understand in "scoring" works in MPF

MPF includes a system module called the [Score Controller](#) 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 Score Controller 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.

Even though it's called the *score* controller, the score controller 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 *scoring:* section of a mode's configuration file.

The *scoring:* section is only valid in a mode's config file (e.g. all scoring events are tied to modes). You cannot add a *scoring:* section to your machine config file.

(B) Add a *scoring:* section to your *base.yaml* config file

The first step is simply to add a *scoring:* 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 *scoring:*, like this:

```
scoring:
```

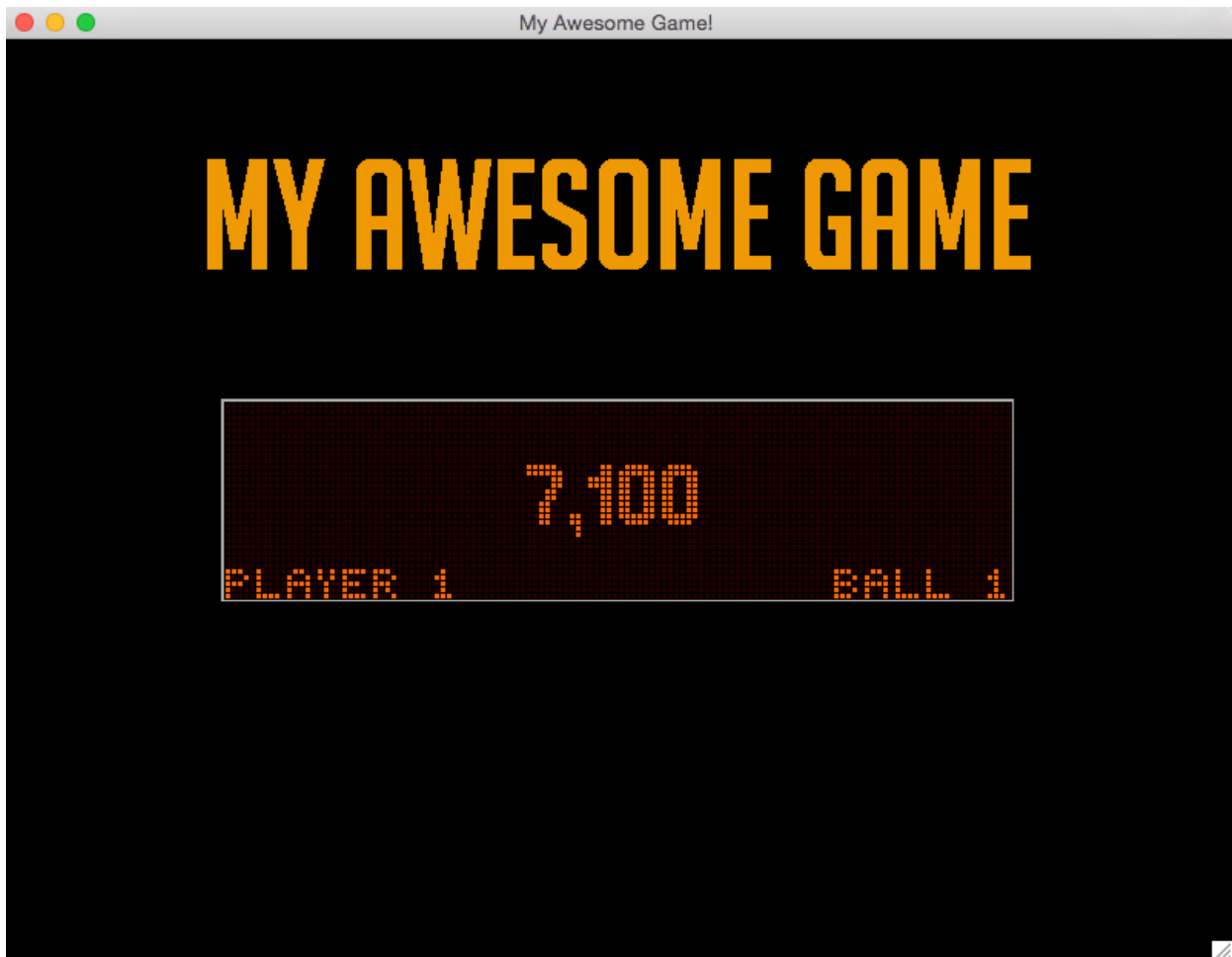
(C) Add point values for events

Then inside the *scoring:* 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 *scoring:* section of your config file, with some switch name followed plus *_active*, like this:

```
scoring:
  s_right_inlane_active:
    score: 100
  s_left_flipper_active:
    score: 1000
```

Now save your config, start a game, hit the "L" key to launch a ball and then hit the "Q" key to trigger the right inlane switch . You should immediately see a score of 100 points, and 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:



Right inlane: 100 points. Hitting the *left flipper* seven times: 7,000 points.

The `slide_player:` entries in MPF that contain player variables (remember the `%score%` text entry?) will automatically update themselves whenever the player variable changes.

At this point you can add different tags to different switches and then add separate entries for each of them in your `scoring:` section to assign different point values to each switch.

When you create more modes in the future, you can actually configure that a score event in a higher-priority mode "blocks" the 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. (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.)

Step 17. Create an attract mode DMD slide show

Now that we have a running game and some basic scoring, let's continue to make the DMD more useful.

(A) Create an attract mode DMD loop

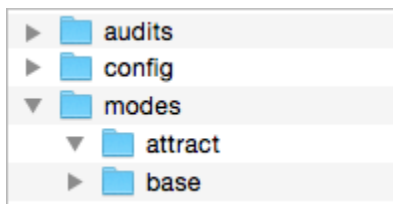
First, let's create a slide show that plays during the attract mode and cycles through a few different slides. ("GAME OVER", "PRESS START", ... that sort of thing.)

(1) Create an attract mode folder structure

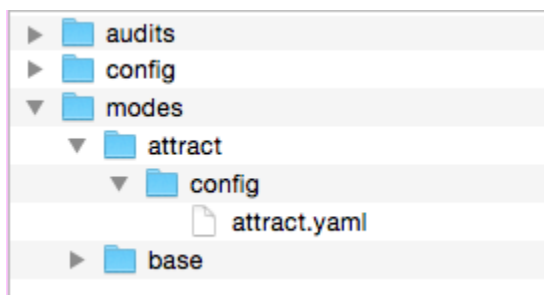
So far it looks like your game only has one mode. (The *base* mode you created a few steps ago.) But MPF actually has a few built-in modes that it uses to do its thing. For example, there's a mode called "attract" which runs the attract mode (including watching for the start button press to start a game), 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.

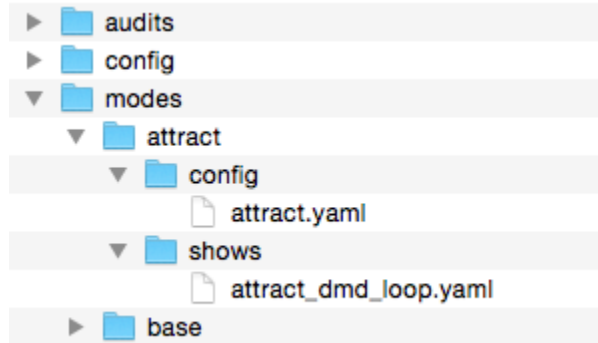
Go into your machine's modes folder (which should only have your base folder in it) and create a new folder called `attract`. Now you should see two folders in it:



Now create a config folder in your attract folder, and then create a new config file called `attract.yaml`:



Finally, create a folder called `shows` in your new `attract` mode folder, and inside that folder, create a new file called `attract_dmd_loop.yaml`:



(2) Edit your show yaml file

MPF has the ability to run "shows" which are coordinates series of lights, sounds, slides, flashers, images, videos, gi, 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_dmd_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 shows used by a mode inside that mode's `shows` folder since it keeps everything together. (In other words, by doing that we keep everything a mode needs in its own consolidated folder.)

Here's a complete sample `attract_dmd_loop.yaml` file you can use as a starting point:

```
- tocks: 3
  display:
    - type: text
      text: YOU ARE AWESOME
      transition:
        type: move_out
        duration: 1s
        direction: left

- tocks: 3
  display:
    - type: Text
      text: PRESS START
      decorators:
        type: blink
        repeats: -1
        on_secs: .5
        off_secs: .5
    - type: Text
```

```

text: FREE PLAY
color: 00ff00
v_pos: bottom
font: small
transition:
  type: move_in
  duration: 1s
  direction: right

- tocks: 3
  display:
    - type: Text
      text: MISSION PINBALL
      color: ff0000
      transition:
        type: move_in
        duration: 1s
        direction: top
    - type: shape
      shape: box
      width: 128
      height: 32

```

Notice that the show file is broken into steps, each beginning with a dash and then a tocks: entry. The "tocks" entry controls the ratio of how long each step plays for. In this case we have three steps, and they're each three tocks, meaning they will each play for the same amount of time. (How long is a tock? That's up to you to set when you play the show. More on that in a moment...

Then in each step, you define what you want it to do. In this case each step of the show is a "display" step, so we enter display:, and then under that we configure settings for the display slides. Note that these settings are the exact same settings you use in the slide_player: section of a config file. You can define one or more display elements, and you can also define transitions that will take place.

At this point we're only using text and shape display elements, but you could also use movies and images.

(3) Configure your show to play automatically

Next we need to configure our new attract mode DMD loop show to start playing automatically when the attract mode starts. To do this, go back to the config file for the attract mode (modes/attract/config/attract.yaml) and add the following:

```

#config_version=3

show_player:
  mode_attract_started:
    - show: attract_dmd_loop
      repeat: yes
      tocks_per_sec: 1

```

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:`, 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. Notice that we also add `repeat: yes` which means the show will repeat (or loop) indefinitely, and we will add `tocks_per_sec: 1` which means this show will play at a speed of 1 tock per second. (And since each of the entries in this show file are set for 3 tocks, that means each slide will be on the DMD for 3 seconds. You can play around with different settings here. For example, try `tocks_per_sec: 30` and then don't blink! :)

You can configure `show_player:` entries to stop shows, though in this case that's not necessary because any running shows that a mode starts are automatically stopped when the mode stops, and the attract mode stops when the game mode starts, so you don't have to manually stop the show here.

(4) Remove the PRESS START slide_player entry from your machine-wide config

One last thing you should do here while you're at it is go back into the machine-wide config and remove the `slide_player:` entry for the `mode_attract_started` with that text that says PRESS START. The attract mode runs at a priority of 10, and any `slide_player:` entries in your machine-wide config run at priority 0, so that's why you don't see that text anymore, but it's still a good idea to remove it just to keep things clean.

Now when you run your game, your attract mode should look something like this:

<https://www.youtube.com/watch?v=fu5EkZELIHQ>

Step 18: Add lights or LEDs

Now that you're able to run a complete (albeit boring) game, let's get your lights configured. In this step, we're going to add your machine's lights and/or LEDs to your machine config file, and then in the next step we're going to create a basic light show that can run during your machine's attract mode.

Note: If you're following this tutorial with virtual hardware, you might want to skip these two steps since you can't really see the effects of your lights working via the console and log files. So feel free to skip to Step 17 (the one about game modes) right now and then you can come back and add lights once you have physical hardware).

Also note that throughout the MPF documentation, we tend to use the term "lights" to refer to any type of playfield light, including traditional #44 or #555 incandescent bulbs plugged into a lamp matrix, #44 and #555 LED "replacement" bulbs used with a lamp matrix, and newer LEDs or RGB LEDs directly connected to LED controllers. If we need to specifically refer to one type of light versus another, we'll make that clear.

(A) Understand the different between matrix lights and LEDs

Before you get started building the configuration for your lights, you have to know whether you'll be configuring matrix lights or LEDs. This is tricky, because you can use LEDs with lamp matrixes, so just because you're using LEDs doesn't mean you configure MPF for LEDs!

Umm... what?

Ok, let's take a step back.

There are two types of lighting systems for pinball machines: lamp matrixes and direct-connected LEDs. All commercial pinball machines from about 1979 through 2012 (give or take) used lamp matrixes (typically with 8 rows and 8 columns of lights). Historically these were used with incandescent light bulbs, (#44, #555, etc.), though 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) via the `matrix_lights:` section of your machine config file. This will apply to you if:

- You're using a Multimorphic P-ROC or FAST WPC controller to write your own game software for an existing Williams System 11, WPC machine, Stern S.A.M, or Sega/Stern Whitestar machine.
- You're using a Multimorphic P-ROC or P3-ROC with a PD-8x8 lamp matrix controller.

Alternately, if you have directly-controlled LEDs (i.e. no lamp matrix), whether single color or RGB, then you'll configure them in the `leds:` section of your machine config file. This will apply to you if:

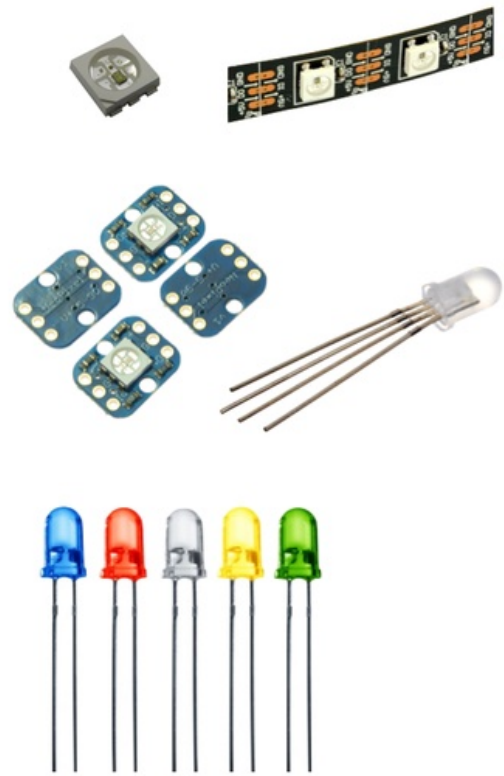
- You're using a Multimorphic PD-LED board to control your LEDs.
- You're using serial RGB LEDs plugged into the RGB LED ports of a FAST controller.
- You're using any type of "[Neopixel](#)" RGB LED

Another way to tell which kind of lights you're is if your lights or LEDs have mini bayonet or mini wedge bases, they're *Matrix Lights*, and everything else is *LEDs*. Here's a diagram that might help:

“MatrixLights”



“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.

(B) Add your lights to your machine config file

Once you figure out what kind of lights you have, 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. For matrix lights, follow the directions for your specific setup as detailed in the [matrix_lights: section](#) of the configuration file reference. If you have LEDs, follow the instructions and the [leds: section](#).

There's nothing too special here. You can apply tags to lights which you can use perform actions on a group (such as, *turn on all the lights tagged with "red,"* or *turn off all lights except those tagged with "mini_playfield,"* though it's simple to add tags later so you don't really have to do it now.

Also keep in mind that the specific format you use to specify which lights are connected to which controller outputs will vary depending on whether you have Multimorphic or FAST pinball controllers and whether you're using OEM or Williams WPC driver boards. Again, just follow the directions in the config reference and [post to the forum](#) with questions.

Here's a small part of the `matrix_lights:` section of our machine configuration file for *Demolition Man*. Notice again that we preface each light with `l_` to make it easier for our code editor's autocomplete function to find light names in the future.

```
matrix_lights:
  l_ball_save:
    number: 111
    label:
    tags:
  l_fortress_multiball:
    number: 112
    label:
    tags:
  l_museum_multiball:
    number: 113
    label:
    tags:
  l_cryoprison_multiball:
    number: 114
    label:
    tags:
  l_wasteland_multiball:
    number: 115
    label:
    tags:
  l_shoot_again:
    number: 116
    label:
    tags:
```

Step 19: Create an attract mode light show

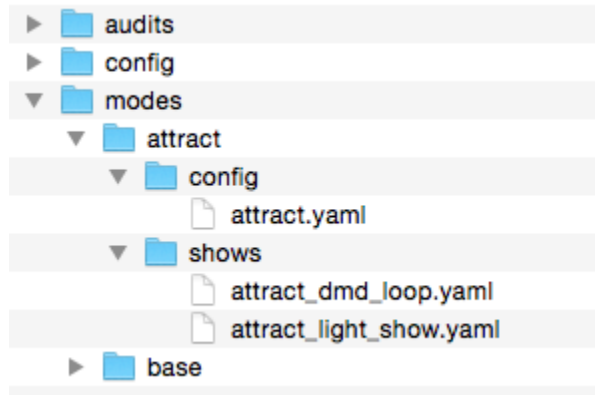
Once you add your lights or LEDs, you need a simple way to test them to make sure they're working. We typically throw together a quick attract mode light show so we can see some blinking lights as soon as MPF boots up.

(A) Create a light show YAML file

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 DMD loop, except this time we configure lights or LEDs for each step instead of display elements.

Show entries for lights and LEDs are very similar, except with LEDs you specify full RGB values whereas with lights you just specify whether they're on or off.

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 machine folder should now look like this:



(B) Add some entries to your light 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 *light1*, *light2*, and *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:

```
- tocks: 1
  lights:
    l_light2: 0
    l_light1: ff
- tocks: 1
  lights:
    l_light1: 0
    l_light2: ff
- tocks: 1
  lights:
    l_light2: 0
    l_light3: ff
- tocks: 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 *light2* and turns on *light1*, the next one turns *light2* and turns off *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:

```
- tocks: 1
  leds:
    l_led1: ff0000
    l_led2: ff0000
    l_led3: ff0000
- tocks: 1
  leds:
    l_led1: ff6600
    l_led2: ff6600
    l_led3: ff6600
- tocks: 1
  leds:
    l_led1: ffcc00
    l_led2: ffcc00
    l_led3: ffcc00
- tocks: 1
  leds:
    l_led1: 00ff00
    l_led2: 00ff00
    l_led3: 00ff00
- tocks: 1
  leds:
    l_led1: 0000ff
    l_led2: 0000ff
    l_led3: 0000ff
- tocks: 1
  leds:
    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.)

(C) Configure your light show to start by itself

Once you create the show, it just sits there, existing but not playing. You have to tell it when to play and when to stop. You can do that in the `light_player:` section of your machine config file. (Notice this is different than the `show_player:` entry for the display show.)

Create that entry like this:

```
light_player:
  mode_attract_started:
    - show: attract_light_show
      repeat: yes
      tocks_per_sec: 1
```

Like the DMD show, notice we also have the setting `tocks_per_sec: 1`. Also notice the `tocks: 1` in each step of the `attract_light_show.yaml` file. What's going on here is that MPF has the ability to play back your light shows at any speed. (In other words, the playback speed of a show is controlled when the show is played, not when the show is built.) This is cool because it means you can do things like making a show play faster and faster without having to create a bunch of different shows for each playback speed.

In this particular case, each step of the light show is configured for 1 tock, so each step will be equal duration. If you want your light show to sit at one step longer than another, you could change the `tocks:` setting for just that step.

So that's it! Save your machine config file, save your light show file, and run your game. You should see your light show start to play once the attract mode starts up.

(D) Configure more light shows to all run at once

Your simple little 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://www.youtube.com/watch?v=_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

We'd probably make every step of every show 1 tock. Then in our `light_player:` configuration, we'd configure the list of shows to play when the attract mode starts instead of just one. For example:

```
light_player:
  mode_attract_started:
    - show: flipper_red_flashing
      repeat: yes
      tocks_per_sec: 2
    - show: purple_mode_sweep
      repeat: yes
      tocks_per_sec: 4
    - show: inlane_alternating
      repeat: yes
      tocks_per_sec: 3
    - show: random_flashing
      repeat: yes
      tocks_per_sec: 2
    - show: car_chase_sweep
      repeat: yes
      tocks_per_sec: 3
    - show: ramp_orbit_sweep
      repeat: yes
      tocks_per_sec: 5
  ... (truncated. you get the idea)
```

(E) Configure the shows to stop when the attract mode stops

At this point if you start a game, you'll see that the light shows continue to run. That's because everything we set up in the light player only has instructions for when the shows should start—it doesn't say anything about them stopping.

You can stop shows in a similar way to how you start them. Create a sub-entry in your `light_player:` section with the event name, but then instead of settings to start shows, you add settings to stop them.

So add an entry for `mode_attract_stopped;` and then create sub-entries for each show. You don't need to add `repeat:` and `tocks_per_sec:` settings since we don't care about those since we're stopping the shows, but you do need to add `action: stop` to the light player knows that entry is to stop them. (The default "action" is "start", which is why you didn't have to enter an action to start your shows.)

Here's an example, again truncated:

```

light_player:
  mode_attract_started:
    - show: flipper_red_flashing
      repeat: yes
      tocks_per_sec: 2
    - show: purple_mode_sweep
      repeat: yes
      tocks_per_sec: 4
  mode_attract_stopped:
    - show: flipper_red_flashing
      action: stop
    - show: purple_mode_sweep
      action: stop

```

Step 20: Create your first shot

At this point you have a machine you can turn on, lights flash, the DMD show 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", and you're going to create your first shot.

(A) What's a shot?

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.

Many shots in MPF have lights or LEDs associated with them that indicate what "state" the shot is in. (e.g. "shoot the flashing targets" or "hit the lit ramp" could mean "hit the target that's in the *flashing* state", or "hit the ramp that's in the *lit* state".)

Shots in MPF are similar to other devices we've worked with (like autofires and ball devices) where you group together switches, lights, and/or LEDs into the things we call *shots*. (Not every shot has a light or LED associated with it.)

You can even configure shots that are 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.)

You can also group multiple shots together into a shot group. For example, you might have three lanes at the top of your playfield, each with a rollover switch and a light. In that case you'd configure each of them as a separate shot. Then you could create a shot group that

grouped all three of the shots together into a logical group. A shot group lets you do cool things, like trigger an event when all of the member shots are in the same state (increase Bonus X when all three shots are "lit"), and you can setup things like "shot rotation" that rotates the state of the member shots left or right (lane change via the flipper buttons, for example).

We'll get to that soon. For now, let's create our first shot.

(B) Create your first shot

You define your shots by adding a `shots:` section to your config file. Simple enough.

At this point you're probably aware that there are two different types of config files in your machine. There are your machine-wide config files which live in `<your machine root>/config` folder, and then there are mode-specific config files which live in `<your machine root>/modes/<mode name>/config`.

You can configure shots in either your machine-wide config or in a mode-specific config. Shots you configure in your machine-wide config exist for the entire game, and shots configured in a mode config only exist while that mode is active. What's kind of confusing is that you can also configure the same shot in both your machine-wide config and in a mode-specific config. When you do this, the machine-wide configuration for that shot is available always and acts as the default settings for that shot, and then you can further customize the behavior of a shot in a particular mode by adding settings for it in a mode config.

Our preference is to add all the physical shots on your playfield to your machine-wide config since they never change. (A ramp is a ramp is a ramp, regardless of what modes are active.) Then we customize the shots in the mode configs for the specific modes that need to do something special with them.

As you'll see as we dig deeper into shots, shots and modes are tightly integrated in MPF, and in fact a single shot can be configured in multiple modes at the same time. (The shot knows which modes it's configured for, and when it's hit it can track multiple states from multiple modes at the same time!) You can even configure shots in a mode config so that shot is "blocked" while the mode is active. (For example, maybe your pop bumpers count up towards super jets in your base game mode, but you might have a special game mode where you want the bumpers to only be used by that mode.)

We're getting ahead of ourselves a bit. Let's start by creating our first shot in the machine-wide config file. Pick a switch that has a light or LED near it to begin with. This can be a standup target, a rollover lane, or something like that.

If you have matrix lights in your machine, setup the shot like this:

```
shots:
  my_first_shot:
```

```
switch: s_standup_1
light: l_right_middle_triangle
```

If you have LEDs, you'd setup your shot like this:

```
shots:
  my_first_shot:
    switch: s_standup_1
    led: l_right_middle_triangle
```

Notice that the configuration for the shot is the same. The only difference is whether you specific a `light:` or `led:`.

(C) Test your shot

At this point you can start a game and test your shot. When the game starts, you should notice that the light associated with your shot is off. If you hit the switch associated with your shot, the light will come on. Congrats! You now have a shot!

The "state" of the shot (whether it's *unlit* or *lit*, in this case), is saved on a per-player basis. If you start a game, launch a ball, and hit your shot, you'll notice that when the ball drains the shot is still lit. (Later we'll show you how to add scoring and how to control when the shot resets.) Also notice that if you start a multiplayer game, if the first player hits the shot and lights it, then when the second player's ball starts the shot will go back to the *unlit* state since the second player hasn't lit it yet.

Shots are only active while a ball is in play, so you have to start a game in order for your shots to work. (Note that the ball counts as "in play" once it's sitting in the plunger lane waiting to be plunged, so as soon as you hit start and the trough ejects a ball, you should be able to hit the switch associated with your shot and see the light come on.)

(D) Change the shot profile

Right now your shot has two states. It starts unlit, then you hit it, then it becomes lit. Shots in MPF can have *profiles* applied to them that control how they behave, including what the lights do and what happens when you hit them.

When you build your config for your machine, you can create different profiles (each profile has a name), and then you can apply those profiles to your shots. MPF has a built-in profile called "default," and this default profile is automatically applied to shots where you don't specify that another profile should be used.

The default profile has two states. The first is called "unlit" and it sets the light to be off. The second is called "lit" and it sets the light to be on.

So what's happening behind the scenes is you setup your shot, you did not specify a profile so MPF applied the profile called default, the first step in that default profile is called unlit and has the light set to be off which is why the light starts off, then when you hit the switch for the shot, the profile is advanced to the next step which is called "lit" and which is configured to turn on the light.

Hopefully that all makes sense? :)

Let's create a new profile and apply it to this shot so you can see it in action.

(1) Create a new profile

You create your shot profiles in the `shot_profiles:` section of your config file. Like the other shot-related settings, you can add your profiles to either your machine-wide or to a mode-specific config. It really doesn't matter. We prefer to add profiles that we'll use throughout our game to the machine-wide config, and then if there's some special profile that we'll just use for one mode then we'll add that to the mode-specific config. (Profiles end up being available across the entire machine regardless of where they're configured, so don't create two different profiles with the same name.)

For now, let's create a new profile in the machine-wide config.yaml file. Start by adding a section called `shot_profiles:`

```
shot_profiles:
```

(You can read about all the profile settings in the [shot_profiles: section](#) of the configuration file reference.)

Let's create a new profile called "hit_me" which will start with the light in a flashing state, and then when the shot is hit, it will change to a solid "on" state. We would set it up like this:

```
shot_profiles:
  hit_me:
    states:
      - name: lit
        light_script: flash
      - name: complete
        light_script: "on"
```

Looking at what we just entered above, you see the profile entry for the profile named "hit_me", then you see an entry for "states", and under there you see the steps for the states. (Note that each step starts with a dash, then a space, then a "name:" entry. The name is the name of that state (or step), and under it you see the `light_script` that will be played when that state is active for that shot.

In this case, we created two states. The first we decided to call "lit" and we play a light_script called "flash", and the second state we're calling "complete" and we play a light_script called "off".

(We haven't talked about light_scripts yet. We'll get to those later, though they're configured via the config files too. You can read more about them [here](#).) MPF includes some built-in light scripts called *on*, *off*, and *flash* which is why we can use them here without defining them elsewhere in our config file.

Also note that we put the "on" script in quotes. This is an annoying YAML thing. Python's YAML processor tries to be smart about values it finds in yaml files. If it sees a value of *on*, it interprets it as a boolean "True" entry. The YAML processor converts several words to boolean True, including *true*, *yes*, and *on*. In this case we don't want our entry to be converted to `light_script: True`, rather, we want it to be `light_script: on`, so that's why we put quotes around it. It's like we're saying, "Dude, just read this as "on", don't try to be all smart and convert it to True."

Anyway, so now we have a profile defined called *hit_me* that has two states: *lit* (flashing) and *complete* (on).

(2) Apply this profile to your shot

Now go back into the config file for your base mode (`/modes/base/config/base.yaml`) and go back to the shots section we setup before. Then add a setting to that shot called `profile:` and add your *hit_me* profile, like this:

```
shots:
  my_first_shot:
    switch: s_standup_1
    led: l_right_middle_triangle
    profile: hit_me
```

(3) Test it out

Be sure to save both of your config files, and then start a game. You should see that the light for your shot is flashing (since the "flash" script is applied in that's the light_script for the first step in the *hit_me* profile). Then when you hit the switch for this shot, the light should turn on solid.

(E) Add some scoring

Remember from one of the [earlier steps](#) in the tutorial that we added a `scoring:` section to your base mode's config file. Also remember that the scoring section has entries for events that you tie scores to. So in order to add scoring to your new shot, we need to know

(not done yet... work in progress....)

5. How To Guides

We've created a bunch of "How To" guides to walk you through adding very specific things to your game. Our intention is that you'll first follow the [step-by-step tutorial](#) (in order) to get to the point where you have a very basic running game, and then you can pick-and-choose the specific How To guides you'd like to do next. (The How To guides are self-contained and can be done in any order.)

Here are the How To guides we've created so far. (If you want to know how to do something else, post a question to our forum and we'll write a how to guide for it!)

[subpages]

How To: Check your version of MPF

Since MPF is a work-in-progress, things can change from version-to-version.

You can check which version of MPF you have from the command line. To do this, run the following command from your MPF folder:

```
python mpf.py --version
```

It should print something like this:

```
Z:\mpf>python mpf.py --version
MPF v0.21.3 (config_version=3, BCP v1.0)

Z:\mpf>
```

The main thing you're looking for is what's after the "MPF" in the results it prints. (The example above is MPF version 0.21.3.)

The first two sections of that number are the major & minor release numbers. (0.21 in this example.) Those are the important numbers. The last section is the patch number which we update if we find major bugs that can't wait until the next release. The patch version doesn't really matter, so if we say you need MPF version 0.21 then it doesn't matter if you have 0.21.0 or 0.21.1 or 0.21.7 or whatever.

You can also see (and download) the most recent version here: <https://github.com/missionpinball/mpf/releases/latest>

How To: Migrate from MPF 0.21 to MPF 0.30

This How To guide walks you through updating your machine configs and code from MPF 0.21 (released in Dec, 2015) to MPF 0.30. Note that we skipped a few versions since so much changed from 0.21, which is why this is now 0.30. Also note that 0.30 is pronounced "thirty", not "three", as MPF 0.3 was released in July 2014. (We use semantic versioning, which means version number order is different than mathematical order. Details [here](#).)

This guide was last updated on **March 16, 2016**. At this point MPF 0.30 is not actually released. So this documentation currently applies to the pre-release "dev" version. Feel free to use it to start playing with MPF 0.30 today, but know that not everything is done yet for MPF 0.30, so some things do not work yet.

This document is written to work with the following versions of MPF:

- MPF core engine 0.30.0.dev700 <-- yeah, there have been 700(!) commits since MPF 0.21.
- MPF Media Controller 0.30.0.dev270

You can view the latest version of MPF by running `mpf --version` from a command prompt. (If this command gives you an error, that means you're not on MPF 0.30.)

What's still missing in MPF 0.30.0.dev700

This guide has been written for a pre-release version of MPF 0.30. There are currently several things missing and things that we still need to finish for MPF 0.30. Here's a list of what does not work currently that we are planning on finishing for the final release of MPF 0.30:

- Shows don't automatically stop when modes end
- Slide and widget expiration time
- Show stopping needs to be tested to ensure that lights and LEDs are properly restored
- Priorities of things launched from shows (LEDs, lights, sounds, slides, etc.) and modes are not set yet.
- The "move out" transition doesn't work
- Physical DMD and physical RGB DMD's are not there yet. (You can still display virtual versions of them in on-screen windows, it's just that connecting to physical ones isn't done yet.)
- Entered_chars and character_picker display widgets aren't written. (This also means that high score entry modes will crash since they need them.)

- Fonts have to be installed in your system in order to work. (Before we release we'll fix it so they can also be in a *fonts* folder in your machine folder.)
- The settings for cropping the tops and bottoms of fonts are not written yet, so vertical placement of fonts will not be exactly right at the moment.

Please read this entire guide as a lot has changed. Here are the steps to migrate to 0.30:

(1) Review the Changes in MPF 0.30

We've posted a [release notes guide with a list of changes to MPF 0.30](#). Before you start your migration to MPF 0.30, you should read that guide to see what's new.

(2) Installation for Windows

Note that these instructions are only for Windows. If you're on a Mac, skip to Section 3.

(2A) Install Python 3

Since MPF 0.30 requires Python 3, the first thing you need to do is install Python 3. On Windows you should use Python 3.4.4, because Kivy does not support Python 3.5 on Windows at this time.

If you're on 64-bit Windows, install the 64-bit version of Python. 32-bit should use 32-bit. The easiest way to tell whether you have 32-bit or 64-bit Windows is to open a command prompt and run (note this is case-sensitive):

```
echo %PROCESSOR_ARCHITECTURE%
```

If it prints "x86", that's 32-bit. If it prints "x64", that's 64-bit.

- You can download the Python 3.4.4. x86 / 32-bit MSI installer [here](#).
- You can download the Python 3.4.4 x64 / 64-bit MSI installer [here](#).

Python 3 and Python 2 can run side-by-side no problem. If you just installed Python for use with MPF, then you can first just remove the old version of Python (via Add/Remove Programs). This will eventually be our recommendation, but since MPF 0.30 is not done yet, you may want to keep Python 2 and MPF 0.21 around for awhile.

Then install Python 3.4.4. On the "Customize Python 3.4.4" screen, we like to select the option "Add python.exe" to Path. That way you can run Python from anywhere.

If you have both Python 3 and Python 2 installed side-by-side, then every Python 3 command will have a "3" added to it. (So you run "python3" instead of "python", "pip3" instead of "pip", etc.)

Once Python is installed, open a command prompt and type `python --version <enter>`. It should print *Python 3.4.4*. If you get a message about how 'python' is not recognized, log out and log back in. (You have to do this when running a non-admin command prompt so it can update the system path setting.) If you get a message that says you have Python 2.x.x, then try `python3 --version`.

(2B) Install MPF

Now open a command prompt and run this (type this, then press <enter>):

```
pip install mpf-mc
```

If you have both Python 2 and Python 3 installed on your system, you need to use the Python 3 version of this command, which is:

```
pip3 install mpf-mc
```

Pip is the name of the Python Package Manager. This command is telling pip to install a package called "mpf-mc", which is the *Mission Pinball Framework - Media Controller* package. When you run this, pip will connect to the internet to the [Python Package Index](#) (called "PyPI") where the *mpf-mc* is registered as a package. Pip will download MPF-MC, as well as all the prerequisites for MPF-MC, and then install them onto your computer.

Note that the MPF-MC package is technically the package for the MPF Media Controller, however, MPF-MC includes MPF in its list of prerequisites, meaning if you install MPF-MC then you also get MPF. So installing MPF-MC right off the bat is just a shortcut to install both MPF and MPF-MC in a single step.

(2C) Install the fonts

The current preview build of MPF does not support using fonts from the MPF *fonts* folder. So if your machine used any of the built-in MPF fonts, you need to install them to your system now. You can find them in the [old MPF 0.21 \(master\) repository](#), in the *mpf/media_controller/fonts* folder. Just double-click each one to install it.

(2D) Download the mpf-examples so you can run the demo_man test

One of the changes we made in MPF 0.30 is we removed all the example machine code from the MPF package. We figured there was no need to include all the examples in every single MPF installation. In this case, however, we want the examples, so download a zip file containing the latest mpf-examples from [here](#). (Note that if you have git, you can simply clone the [mpf-examples repository](#) so they'll always be up-to-date. Just make sure you get the master/dev to match which branch of MPF you're using. If you have no idea what "git" means, that's fine, just ignore that. :)

You can download the mpf-examples zip file and expand it to wherever you want. Once you have it expanded, open two command prompt windows to the *demo_man* folder in mpf-examples, for example:

```
c:\mpf-examples\demo_man\
```

Then from within the first one, run:

```
mpf mc
```

And from within the second one, run:

```
mpf
```

If you see the *demo_man* window with a DMD in it pop up, you're good to go. You can play around with this if you want. Click on the popup window so it has focus, and then the `s` key starts a game and adds players, the `x` key is a slingshot which will give points, and the `1` key will drain the ball. It will probably crash when the game ends since we don't have the *character_picker* and *entered_chars* display widgets done that the high score entry mode needs.

Now skip to Step (4) below

(3) Installation for Mac OS X

We don't have a procedure ready yet for the Mac. Actually we have the MPF package for Mac done (it's precompiled and everything), but at the moment we've having issues getting Kivy installed for Mac in a repeatable way. (See [here](#) and [here](#).) Ultimately we're planning to create a proper Mac MPF.app in a DMG file, so we'll just proceed with that. Until then, sorry Mac users. (Which, btw, is 2 out of the 4 core MPF devs. :(

(4) Migrate your machine files

One of the big changes in MPF 0.30 is that MPF is installed into the central Python site-packages folder. With old versions of MPF, you downloaded and worked within the MPF folder itself. That doesn't happen anymore, as the MPF folder is essentially hidden away inside your computer.

So instead, you now work within your actual machine folder.

Since you're migrating from MPF 0.21, you probably have an mpf/machine_files folder (or something like it, with your machine inside there. So the first thing to do is to move or copy your own machine folder to some location that's not part of the old MPF installation.

Then you can delete the old MPF folder.

(4A) Migrate your machine's config and show files

Now open a console window (Command Prompt on Windows, or Terminal on Mac) and switch to your machine's root folder. You want to be in the folder of your machine, that contains subfolders like config, shows, modes, etc. (e.g. you're not in your config folder, you're in the folder containing your config folder.)

Now you'll run the MPF migration tool which will scan through your machine folder and migrate any YAML configuration and show files it finds.

To do this, on windows, just run:

```
mpf migrate
```

The migrator should migrate everything for you.

You'll notice that it creates a folder called "previous_config_files" in your machine folder, and under there is a folder with the time and date stamp, and then under that is your old folder structure with all your original YAML files in their original locations.

You'll also see a "logs" folder (in MPF 0.30, logs are stored in your machine folder instead of in the MPF folder since MPF is now in a central installed location.) You should see a log for the migration which will show you details of everything the migrator did.

As far as we know, the migrator does everything and there are no known bugs. (We tested it on 6 or 7 different machine configs.) But really who knows? If it crashes or does something weird, we can take your config files and test them ourselves and get the migrator fixed and/or manually fix your files.

The only weird thing about the migrator is we couldn't figure out how to add spaces between new sections that are added to existing config files. So while everything the migrator generates will be syntactically correct, you might need to go in and clean things up visually a little bit.

Also the migrator exports files with four spaces for indentation. We'll add an option soon to let you specify if you want to use two spaces, so if that's the case, hold off for a few days.

(4B) Update any custom mode code or scriptlets

If you have any custom Python code as part of your machine (either in mode code or in scriptlets), you'll need to migrate that code too.

The first part of the migration is to convert your code to Python 3. Python 3 includes a tool called "2to3" which you can use for this. You might be nervous to run an automated tool to convert your code, but you should be fine. We used the same 2to3 tool for the 15k lines of code in the entire MPF package when we converted MPF to Python 3, and there were only 4

little things we had to go back and fix manually. And most likely everything you're doing in your mode code or scriptlets is pretty straightforward and should migrate easily.

Run this tool from the command line, from the root of your machine folder, like this:

On Windows:

```
2to3 -w .
```

The tool is called "2to3", the -w means you want to "write" your changes, and the dot means to use your current folder.

Also you'll need to do a "find and replace" for your all your code. If you're using PyCharm or Atom, it's just SHIFT+CTRL+F (or SHIFT+CMD+F), so it should just take a second.

Old value (find): `mpf.system`

New value (replace): `mpf.core`

It's possible you might need to change some other things in your code too. If so, just post to the dev forum. You can just try to run your game and see what happens.

Also, you can remove all those empty `__init__.py` files you had to add to your machine folder and scriptlets and mode code folders, as Python 3 does not require them.

Here's an overview of this entire migration process in action from the *demo_man* sample machine:

<https://www.youtube.com/watch?v=GrRnYFGsIL8>

(5) Run your game and see what happens!

Now you're ready to run your game. To do this (for now), open two command windows. Then from within your machine folder (the same folder you ran the migration utility from), in the first window run:

```
mpf mc
```

This will start the media controller. In the second window, run:

```
mpf
```

This will start MPF.

Then cross your fingers and hope it doesn't explode! All of the test games we tried now work except for two which have a lot of custom code that will need to be ported manually. (In both

cases we offered to do this for the game creators, since it's faster for us to do it rather than explain what to do. :)

(6) Next Steps

At this point you can feel free to start editing your config files and playing with MPF 0.30. We understand that with no documentation yet (apart from what's in the [release notes](#)), there's probably not much you can do. And again, if you have problems, post to the forum. We want to make this process as painless as possible, and will help you get everything converted over.

The good news is the config file and show file formats are finalized for MPF 0.30, so even though not everything works yet, you can start working with your config and show files now with the confidence that they won't change between now and the final release.

How To: Use a P-ROC with MPF

This how to guide explains how to setup your MPF configuration files to interface with a Multimorphic P-ROC pinball controller. Most of this information is available elsewhere in the documentation, but this guide pulls it all together in one place.

This guide applies in cases where you're using a P-ROC to control an existing machine (either a Williams WPC, Williams System 11, Stern Whitestar, or Stern S.A.M. machine) and also when you're using a P-ROC with the "P-ROC driver boards" (the PD-16, PD-8x8, and/or the PD-LED) to control a custom machine built from scratch.

(A) Installing the P-ROC drivers on your computer

In order to use a P-ROC with MPF, you need to have the P-ROC software libraries installed on your computer. These are called *libpinproc* and *pypinproc*. (*libpinproc* is the actual interface library, written in C, and *pypinproc* is a wrapper for that library that makes it available to Python programs such as MPF.)

The MPF all-in-one installer for Windows or Linux will install the necessary drivers for you. (If you don't want to use the all-in-one installer, you can still follow the [platform-specific links](#) for installing MPF which have the P-ROC software libraries pre-compiled and ready to copy to your computer.)

(B) Configuring your platform settings for a P-ROC

To use MPF with a P-ROC, you need to configure your platform as `p_roc`, like this:

```
hardware:
  platform: p_roc
  driverboards: wpc
```

You also need to configure the `driverboards:` entry for what kind of driver boards you're controlling: `wpc` for Williams WPC, `pdb` for P-ROC driver boards like the PD-16 and PD-8x8, etc. (See the [hardware:](#) section of the configuration file reference for the full list.)

(C) Configuring switches

For switches, you can use all the settings as outlined in the [switches:](#) section of the config file reference.

Number

If you're using your P-ROC in you own custom machine with the switch matrix headers connected directly to your P-ROC, the number format to use is column/row (starting with zero). For example, the switch connected to Column 0, Row 4 would be entered like this:

```
switches:
  some_switch:
    number: 0/4
```

If you're using your P-ROC to control an existing machine (like a Williams WPC or System 11 machine), then you can enter the switches based on their numbers in the operator's manual. Details about that are in the [switches:](#) section of the config file reference.

Debounce

The P-ROC doesn't support variable debounce times—it's either on or off. So with P-ROC controllers, a debounce value of `True` (or any non-zero integer) means that debounce is enabled for that switch, and a value of `False` (or `0`) means that debounce is disabled.

(D) Configuring coils

There are a few things to know about controlling drivers and coils with the P-ROC.

Number

The first is you need to know the number for each coil. Like switches, this varies depending on whether you're using your P-ROC with an existing machine or with a new machine and a PD-16.

For PD-16-based coils, the coil numbering format is *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 (0-7)

- y : Bank address (0 for A, 1 for B)
- z : Output number (0-7)

For the numbering of coils in existing machines (WPC, etc.), the P-ROC uses the same numbering as any other controller on those systems, so you can read how to do that in the [coils:](#) section of the config file reference.

Hold Power

If you want to hold a driver on at less than full power, the P-ROC does this by using "on time" and "off time" parameters, in milliseconds, that it switches back-and-forth.

If you set a `hold_power:` entry (which is a power value from 0 to 8), then the following actual settings will be applied with a P-ROC:

- 0: solid off
- 1: 1ms on / 7ms off
- 2: 1ms on / 3ms off
- 3: 3ms on / 5ms off
- 4: 1ms on / 1ms off
- 5: 5ms on / 3ms off
- 6: 3ms on / 1ms off
- 7: 7ms on / 1ms off
- 8: solid on 100%

If you want more precise control over the timing of how your P-ROC drivers will be enabled, you can use the optional settings `pwm_on_ms` and `pwm_off_ms`.

For example, to configure a coil to enable with a pattern of 2ms on / 9ms off, you'd use the following:

```
coils:
  some_coil:
    number: A0-B1-6
    pwm_on_ms: 2
    pwm_off_ms: 9
```

(E) Configuring lights on a lamp matrix

If you're using your P-ROC to control a lamp matrix via a PD-8x8, then you need to configure the number for each lamp in your `matrix_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.

```
matrix_lights:
  some_light:
    number: C-A2-B0-0:R-A2-B1-0
```

Notice there are two parts to the number, separated by a colon. The first part starts with `C` which means “column.” The `A2` means the column output is on the PD-8×8 board at address 2. `B0` means that it’s connected to “bank 0”, and the final `-0` means it’s connected to “output 0”. The same is true for the row side after the colon.

So putting it all together, `C-A2-B0-0:R-A2-B1-0` means that the light *some_light* is connected the column which is on output 0 of bank 1 of the PD-8×8 at address 2, and it’s connected to the row which leads back to output 0 of bank 1 of the PD-8×8 running at address 2. (Phew!) Luckily this is only something you have to work out once. :)

(F) Configuring LEDs connected to a PD-LED board

LED numbering with PD-LED boards

The [PD-LED board](#) (which is used with a P-ROC or P3-ROC pinball controller) directly drives single color LED outputs. When you use it with RGB LEDs, you combine three output per LED. The PD-LED supports both common cathode and common anode 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. The first part is the address of the PD-LED board on the serial chain (as configured via the DIP switches on the PD-LED, the second is the output number of the red segment, the third is the green segment, and the fourth is the blue segment. You separate these all with dashes, so an example PD-LED configuration might look like this:

```
l_led0:
  number: 8-0-1-2
```

The example above configures the LED that you’ll refer to as “led0” as the LED connected to PD-LED board at address 8, using outputs 0, 1, and 2 as its R, G, and B connections.

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:

```
l_shoot_again:
  number: 8-60-61-62
  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.

(G) Configuring flashers

If you have flashers which are connected up to a PD-16, then you configure their number in the same way you configure a coil number as outlined in Section C.

How To: Use a P3-ROC with MPF

This how to guide explains how to setup your MPF configuration files to interface with a Multimorphic P3-ROC pinball controller. Most of this information is available elsewhere in the documentation, but this guide pulls it all together in one place.

(A) Installing the P3-ROC drivers on your computer

In order to use a P3-ROC with MPF, you need to have the P3-ROC software libraries installed on your computer. These are called *libpinproc* and *pypinproc*. (*Libpinproc* is the actual interface library, written in C, and *pypinproc* is a wrapper for that library that makes it available to Python programs such as MPF.)

The MPF all-in-one installer for Windows or Linux will install the necessary drivers for you. (If you don't want to use the all-in-one installer, you can still follow the [platform-specific links](#) for installing MPF which have the P-ROC software libraries pre-compiled and ready to copy to your computer.)

(B) Configuring your platform settings for a P3-ROC

To use MPF with a P3-ROC, you need to configure your platform as `p3_roc`, like this:

```
hardware:
  platform: p3_roc
  driverboards: pdb
```

You also need to configure the `driverboards:` entry for P-ROC driver boards (`pdb`).

(C) Configuring switches

For switches, you can use all the settings as outlined in the [switches:](#) section of the config file reference.

Number

The P3-ROC uses switches connected to SW-16 boards. You have two options for entering switch numbers:

- You can enter the raw switch number. Board address 0 is 0-15, board address 1 is 16-31, etc.
- You can enter them as a combination of board/bank/switch, like 1/0/2.

```
switches:
  some_switch:
    number: 0/1/4
```

Debounce

The P3-ROC doesn't support variable debounce times—it's either on or off. So with P3-ROC controllers, a debounce value of `True` (or any non-zero integer) means that debounce is enabled for that switch, and a value of `False` (or 0) means that debounce is disabled.

(D) Configuring coils

There are a few things to know about controlling drivers and coils with the P3-ROC.

Number

For PD-16-based coils, the coil numbering format is *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 (0-7)
- y : Bank address (0 for A, 1 for B)
- z : Output number (0-7)

Hold Power

If you want to hold a driver on at less than full power, the P3-ROC does this by using "on time" and "off time" parameters, in milliseconds, that it switches back-and-forth.

If you set a `hold_power:` entry (which is a power value from 0 to 8), then the following actual settings will be applied with a P3-ROC:

- 0: solid off
- 1: 1ms on / 7ms off

- 2: 1ms on / 3ms off
- 3: 3ms on / 5ms off
- 4: 1ms on / 1ms off
- 5: 5ms on / 3ms off
- 6: 3ms on / 1ms off
- 7: 7ms on / 1ms off
- 8: solid on 100%

If you want more precise control over the timing of how your P3-ROC drivers will be enabled, you can use the optional settings *pwm_on_ms* and *pwm_off_ms*.

For example, to configure a coil to enable with a pattern of 2ms on / 9ms off, you'd use the following:

```
coils:
  some_coil:
    number: A0-B1-6
    pwm_on_ms: 2
    pwm_off_ms: 9
```

(E) Configuring lights on a lamp matrix

If you're using your P3-ROC to control a lamp matrix via a PD-8x8, then you need to configure the number for each lamp in your *matrix_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.

```
matrix_lights:
  some_light:
    number: C-A2-B0-0:R-A2-B1-0
```

Notice there are two parts to the number, separated by a colon. The first part starts with *C* which means "column." The *A2* means the column output is on the PD-8x8 board at address 2. *B0* means that it's connected to "bank 0", and the final *-0* means it's connected to "output 0". The same is true for the row side after the colon.

So putting it all together, *C-A2-B0-0:R-A2-B1-0* means that the light *some_light* is connected the column which is on output 0 of bank 1 of the PD-8x8 at address 2, and it's connected to the row which leads back to output 0 of bank 1 of the PD-8x8 running at address 2. (Phew!) Luckily this is only something you have to work out once. :)

(F) Configuring LEDs connected to a PD-LED board

LED numbering with PD-LED boards

The [PD-LED board](#) directly drives single color LED outputs. When you use it with RGB LEDs, you combine three output per LED. The PD-LED supports both common cathode and common anode 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. The first part is the address of the PD-LED board on the serial chain (as configured via the DIP switches on the PD-LED, the second is the output number of the red segment, the third is the green segment, and the fourth is the blue segment. You separate these all with dashes, so an example PD-LED configuration might look like this:

```
l_led0:
  number: 8-0-1-2
```

The example above configures the LED that you'll refer to as "led0" as the LED connected to PD-LED board at address 8, using outputs 0, 1, and 2 as its R, G, and B connections.

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:

```
l_shoot_again:
  number: 8-60-61-62
  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.

(G) Configuring flashers

If you have flashers which are connected up to a PD-16, then you configure their number in the same way you configure a coil number as outlined in Section C.

(H) Using the accelerometer

The P3-ROC has an accelerometer built-in which can be used to detect vibrations (for tilt) as well as to detect whether the pinball machine is level. At this point we don't have support in MPF for the accelerometer, though we have started playing with it and it's something that we'll add at some point.

How To: Use a FAST Pinball controller with MPF

This how to guide explains how to setup your MPF configuration files to interface with a FAST Pinball controller. It applies to all three of their models—the Core, Nano, and WPC controllers.

(A) Configuring the Hardware platform for FAST

To use MPF with a FAST, you need to configure your platform as *fast*, like this:

```
hardware:
  platform: fast
  driverboards: fast
```

You also need to configure the `driverboards:` entry for what kind of driver boards you're controlling: *wpc* for Williams WPC, or *fast* for FAST I/O boards like the 3208, 1616, or 0804.

(B) Configuring the FAST-specific hardware settings

When you use FAST hardware with MPF, you also need to add a [fast: section](#) to your machine-wide config which contains some FAST-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 FAST hardware 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 plug-in and turn 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 and what else you've plugged in in the past. The mapping order of the four ports is the same across all FAST controllers:

- First (lowest numbered) port: DMD processor

- Second: NET processor (the main processor)
- Third: RGB LED processor
- Fourth: Unused

Note that the FAST Nano controller does not have a DMD processor, so on that device, both the first and fourth ports are unused.

There are transmit/receive headers on the FAST controller boards for the unused ports, so you can develop your own hardware or plug in whatever you want to the boards themselves and then just use the unused FAST ports to connect to the computer. (Saves having to buy another \$20 FTDI breakout board!)

Adding the ports to your config file

If you're using a FAST Core or FAST WPC controller, you need to add the first three to your MPF config. So if you plug in the Core or WPC controller and see ports *com3*, *com4*, *com5*, and *com6* appear, you'd set your config like this:

```
fast:
  ports: com3, com4, com5
```

Note that if 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.

If you're using a FAST Nano controller, you need to add the the middle two ports that show up. So if you plug in the Nano and see ports *com3*, *com4*, *com5*, and *com6* appear, you'd set your config like this:

```
fast:
  ports: com4, com5
```

Full details of the port options as well as the other options available here are in the [fast: 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.

(C) Configuring 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 FAST-specific:

number:

FAST switches are numbered sequentially, starting with zero, from the IO board closest to the controller and then out from there. So if you have a 1616 (16 switches) and then a 3208 (32 switches), the switches from the 1616 will be numbered 0-15 and then the switches from the 3208 will be numbered 16-47.

You can enter your FAST switch numbers as either integers or hex, whichever is easiest for you. Some people like hex because that's what the serial terminal shows when you hit switches manually. Other people like integers because they're normal humans. (You can specify whether your numbers will be in hex or integer format in the [fast: config_number_format: section](#) of your config file.)

connection:

FAST switches have an optional setting called connection: which is used to specify whether the switches are connected locally to the FAST controller or whether they're on the I/O board network. Currently the only FAST controller that has local switches is the FAST WPC controller for the switches in the WPC machine that connect directly to it.

Because of that, if your driverboards: setting is wpc, then MPF will assume your switches are "local", and if your driverboards: setting is fast, then MPF will assume your switches are network. However if you make a cool mod that requires adding a FAST I/O board to a WPC machine, then you can add connection: network to those switches to differentiate them from the local WPC switches.

Debounce options

FAST controllers have advanced capabilities when it comes to debouncing switches. You can specify a debounce time (in milliseconds) from 0 to 255 ms. This can be different for each switch, and it can be different for debounce open versus debounce closed.

By default, the debounce settings will be whatever you have configured in the fast: section of your machine configuration file, but you can add debounce_open: and debounce_close: values to any of your switches to fine-tune them. For example:

```
switches:
  some_switch:
    number: 0a
```

```
debounce_open: 12
debounce_close: 6
```

(D) Configuring Coils

Coil and driver numbering with FAST I/O boards is similar to switch numbering. The drivers are number in order, starting with zero, and starting with the I/O board closest to the controller. Then they count up from there.

Also like switches, you can specify whether the number format is in hex or int.

And, also again like switches, FAST controllers differentiate between local and network drivers. Local drivers are used for WPC drivers, and network drivers are anything connected to FAST I/O boards. Again these defaults are set automatically based on your driverboards: setting.

Hold power

When you "hold" a driver on in MPF, you can set the power level so you don't burn up your coils. (In WPC machines, coils that were held one ran with lower voltage, so they could be held on at 100% no problem. But if you're building a new machine, it's probably easier to hold a coil on at less than 100% power rather than getting another power supply for lower hold voltage.)

FAST controllers hold coils on with a pulse-width modulation (pwm) mask which basically lets you configure eight ones and zeros that correspond to each millisecond of a pattern that's repeated every 8 milliseconds.

In other words, if the pwm pattern is 11001100, then the coil will be on for 2ms, then off for 2ms, then on for 2, etc...

There are two ways to configure this in MPF with FAST hardware.

The first is to use the coil's "hold_power" setting which is a numeric value between 0 and 8 which corresponds to a power level. 0 is 0% power (e.g. "off"), 8 is 100% power (e.g. "solid on), 4 is 50% power, 3 is 37.5% power, etc.

To configure a coil with a hold power value of less than 8 (full power), you simply set it up like this:

```
coils:
  some_coil:
    number: 1b
    hold_power: 3
```

Pulse power

The FAST hardware also has the ability to specify the "pulse power". Pulse power is like hold power, though it's only used during the coil's initial pulse time. For example, consider the following configuration:

```
coils:
  some_coil:
    number: 1b
    pulse_ms: 30
    pulse_power: 4
```

When MPF sends this coil a pulse command, the coil will be fired for 30ms at 50% power.

You can even combine pulse_power and hold_power, like this:

```
coils:
  some_coil:
    number: 1b
    pulse_ms: 30
    pulse_power: 4
    hold_power: 2
```

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.

Fine-tuning power values

Since FAST uses an 8-bit pwm mask to control the pulse and hold power of drivers, when you enter a pulse_power or hold_power setting, MPF automatically converts the numeric value into an 8-bit pwm mask, like this:

- 0: 00000000
- 1: 00000001
- 2: 10001000
- 3: 10010010
- 4: 10101010
- 5: 10111010
- 6: 11101110
- 7: 11111110
- 8: 11111111

That should work fine for most cases, but we could envision scenarios where you might want more fine-grained control. To enable this, you can use pulse_pwm_mask and

hold_pwm_mask settings where you actually enter an 8-digit strings of ones and zeros for the mask. For example:

```
coils:
  some_coil:
    number: 1b
    pulse_ms: 30
    hold_pwm_mask: 11001100
```

For really fine-grained scenarios, FAST also has the ability to use 32-bit pwm masks, though we haven't added that functionality to MPF yet. If you need it, contact us and we'll get it added.

(E) Configuring LEDs

Each FAST Pinball Controllers 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.)

There are two ways you can configure RGB LEDs for your FAST controller: by channel & output number, or directly with the FAST hardware number. It's more straightforward to configure them by channel and output, like this:

```
l_led0:
  number: 0-0
l_right_ramp:
  number: 2-28
```

In the example above, RGB LED *l_led0* is LED #0 on channel 0, and *l_right_ramp* is LED #28 on channel 2. Note both the channel and LED numbers start with 0, so your channel options for a FAST controller are 0-3, and your LED number options are 0-63. Also note that when you enter your FAST LED numbers with a dash like this, the values are integers, even if the rest of your FAST settings are in hex.

(F) Configuring matrix lamps

The FAST WPC controller controls the lamp matrix of WPC machines. This means you have to configure those lights in the `matrix_lights:` section of your machine configuration file. Like the other WPC-related settings, you can enter the numbers right out of your operators manual, so there's nothing FAST-specific you have to do.

(G) Configuring a DMD

The FAST WPC and Core controllers can control traditional mono-color pinball DMDs via the 14-pin DMD connector cable that's been in most pinball machines for the past 25 years. To do this, just make sure that you have your `dmd: section` set to `physical: yes` and everything else should just automatically work.

If you want to control a color DMD, an LCD-based DMD, or a SmartMatrix RGB LED-based DMD, then you can do that with any FAST Pinball controller.

How To: Use a System 11 machine with MPF

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 switched, some are controlled, some are flippers, and some are special.

Here's the solenoid table from Pin*Bot:

PIN-BOT Solenoid Table

Sol. No.	Function	Solenoid Type	Wire Color ¹	Connections		Driver Trans.	Solenoid Part No.
				CPU Bd.	Playfield/ Cabinet		
01A ³	Outhole	Switched	{Vio-Brn}	1P11-1	8P3-1 (to B1 on Diode Sw. Bd.)	Q33	AE-23-800-01
01C ³	Knocker	Switched	{Blk-Brn}	(Gry-Brn)	Diode Sw. Bd.)	Q33	AE-23-800-02
02A ³	Ball Trough Feeder	Switched	{Vio-Red}	1P11-3	8P3-2 (to B2 on Diode Sw. Bd.)	Q25	AE-23-800-03
02C ³	Upper P'fld & "Top" Flashers (2)	Switched	{Blk-Red}	(Gry-Red)	Diode Sw. Bd.)	Q25	#89 flashlamps
03A ³	Single Eject Hole	Switched	{Vio-Orn}	1P11-4	8P3-3 (to B3 on Diode Sw. Bd.)	Q32	AE-23-800-03
03C ³	Left Insert Bd. Flasher	Switched	{Blk-Orn}	(Gry-Orn)	Diode Sw. Bd.)	Q32	#89 flashlamps
04A ³	Drop Target (3-Bank)	Switched	{Vio-Yel}	1P11-5	8P3-4 (to B4 on Diode Sw. Bd.)	Q24	AE-23-800-04
04C ³	Right Insert Bd. Flasher	Switched	{Blk-Yel}	(Gry-Yel)	Diode Sw. Bd.)	Q24	#89 flashlamps
05A ³	Ramp Raise	Switched	{Vio-Grn}	1P11-6	8P3-5 (to B5 on Diode Sw. Bd.)	Q31	AE-24-900-02
05C ³	Lower P'fld & "Top" Flashers (1)	Switched	{Blk-Grn}	(Gry-Grn)	Diode Sw. Bd.)	Q31	#89 flashlamps
06A ³	Ramp Lower (Outer)	Switched	{Vio-Blu}	1P11-7	8P3-6 (to B6 on Diode Sw. Bd.)	Q23	SM-26-600-DC
06C ³	Energy Flashers	Switched	{Blk-Blu}	(Gry-Blu)	Diode Sw. Bd.)	Q23	#89 flashlamps
07A ³	Left Eject Hole (Visor)	Switched	{Vio-Vio}	1P11-8	8P3-7 (to B7 on Diode Sw. Bd.)	Q30	AE-23-800-03
07C ³	Left Playfield Flasher	Switched	{Blk-Vio}	(Gry-Vio)	Diode Sw. Bd.)	Q30	#89 flashlamps
08A ³	Right Eject Hole (Visor)	Switched	{Vio-Gry}	1P11-9	8P3-8 (to B8 on Diode Sw. Bd.)	Q22	AE-23-800-03
08C ³	Sun Flasher	Switched	{Blk-Gry}	(Gry-Blk)	Diode Sw. Bd.)	Q22	#89 flashlamps
09	Robot Face - Insert Bd.	Controlled	Brn-Blk	1P12-1	8P3-9	Q17	#1251 flashlamps
10	Right Visor - Gen. Illumin.	Controlled	Brn-Red	1P12-2	8P3-10	Q9	#1251 flashlamps
11	General Illumin. - Insert Bd.	Controlled	Brn-Orn	1P12-4	8P3-12	Q16	5580-09555-01 ⁴
12	General Illumin. - Playfield	Controlled	Brn-Yel	1P12-5	3P7-1	Q8	5580-09555-01 ⁴
13	Visor Motor	Controlled	Brn-Grn	1P12-6	8P3-13	Q15	5580-09555-01 ⁴
14	Solenoid Select Relay	Controlled	Brn-Blu	1P12-7	8P3-14	Q7	5580-09555-01 ⁴
15	"Top" Flashers (3)	Controlled	Brn-Vio	1P12-8	8P3-15	Q14	#89 flashlamps
16	"Top" Flashers (4, center)	Controlled	Brn-Gry	1P12-9	8P3-16	Q6	#89 flashlamps
17	Lower Jet Bumper	Special #1	Blu-Brn	1P19-7	8P3-17	Q75	AE-23-800-03
18	Left Visor Gen. Illumin.	Special #2	Blu-Red	1P19-4	8P3-18	Q71	#1251 flashlamps
19	Left Jet Bumper	Special #3	Blu-Orn	1P19-3	8P3-19	Q73	AE-23-800-03
20	Left Kicker	Special #4	Blu-Yel	1P19-6	8P3-20	Q69	AE-23-800-03
21	Right Kicker	Special #5	Blu-Grn	1P19-8	8P3-21	Q77	AE-23-800-03
22	Upper Jet Bumper	Special #6	Blu-Blk	1P19-9	8P3-22	Q79	AE-23-800-03
-	Right Flipper	-	Orn-Vio {Blu-Vio}	1P19-1	7P1-20 {7J1-21,8P3-34} ²	-	FL23/600-30/2600-50VDC
-	Left Flipper	-	Orn-Gry {Blu-Gry}	1P19-2	7P1-23 {7J1-24,8P3-32} ²	-	FL23/600-30/2600-50VDC

Notes: 1. Wire colors, except flipper Orn-Vio and Orn-Gry, are ground connections (to coil terminal with unbanded end of diode). Flipper Orn-Vio and Orn-Gry wires connect from CPU Board to flipper switch. 2. Flipper connections shown in braces are from flipper switch to flipper coil. 3. "A" coils are pulsed, when Sol. 14 is de-energized; "C" coils are pulsed, with Sol. 14 energized. Wire colors in brackets are those from respective A and C terminals corresponding to the B terminal connection listed for the Diode Switching Board, which controls the device pulsing by Sol. 14. 4. Relay (p/n 5580-09555-01) is mounted on Relay Snubber Ckt. Bd. p/n C-11232-1.

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?

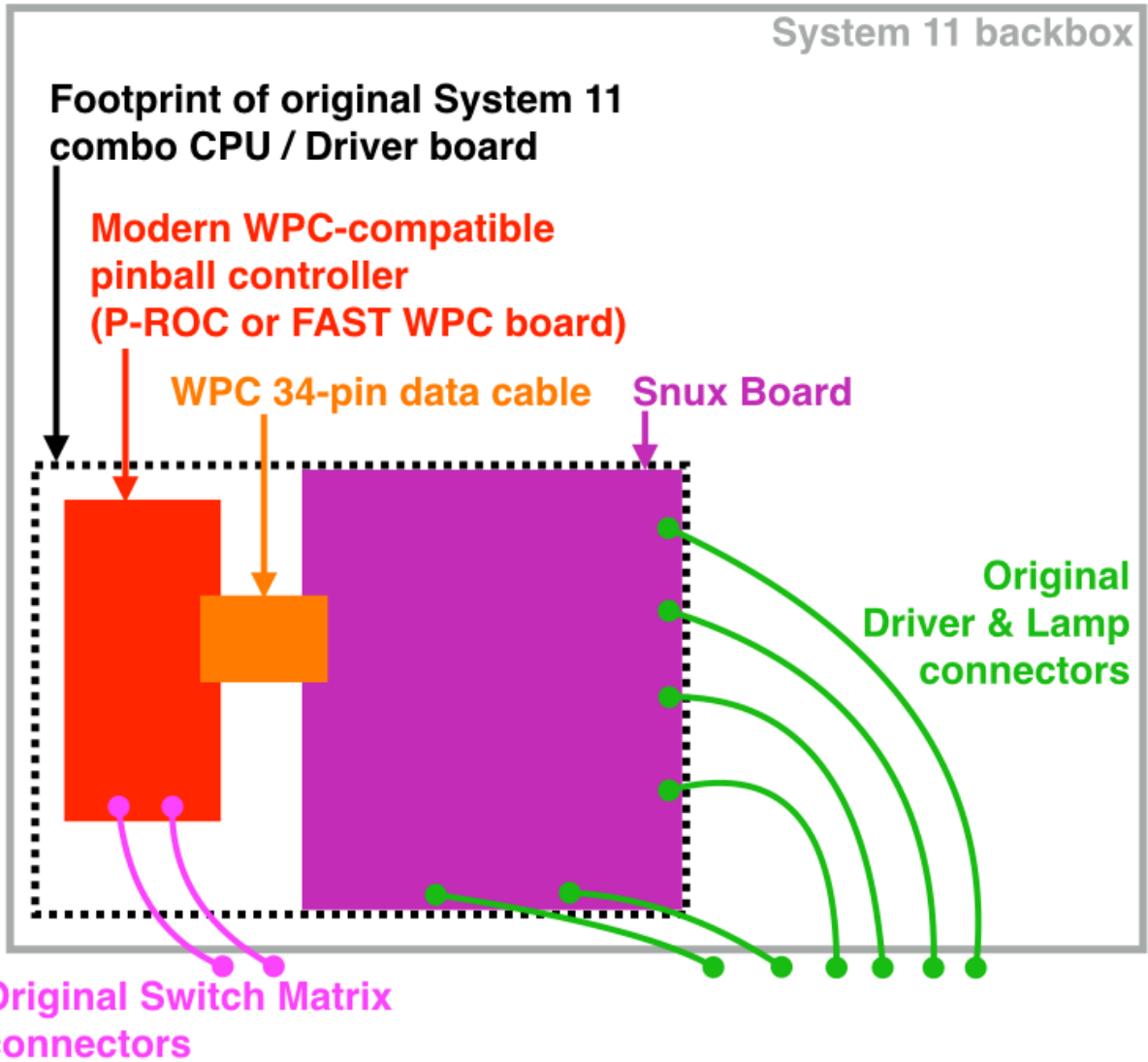
The usual way you control an existing machine is to remove the original CPU board and to replace it with either a P-ROC or FAST WPC controller. The new pinball controller plugs into the backbox and uses the existing driver board. The problem with System 11 is that unlike more modern machines, the System 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 or FAST WPC controller into a System 11 machine, you don't have a driver board. :(

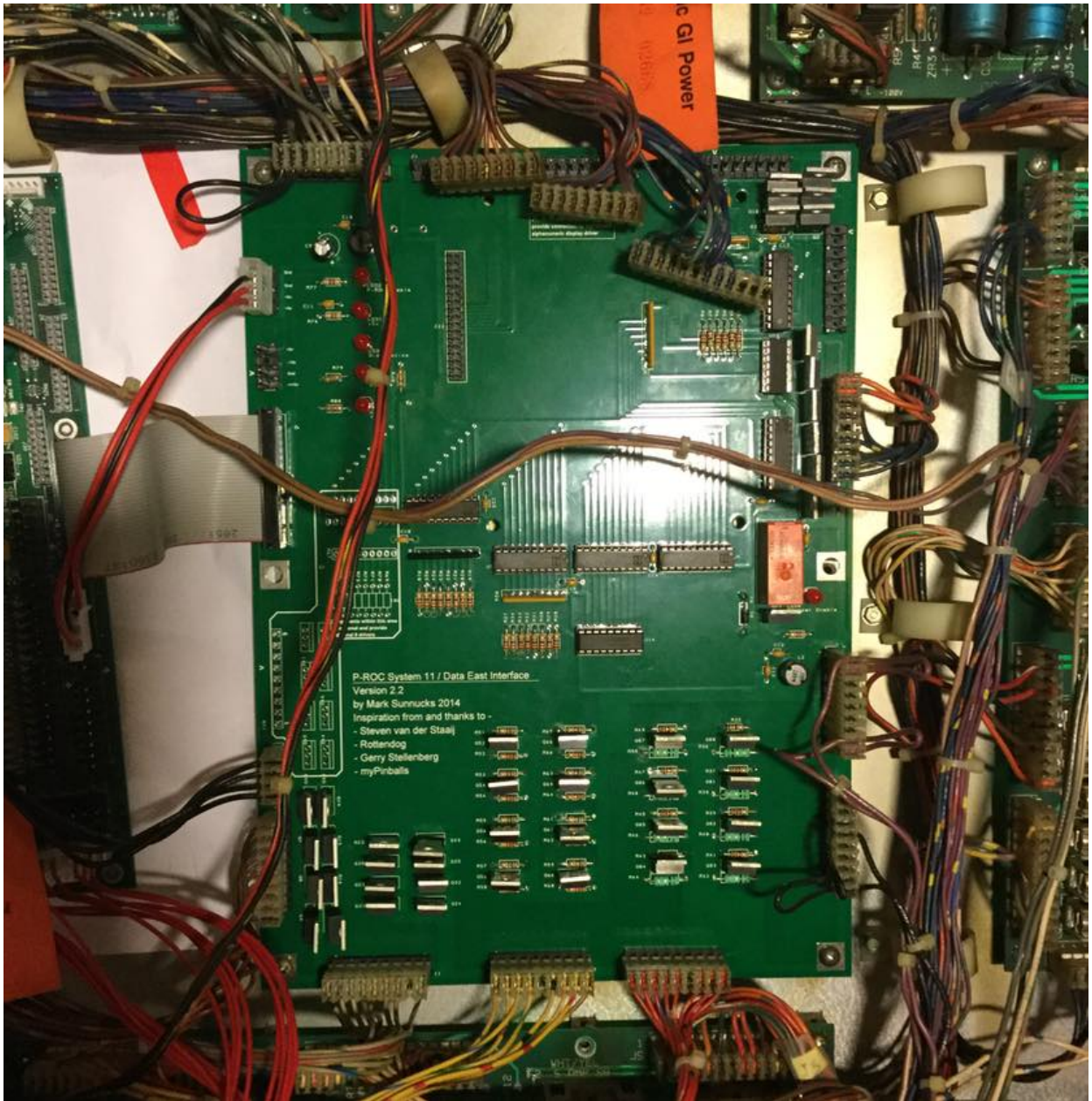
This is where the Snux board comes in. The Snux board (which is our name for it) is a System 11 driver board created by Mark Sunnucks. (His online handle is Snux which is why we call it the Snux board.) Mark developed this board a few years ago because he wanted to control an F-14 machine with a P-ROC.

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 or FAST WPC controller in it's place. The Snux board connects to the P-ROC or the FAST WPC 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.

Here's a diagram of how it's hooked up:



Here's a photo of it installed in a *Jokerz!* machine. You can just see the edge of the P-ROC on the left of the photo.



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. You can buy it in kit form (where you get the board and a bag of parts and assemble it yourself), or you can buy them complete. Mark lives in the UK, so the exact price you pay depends on the exchange rate, shipping to your country, and whether you buy the kit or the complete board, but in USD it's somewhere in the \$170-\$250 range. (Then you also have to buy a P-ROC or FAST WPC controller to drive it.)

You can contact Mark via PM. Here are the links to his profiles pages on MissionPinball.com, PinballControllers.com, and Pinside, so contact him via one of these, PayPal him some money, and enjoy your System 11 machine!

More information on his board is available on the PinballControllers.com forum. [Here's the thread](#) with the latest on his board, and [here's the original thread](#) from where Mark shared the development process of this board as he was designing and testing it.

(C) Understand how MPF works with the Snux board

Once you have your P-ROC or FAST WPC 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 or FAST WPC controller, the main platform interface MPF uses is either the P-ROC or the FAST platform. Then the Snux platform overlay layers on top of it to handle the special cases that arise when using the P-ROC or FAST controller 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 *fast* or *p_roc*, but then for *driverboards* you configure it as *snux*, like this:

```
hardware:
  platform: p_roc
  driverboards: snux
```

or

```
hardware:
  platform: fast
  driverboards: snux
```

Adding the *driverboards: snux* option automatically activates the Snux platform overlay.

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:

```
snux:
  flipper_enable_driver_number: c23
  diag_led_driver_number: c24
```

The Snux board maps driver 23 to the flipper enable relay, and it maps 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 specify 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. (We don't know of any yet though. FAST Pinball has talked about creating a System 11 interface board, but no details or timeframes are available yet.)

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. Configuring driver numbers

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
```

```

flashers:
  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!

5. 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:

```

matrix_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.

6. 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*:

```

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
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.

7. Create driver-enabled devices

One of the types of devices in MPF is called a [driver-enabled device](#). This is a device which, rather than being directly controlled, is simply enabled or disabled when a particular driver is enabled or disabled. In System 11 machines, this is the flippers and usually also the pop bumpers and slingshots.

You configure these devices in the [driver_enabled:section](#) of your machine-wide config. Here's this section from *Pin*Bot*:

```
driver_enabled:
  playfield_devices:
    number: c23
```

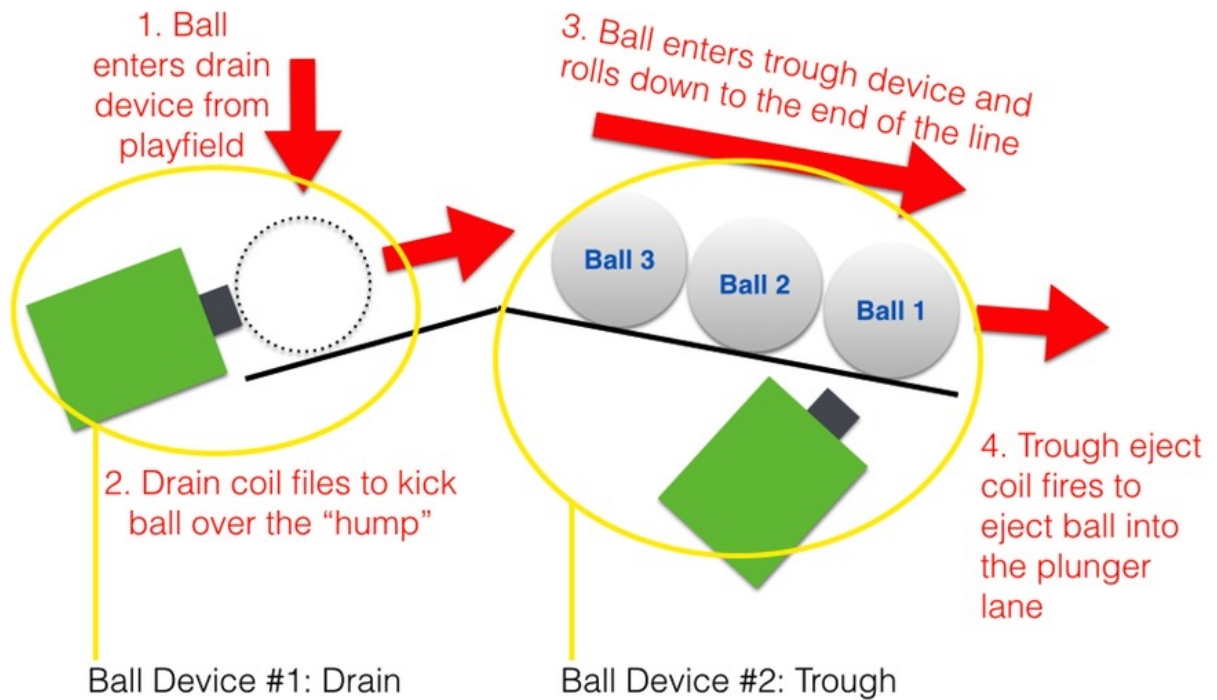
Note that we named this driver-enabled device *playfield devices*. That's because in *Pin*Bot*, all of these devices are controlled via the same driver output, so there's no sense setting them all up as separate devices. (This also means that the *Pin*Bot* MPF hardware config won't have *flippers:* and *autofire_coils:* sections.)

By default, driver-enabled devices are enabled when the ball starts and disabled when the ball ends.

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.

This diagram shows that in action:



We have a separate How To guide which details [how to setup a System 11 1980s-style trough](#) (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
  plunger_lane:
    ball_switches: plunger_lane
    mechanical_eject: true
    tags: home, ball_add_live
    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 its exact role. (And hey! Now you know why these are all separate tags in MPF instead of a single tag called "trough".)

(E) Final Steps

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.

How To: Configure RGB LED SmartMatrix displays

This How To guide explains how to connect an RGB LED-based matrix (sometimes called a "SmartMatrix") to a pinball machine running MPF. This has been called a "real" Color DMD 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 actually dark gray.

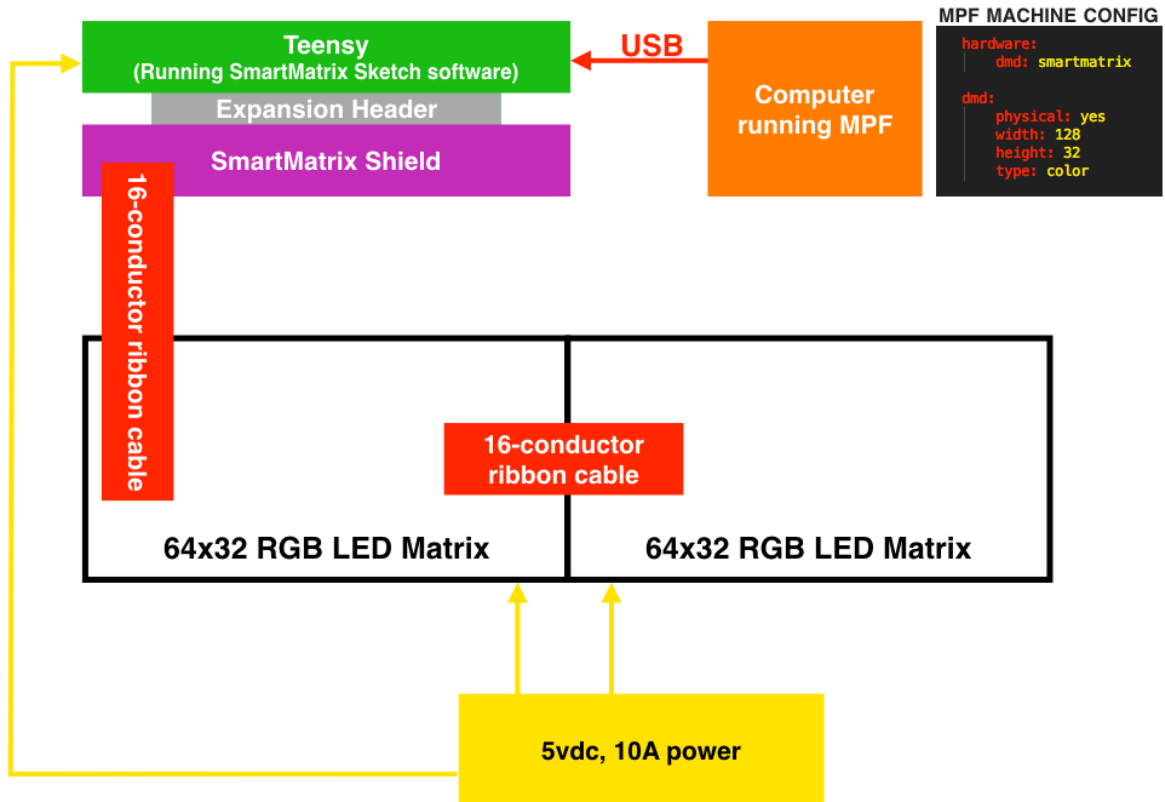
This video explains it all:

<https://www.youtube.com/embed/zbZQCBYeXOU>

Note that everything described in this How To guide is assuming that you're controlling the content of the DMD via a USB connection from a host computer running MPF. If you want to build an RGB LED display which you use in an existing pinball machine connected to the existing 14-pin DMD cable, a guy named Eli Curtz is creating an alternative to the SmartMatrix shield that can be controlled by both the computer via USB and by an existing 14-pin DMD signal from an existing machine. See [this Pinside thread](#) for details.

Here's a hookup diagram which shows how all the components you need fit together. (Click it to expand it to full size.)

RGB LED “Real” Color DMD SmartMatrix Diagram



(A) Shout-out to Eli Curtz!

First, this project wouldn't exist without Eli Curtz. [Eli first posted about these panels](#) in the P-ROC forum a year ago. 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!

(B) 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.

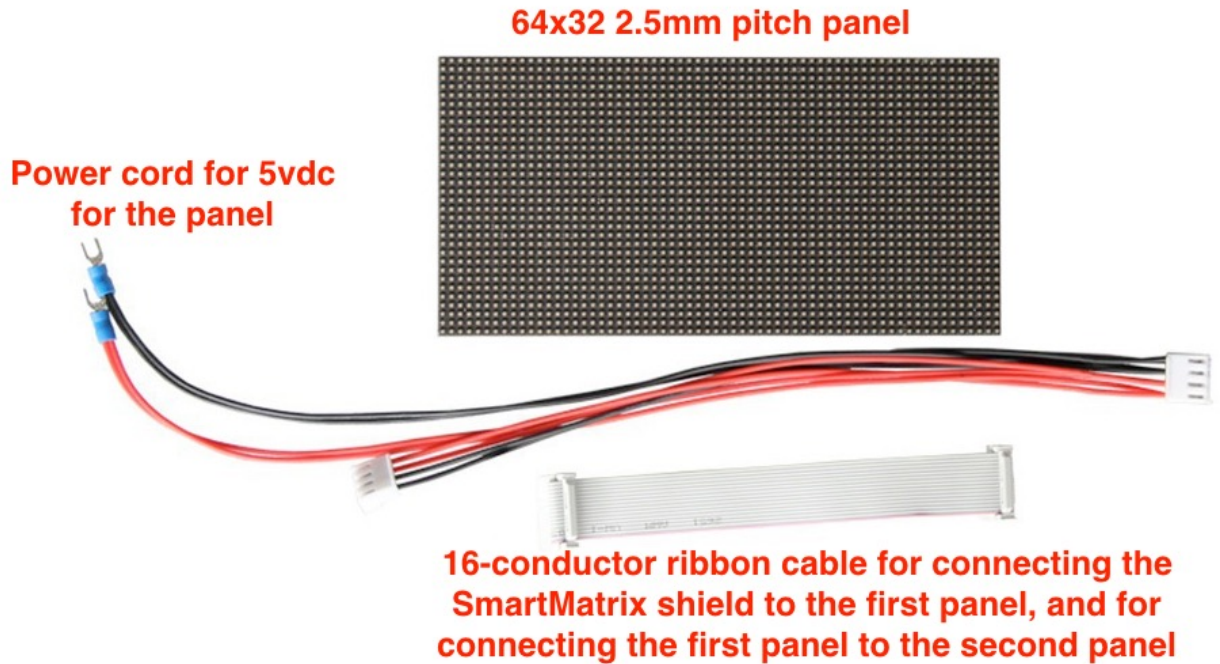
1. The Panels

The main part of this project is the display panels. The panels you'll find are sold for commercial purposes for "video walls" and "Jumbo-trons" and stuff like that. There are many sizes of panels available with different size spacing between the individual pixels. The ones that are most like an existing pinball DMD have 2.5mm spacing (also referred to as "P2.5"), which results in a 128x32 display of 320x76.8mm. This overall display will be within a 1/4" or so of the original DMD size and will fit nicely into an existing DMD/speaker panel from an existing machine.

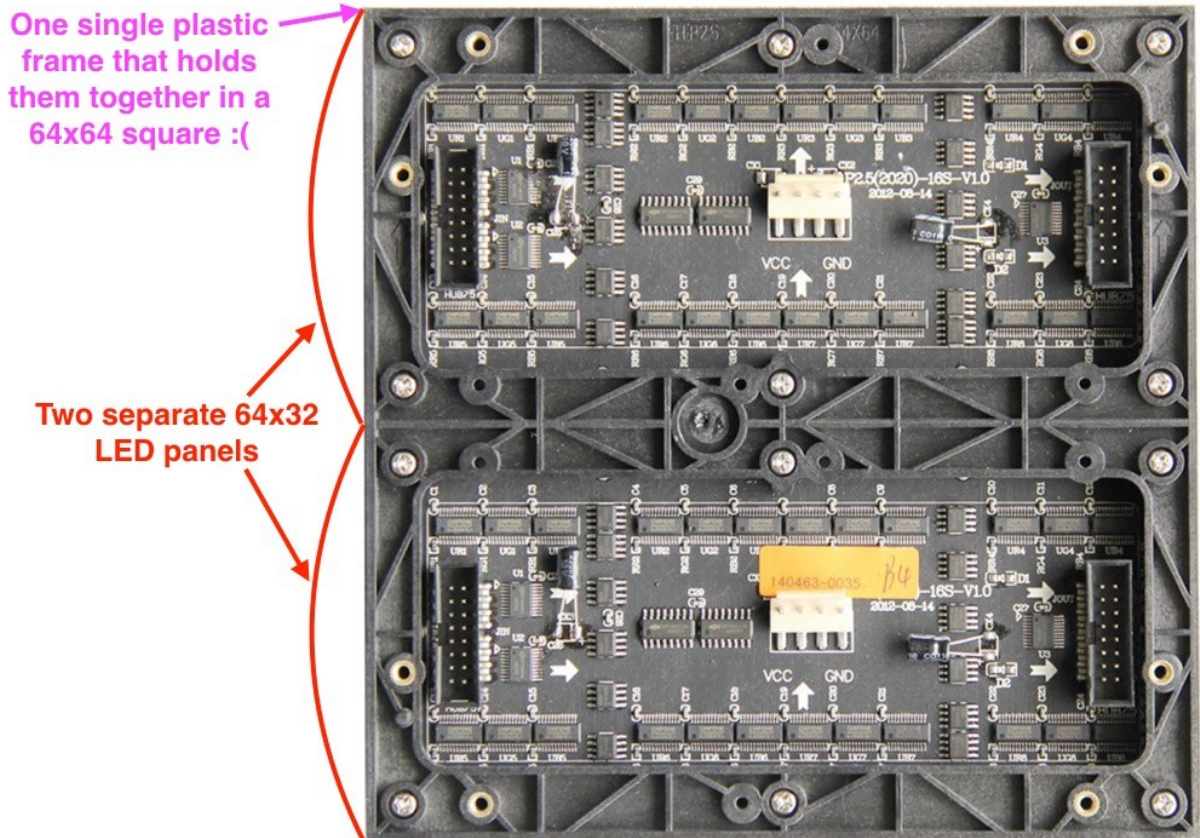
Currently the only place to buy these panels is from AliExpress. They don't sell 128x32 panels, but you can buy two 64x32 ones and connect them side-by-side. Since these things are designed for huge video walls, they are seamless. Buying stuff from AliExpress is somewhat of an adventure, and it's not always possible to tell exactly what you're going to get.

The exact panels we ordered were \$25 each for 64x32, though it appears that supplier doesn't offer them anymore. Your best bet is to search for [\[RGB LED 64x32 p2.5\]](#) and hope for the best. Prices tend to range from \$50 to \$70 per panel (and remember you need two), depending on luck as far as we can tell. (We've seen some panels as cheap as \$25, but after ordering many different models, it's clear to us that the \$25 ones were ones that had manufacturing errors that have been repaired. This isn't necessarily a bad thing, but it does seem that availability at the sub-\$50 price point is hit-or-miss.) The key is to make sure you get "P2.5" (the 2.5mm pitch), because if you accidentally end up with P4 (4mm pitch) then you're going to end up with a 2-foot by 6-inch DMD. :)

Ideally you'd end up with something like this:



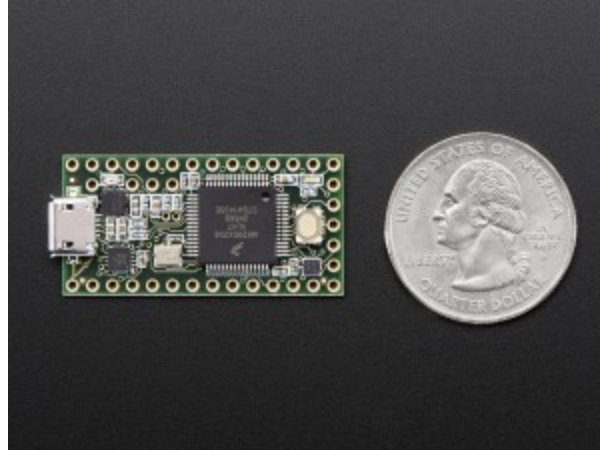
Unfortunately when you actually place the order for two of them, you might end up with them mounted in a 64x64 configuration onto a single plastic frame. In this case you will probably still have two 64x32 panels, but you'll have to remove them from the plastic frame and cut the frame in half, or you'll have to make your own frame.



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 we assume is a standard in the gigantic outdoor display industry since it seems that all the panels have the same standards here.)

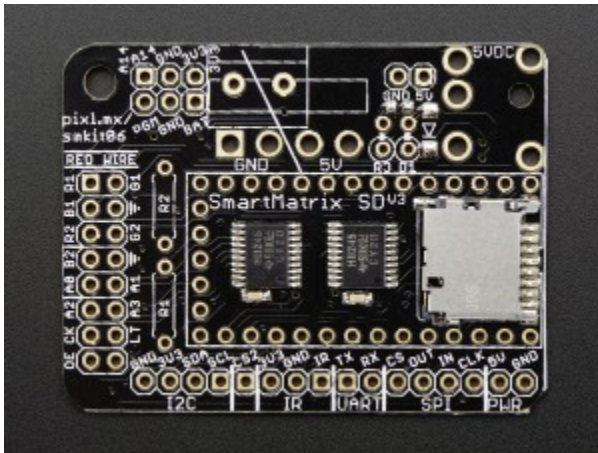
We use the Teensy 3.2 which is available from multiple sources for about \$20. ([Here's the link to the website](#) of the guy who actually built it.) The Teensy is similar to an Arduino (in fact it runs the same software sketches as Arduinos) though it has a slightly different processor architecture which is needed for the rapid bit-shifting of data needed to control these panels.



The software to run the Teensy is open source (more on that in Step C) 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 ([\\$15 from the guy who made the Teensy](#), though out of stock at the moment, so you might have to spend [\\$25 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.



As we wrote in the introduction, this Teensy+SmartMatrix Shield combo can only be used with source data coming from a computer via USB. If you want a way to "colorize" an existing 14-pin DMD signal from an existing pinball machine, you'll need to use a different board than the Teensy. Eli Curtz is developing a dual-purpose board which can get its signal from a computer or from a 14-pin DMD cable, so as long as that board costs \$25 or less, than you should probably buy it instead of the SmartMatrix Shield since that way the DMD you build here will be usable in regular pinball machines too if you decide not to use it in your MPF-powered machine.

Here's a look at Eli's board so far. Check back in [that Pinside thread](#) for details about its availability.



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!

[Here's a 5v 8A power supply](#) for \$15 from Amazon. Unfortunately 8A is not enough if you want to run these things at full brightness. I ended up buying [this 5vdc 60A power supply](#)

from Amazon for \$60 instead. (An ATX computer power supply will probably have a decent amount of amps also, just check the specs.)

That said, you might be fine with a 8A supply. The thing is, at 100% brightness, these things are bright. I mean like "burn your retinas" bright. You can set the overall brightness level (more on that in the next step), and when you do that, the displays require less power. Even at 50% brightness, 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, and that \$15 8A power supply should be fine if you don't have enough 5v power elsewhere.

You need to connect that power supply up to both 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!)

(C) Load the SmartMatrix code onto the Teensy

Once your hardware's built, you need to load the code onto the Teensy which receives your 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](#) if you want to look at it, and [here's is the original sample code](#) we based our code on.)

Note that the width and height of your display is set in lines 11 & 12. You can change that if you want to a different size display. (Snux 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 here too. By default it's 100%, but you can change that in line 22 to whatever you want.

Here's a quick overview of how to install this code onto the Teensy. Full instructions are [here](#).

- Install the Arduino IDE v1.6.5
- 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
- Push the button on the Teensy to put it into programming mode
- Compile & load the code onto the Teensy from the Arduino IDE

Note that if you're using Eli's board with the 14-pin DMD input option, then you'll install different code (from him) instead of the code from MPF.

(D) Configure MPF to use it

Finally you need to update your MPF machine-wide configuration to use the new display.

1. Configure your DMD

Follow the existing instructions for setting up a DMD ([Step 6\(C\) from the MPF tutorial.](#)) Then set *physical: yes* and *type: color*, so your final DMD config looks like this:

```
dmd:
  physical: yes
  width: 128
  height: 32
  type: color
```

2. Configure your DMD platform to use the SmartMatrix

Next you need to make a platform setting that tells MPF that it should use the *smartmatrix* platform interface for your DMD rather than using the existing platform you have set (P-ROC, FAST, etc.). To do this, go to your *hardware:* section and add *dmd: smartmatrix*. For example:

```
hardware:
  platform: fast
  driverboards: fast
  dmd: smartmatrix
```

3. Configure your SmartMatrix settings

Finally, add a *smartmatrix:* section to your machine-wide config and then configure the two options for your port and whether or not you'll run the SmartMatrix communication code in a separate thread:

```
smartmatrix:
  port: com12
  use_separate_thread: yes
```

The port is just whatever virtual serial port appears when you plug in the Teensy. The correct setting for the thread will depend on the specifics of your hardware and what size display you're running. For example, on my test system (MPF running in a Windows VM on a MacBook), it didn't matter what I set the thread to—it was the same either way. For Snux who used the 128x64 display, he had to set *use_separate_thread: no* to get good performance. So basically try it with both settings and see which one works better.

(E) Color your slides

At this point you should be able to run your game, though all the content on the DMD will be white since it doesn't include any color data.

We have an existing How To guide which explains [how to configure a color DMD](#) and walks you through setting adding colors to the dynamic elements (text, shapes, etc.) that you add to your display. Even though that guide is written for a "virtual" (on screen, LCD-based) color DMD, the techniques are the same. (The only difference is that you'll have the *physical* setting set to *yes* which means whatever shows up on the DMD will be sent to the physical DMD.)

Essentially any images or videos you play will just sort of automatically work in color. And then any other display elements you add, you'll include a *color:* setting with a 6-character RGB color value instead of the current *shade:* setting which specifies an intensity from 0-15.

Troubleshooting

We'll add troubleshooting tips here as we find them. So far the only thing that came up is one guy didn't have any luck with anything appearing on the display, but that's because he was running MPF with the `-x` option which is "no hardware." So if you use `-x`, MPF will not connect to any physical hardware and it won't work.

If you want to play with the SmartMatrix display without a FAST or P-ROC connected, then set your platform: section like this:

```
hardware:
  platform: virtual
  dmd: smartmatrix
```

Then run MPF but *do not use -x*, and it should work.

Also there's a Python script called `eli_test.py` in the tools folder which you can run to display some moving rainbow bars on the display. To use it, open it up and set your port in line 4, and then change to the `/mpf/tools` folder and run `python eli_test.py`. You should see the bars appear on the display at that point.

How To: Use an event to disable a switch

Question from the forum:

I have a coil that can be fired by either switch A or switch B. However, if the drop bank is completed, then I need to disable switch A until

switch B is triggered. Then it reverts back to using either switch again. Also, the coil needs to hold for about a second.

Right now I have a mode watching for the switches to be activated. Then it fires and holds the coil as it should. Is there a way in MPF to disable a switch or to tell the mode to stop watching that switch until another event happens? Is there a better way to do this? Thanks!

Here's how we'll handle this.

We're going to do this by writing some custom Python code. It's probably possible to do it in config files with logic blocks and stuff, but this is one of the scenarios where it's much faster and less confusing just to write it in Python.

First, we have to decide whether we'll register switch handlers to talk to the Switch Controller and watch for the switches to change, or whether we'll look for the events that are posted when a switch is activated. (Remember that all switches post an event `<switchname>_active` when they're activated.

We're going to use the event method here. Why? Two reasons. First, when you write code for modes, the Mode parent class automatically cleans up and removes any event handlers you've registered in that mode when the mode stops. So that means you don't have to worry about doing that yourself.

Second, registering for events is the more "MPF-way" to do things. Using events means this code can be easily changed to work with any event. (For example, maybe you want to look for shot completion events instead of raw switch hits. In that case it would be the same code, just looking for different events.

So let's setup the code now. First create the skeleton for your mode code if you don't have it yet. We'll call this mode "Alien":

```
from mpf.system.mode import Mode

class Alien(Mode):
```

For this demonstration, let's assume the drop target bank is called "*drop_targets_1*", and the switches are *switch_a* and *switch_b*.

The way we're going to do this is to create two modes of operation. One is when either switch can be hit to fire the coil, and the second is while the blocking is in place and it's waiting for *switch_b* to trigger. So first let's create a method that actually fires our coil. We'll use the driver method `timed_enable()` which lets you to fire a coil for a set time that's longer than 255ms (since 255ms is the maximum pulse time.

```
def fire_coil(self, **kwargs):
    self.machine.coils['some_coil'].timed_enable(1000)
```


Next let's add a method called *disable_blocking()* that allows both *switch_a* and *switch_b* to fire the coil:

```
def disable_blocking(self):
    self.add_mode_event_handler(event='switch_a_active',
                                handler=self.fire_coil)
    self.add_mode_event_handler(event='switch_b_active',
                                handler=self.fire_coil)
```

Then we'll create a method called *enable_blocking()* which removes all the events with *self.fire_coil()* as their callback and then re-sets up *switch_a* to fire the coil:

```
def enable_blocking(self):
    self.machine.events.remove_handler(self.fire_coil)
    self.add_mode_event_handler(event='switch_b_active',
                                handler=self.fire_coil)
```

Finally, add an entry to your *mode_start()* method to start with either blocking enabled or disabled, like this:

```
def mode_start(self, **kwargs):
    self.disable_blocking()
```

The complete code should look like this:

```
from mpf.system.mode import Mode

class Alien(Mode):
    def mode_start(self, **kwargs):
        self.disable_blocking()
    def fire_coil(self, **kwargs):
        self.machine.coils['some_coil'].timed_enable(1000)

    def disable_blocking(self):
        self.add_mode_event_handler(event='switch_a_active',
                                    handler=self.fire_coil)
        self.add_mode_event_handler(event='switch_b_active',
                                    handler=self.fire_coil)
    def enable_blocking(self):
        self.machine.events.remove_handler(self.fire_coil)
        self.add_mode_event_handler(event='switch_b_active',
                                    handler=self.fire_coil)
```

So now we have a coil that is fired for one second if either *switch_a* or *switch_b* is hit, and then after either switch is hit and the coil is fired, the "blocking" is enabled and only *switch_b* works.

But this isn't quite what the question asked. In the question, there's an external trigger event (the drop target bank being completed) that triggers the blocking to start. So we need to change the code a bit.

In MPF, when a drop target bank is completed, it post an event `<drop_target_bank_name>_down`. So in this case with `drop_targets_1`, we're looking for an event `drop_targets_1_down`. So change our `mode_start()` method so it looks for that event, and then it enables the blocking when it happens. (Note that we keep `self.disable_blocking()` in there because we still need to set it up to look for both `switch_a` and `switch_b` events.)

```
def mode_start(self, **kwargs):
    self.disable_blocking()
    self.add_mode_event_handler(event='drop_targets_1_down',
                                handler=self.enable_blocking)
```

The only other change we could make here is what happens when `switch_b` is hit while the blocking is enabled? Currently this code fires the coil for 1 second. But maybe you don't want `switch_b` to fire the coil when blocking is enabled, and instead you want it to just remove the blocking? Simple! Just change your `enable_blocking()` method handler for `switch_b` to be `self.disable_blocking` instead of `self.fire_coil`.

So now your mode code looks like this:

```
from mpf.system.mode import Mode

class Alien(Mode):

    def mode_start(self, **kwargs):
        self.disable_blocking()
        self.add_mode_event_handler(event='drop_targets_1_down',
                                    handler=self.enable_blocking)

    def fire_coil(self, **kwargs):
        self.machine.coils['some_coil'].timed_enable(1000)

    def disable_blocking(self, **kwargs):
        self.add_mode_event_handler(event='switch_a_active',
                                    handler=self.fire_coil)

        self.add_mode_event_handler(event='switch_b_active',
                                    handler=self.fire_coil)

    def enable_blocking(self, **kwargs):
        self.machine.events.remove_handler(self.fire_coil)

        self.add_mode_event_handler(event='switch_b_active',
                                    handler=self.disable_blocking)
```

We're almost there, except there's one thing that's still not quite right. The problem now is that when the mode starts, both `switch_a` and `switch_b` are active to fire the coil. Then when the drop target bank is completed, blocking is enabled and the switches no longer fire the coil. So far, so good. But when `switch_b` is hit at this point, it will call `self.disable_blocking()` which will register `switch_a` and `switch_b` to fire the coil (which

is good), but *switch_b* is still registered to also *disable_blocking*, which probably isn't right. (It should be that only the drop target bank being completed disables blocking.)

So we need to remove the *event_handler* for *switch_b* in our `disable_blocking()` method, meaning our new complete code looks like this:

```
from mpf.system.mode import Mode

class Alien(Mode):

    def mode_start(self, **kwargs):
        self.disable_blocking()
        self.add_mode_event_handler(event='drop_targets_1_down',
                                    handler=self.enable_blocking)

    def fire_coil(self, **kwargs):
        self.machine.coils['some_coil'].timed_enable(1000)

    def disable_blocking(self, **kwargs):
        self.add_mode_event_handler(event='switch_a_active',
                                    handler=self.fire_coil)

        self.add_mode_event_handler(event='switch_b_active',
                                    handler=self.fire_coil)
        self.machine.events.remove_handler(self.disable_blocking)

    def enable_blocking(self, **kwargs):
        self.machine.events.remove_handler(self.fire_coil)

        self.add_mode_event_handler(event='switch_b_active',
                                    handler=self.disable_blocking)
```

So there you have it!

You can use these techniques in your own game code to set up complex scenarios and logic based on events and switches. It seems like a lot at first, but once you break it down step-by-step as we've done here, you see it's not too bad.

A few notes:

- You can safely remove an event handler even if no handlers are registered. For example, this code called `disable_blocking()` when the mode starts, and that code will remove the event handlers registered for `disable_blocking()`. The first time this is called there will not be any handlers for that callback, but that's ok.
- Since the event handlers are registered via `self.add_mode_event_handler()` versus the machine-wide `self.machine.events.add_handler()`, they will automatically be removed when the mode stops, so you don't have to worry about removing them yourself.
- When you add event handlers, notice that you do not include parenthesis in the method you specify as your handler. Why? Because in Python, when you add

parenthesis, that means you're calling that method right there. In our handlers, we just want to pass the method as the handler, rather than calling it on the spot, which is why there are no parenthesis there.

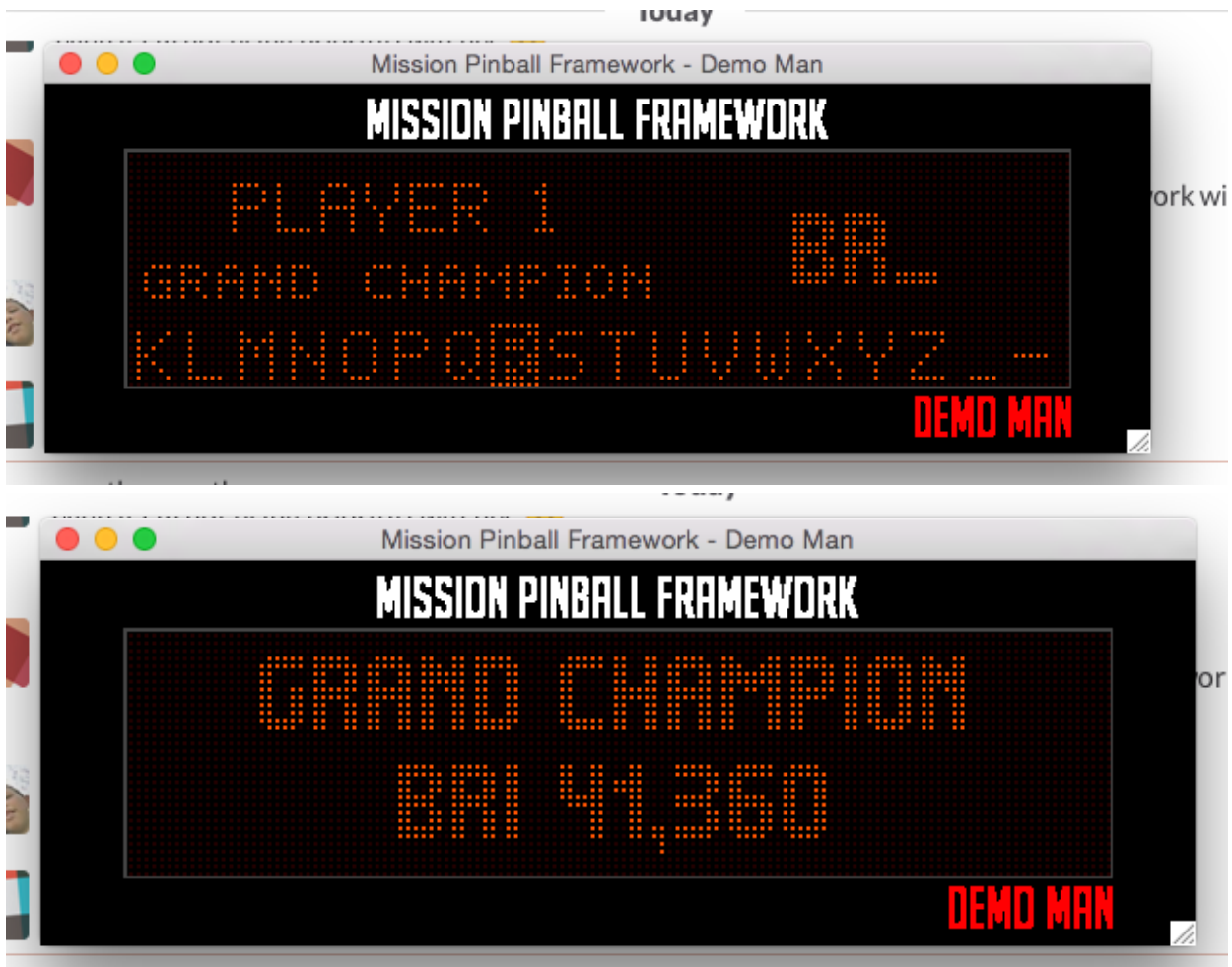
How To: High Scores

This How To guide explains how to setup a high score mode in MPF. The MPF package contains a the code for a high score mode, so all you have to do to use add some configs to your machine's *modes* folder and you're all set.

Features of the high score mode 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. (This is done with regular events you can use with the `slide_player`, and two new display elements have been created—a character picker and a text field that shows the characters entered.) You can set up slides like any slide in MPF.
- BCP events for high score awards and entry, so you can configure high score entry screens for the built-in MPF media controller or the Unity backbox controller.

Here are a few examples of the high score mode in action. These examples are with a traditional single-color DMD, but they're generated with standard events and display elements, so you can make a color DMD, a high-def LCD display, etc.



Now lets dig in to actually getting this setup:

(A) Create your `high_score` mode folder

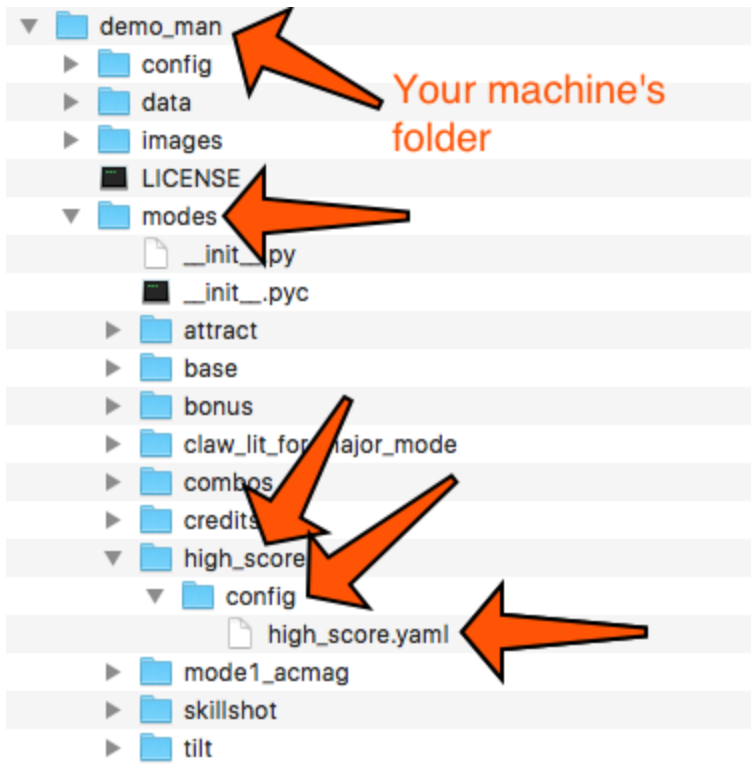
The high score mode works like any other mode in MPF. You'll create a folder called `high_score` in your machine's `modes` folder, and that folder will contain subfolders config files, images, etc.

So to begin, create a folder called `<your_machine>/modes/high_score`.

Then inside there, create another folder called `config`.

Then inside there, create a file called `high_score.yaml`. (So that file should be at `<your_machine>/modes/high_score/config/high_score.yaml`.)

Your folder structure should look something like this:



(B) Decide what things you want to track high scores from

MPF uses the concept of *player variables* ([more info here](#)) to keep track of various things on a per-player basis. The most obvious example of this is the score. Each player has a *score* variable since obviously you need to track the score separately for each player. There are also player variables for things like the player number, current ball number, how many extra balls the player has, etc.

As you build-out your game config, you can leverage player variables to track anything you want on a per-player basis. You can access and update player variables via logic blocks, the scoring settings, or directly via custom code you write.

The concept of player variables is tightly-coupled with MPF's high score mode. When you decide how high scores will work in your machine, what you're actually doing is mapping player variable names to high score categories and names.

So the first thing you need to decide is what player variables you want to track as high score achievements. For example, you probably want to track the score variable and maybe keep the top 5 highest scores in your list.

You might also track loops for a "loops champion". You could set it so that every time a loop shot was created, a player variable called loops was incremented by one, and then you could use the high score system to track whichever player had the most loops in a game.

Again, you can base your high scores on any player variable you want, and you can track as many entries for each as you want. So the first step to implementing high scores is to decide what you want to track.

(C) Add settings to your `high_score.yaml` config file

Once you decide what you want to track, the next step is to actually build-out the config file for it.

Open up the high score mode's config file that you just copied into your machine folder. It should be at `<your_machine>/modes/high_score/config/high_score.yaml`.

Since this file is totally blank, add the required `#config_version=3` to the top line.

Next, add a section called `high_score:`, 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 `high_score.yaml` file should now look like this:

```
#config_version=3

high_score:
  categories:
```

Now you need to add sub-entries in the categories section for each player variable you'd like to track high scores for, and then under those, the names (or "labels") you want to use to refer to each position. If you want to track the player variable called `score` (which is the score, so that makes sense), add a `score:` section, and then under there create an entry for what you want to call each score position. For example:

```
categories:
- score:
  - GRAND CHAMPION
  - HIGH SCORE 1
  - HIGH SCORE 2
  - HIGH SCORE 3
  - HIGH SCORE 4
```

The sub entry labels correspond to the ranking order of them, so the player with the highest `score` player variable will get an award named `GRAND CHAMPION`, the second-highest `score` player variable will be assigned `HIGH SCORE 1`, and so on. Feel free to call these whatever you want or to add as many or as few as you want. The number of entries in the list is how many of each of the top values will be saved.

This is also where you configure the high score system to track the highest results of other player variables. For example, if you want to track the player variable called `loops` with a single award called `LOOP CHAMP`, you'd enter that here too, like this:

```
#config_version=3

high_score:
  categories:
    - score:
      - GRAND CHAMPION
      - HIGH SCORE 1
      - HIGH SCORE 2
      - HIGH SCORE 3
      - HIGH SCORE 4
    - loops:
      - LOOP CHAMP
```

Notice that all of these have dashes in front of them (with a space between the dash and the word). This is a YAML formatting thing which tells the YAML processor that these items are in a list that should be processed in order. (Not only does the order of the individual placement labels correspond to the name of the position on the list, but the order of the player variables in the categories list corresponds to the order they'll be processed when your high score mode is running and collecting the players' names or initials.)

The keys to remember are:

- The name of the section in the categories entry corresponds to a player variable of the same name.
- The sub-entries under a player variable are the labels for each slot.
- The order of the sub-entries correspond to the order the labels will be applied to each position.
- The number of sub-entries you have controls how many positions will be saved for each category.

Note that at this point, you are not configuring settings like how many characters a player can enter, how many of these entries are shown in your attract mode, or whether the player wins free credits for an entry. All you're doing now is deciding what player variables you'll track, how many positions of each variable you'll track, and what those positions are called.

(D) Add the high score mode to your list of modes

Now that you have some basic high score settings configured, you can add the high score mode to the list of modes that are used in your machine. To do this, add `- high_score` to the *modes*: section in your machine-wide config, like this:

```
modes:
  - base
  - some_existing_mode
  - another_mode_you_might_have
  - tilt
  - bonus
  - high_score
```


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 high score mode as a mode that your machine will use.

You might be wondering why your new *high_score.yaml* mode configuration file doesn't have a *mode:* section? That's because the *high_score* mode is built-in to MPF (in the *mpf/modes/high_score*) folder, so when you add a *high_score* 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 *high_score* mode's config (at *mpf/modes/high_score/config/high_score.yaml*), you'll see it has the following *mode:* section:

```
mode:
  code: high_score.HighScore
  priority: 500
  start_events: game_ending, start_high_score
  use_wait_queue: true
```

So those are the base settings for the *high_score* mode which will be applied unless you override them in your own machine's *high_score* config file.

Notice that this configuration has the high score mode configured to automatically start when the *game_ending* event is posted. When the high score mode starts, it will look at all the values of the player variables for the high score categories you configured and compare them to the existing high scores. If any players from the current game have achieved a high score in any category, the high score mode will kick off events to collect the player's initials and display the high score screens. If no players in that game have achieved a high score, the high score mode will stop and the game ending process will continue.

In other words, the high score mode always starts whenever a game ends, but if no player has achieved a high score, then the high score mode stops again. This whole process only takes a few milliseconds.

(E) Configure your display slides to allow high score entry

After your high score mode is added, you need to add a *slide_player:* entry in your high score config file so that it displays the high score slides in the format that you like. If you have a 128x32 DMD-based game, you can probably use the sample configuration pretty much as it is. If you're using an HD display or an LCD window, you can still use the default config as a starting point, but you'll most like want to tweak some things.

We'll go through the *slide_player:* sections one-by-one here. There's a complete *high_score.yaml* file at the end of this How To guide which you can copy to use as your starting point.

Understanding the high score slides

When the high score mode detects that a player has achieved a high score, it will post an event called *high_score*. (If the player has achieved multiple awards, or multiple players have achieved awards, these will be sent one-by-one. More on that in a bit.)

This *high_score* event will have several parameters, including:

- *award* - The name (label) of the award that the player got, like GRAND CHAMPION or ALIEN KING or whatever you have configured in the *categories:* section of your *high_score:* config.
- *player_num* - The number of the player who earned that award.
- *value* - The value of player variable for the award. (e.g. their score or the number of aliens they got or the total loops or whatever)

At the most basic level, you can add the *high_score* event to your *slide_player:* configuration which means you can use the parameters it passed to build the display slide the player will use to enter their initials. This is no different than any other slide which shows dynamic data (like your score display).

The "magic" in the high score slide is that you'll use two interactive types of display elements: the *character_picker* element and the *entered_chars* element. (Remember that [display elements](#) are the "things" you put on a slide, like *text*, *image*, *animation*, etc. *Character_picker* and *entered_chars* are just two more types of display elements that you can add to any slide, and in this case you'll use them for the high score.)

Here's how the template *high_score* slide is built:



There are four display elements on this slide (and in the *high_score* slide from the template you're basing your mode on), so let's look at each of these one-by-one:

The player number text element

The first display element on the slide is just a regular [text element](#) like any other text element. It uses the *player_num* event parameter to dynamically show the player number for the player who's entering their name. You can set the size, position, font, decorators, and anything else you want just like any other text display element.

Here's what the template includes. This is the red section in the image above. Feel free to change it as you wish.

```
- type: text
  text: PLAYER %player_num%
  font: medium
  v_pos: bottom
  h_pos: center
  x: -27
  y: -21
```

The award name text element

The award name (the green box in the image above) is also just a regular text display element that basis its text on the award parameter passed along with the *high_score* event:

```
- type: text
  text: "%award%"
  font: small
  v_pos: bottom
  h_pos: center
  x: -27
  y: -12
```

The character picker element

The *character picker* display element (the blue box in the image) is what shows the list of characters the player can chose from to enter their name or initials. It also has settings for the spacing, the list of characters, and related settings. Here's what the template configuration contains. See the documentation for the [character picker display element](#) for details on what all these settings do.

```
- type: character_picker
  name: high_score
  slide_name: high_score
  clear_slide: true
  persist: no
  height: 9
  font: medium
  v_pos: bottom
  selected_char_color: 0
```

```

selected_char_bg: 15
char_x_offset: 1
char_y_offset: 1
char_width: 7
char_list: "ABCDEFGHIJKLMNOPQRSTUVWXYZ_ - "
back_char: back_arrow_7x7
end_char: end_11x7
back_char_selected: back_arrow_7x7_selected
end_char_selected: end_11x7_selected
image_padding: 1
shift_left_tag: left_flipper
shift_right_tag: right_flipper
select_tag: start
max_chars: 3
timeout: 30s
return_param: award

```

The entered chars element

The *entered chars* (short for "entered characters") display element (the purplish-pink box in the image above) is used to show the characters that the player has selected so far. It also includes a flashing cursor character showing the current spot they're picking a character for.

Like the other display elements, you can make this whatever font you want, and set its size and position like any display element.

Note that in the config snippet below, the *h_pos*: is *left* (even though the element itself is towards the right side of the display). That's so that the text in the element is left-justified, so we position the display element *left* and then use the *x*: setting to move the element where we want it (at the 90th pixel from the left edge of the display).

The only other important thing about the entered chars display element is that it has a setting for *character_picker* which is where you specify the name of the character picker display element which is the "source" for the entered chars. So notice that *character_picker: high_score* matches *name: high_score* from the *character_picker* settings above.

See the documentation on the [entered chars display element](#) for full details.

```

- type: entered_chars
  character_picker: high_score
  cursor_char: _
  v_pos: bottom
  h_pos: left
  x: 90
  y: -12
  cursor_offset_x: 0
  cursor_offset_y: 0
  cursor_decorators:
    type: blink

```

(F) Configure your high score award slide

Once the player enters their name or initials, MPF will then post two events you can use for the high score award display slide. The high score award display is the slide that is shown for a few seconds after the player enters their initials. For example: "*GRAND CHAMPION: BRI*" (or whatever you want).

Two events are posted. One is always called *high_score_award_display*, and the other is dynamically created based on the award name itself in the form *<award_name>_award_display* (e.g. *grand_champion_award_display*). The reason there are two is in case you want to have custom award display slides for each kind of award. Otherwise you can just create a single award display slide based on the generic *high_score_award_display* event.

Both of these events have the same parameters which you can use in your award slide:

- *player_name* - The name (or initials) the player just entered in the high score entry screen.
- *award* - The name (label) of the award that the player got, like GRAND CHAMPION or ALIEN KING or whatever you have configured in the categories section of your high_score config.
- *value* - The value of player variable for the award. (e.g. their score or the number of aliens they got or the total loops or whatever)

The high score mode template config file contains a single entry in the slide_player for award slides:

```
high_score_award_display:
- type: text
  text: "%player_name%"
  color: 0
  bg_color: 15
  v_pos: center
  y: 2
  decorators:
    type: blink
    on_secs: .05
    off_secs: .05
- type: text
  text: "%award%"
  font: medium
  v_pos: top
  y: 2
```

This slide will show the award in the top row with the player's name under it. You can customize this however you want. Note that the time this slide is shown is controlled in the *high_score: settings*, via the *award_slide_display_time:* setting. The template mode sets this to 4 seconds. This is a setting of the high score mode, rather than a slide expiration setting,

because the high score mode needs to know how long to delay itself before moving on to the next award entry or finishing the game ending process.

(G) Add tags to your switches

The high score mode requires three switches to be configured. One is to move the cursor to the left, another to move the cursor to the right, and a third to select the highlighted character. MPF uses switch tags for this. By default, it's configured to look for tags called *left_flipper*, *right_flipper*, and *start*.

Chances are you already have these switches tagged with these names, but if not, go into your machine-wide config (not your high_score config) and into the *switches:* section and add those tags, like this:

```
switches:
  s_flipper_lower_right:
    number: sf2
    tags: right_flipper
  s_flipper_lower_left:
    number: sf4
    tags: left_flipper
  s_start:
    number: s13
    tags: start
```

(H) Copy or create images for the 'back' and 'end' characters

Notice in the character_picker display element that there are four image files referenced:

```
back_char: back_arrow_7x7
end_char: end_11x7
back_char_selected: back_arrow_7x7_selected
end_char_selected: end_11x7_selected
```

The full descriptions of these are in the character_picker config file reference, but the quick overview is these are the images used for the "back" and "end" characters in both their selected and unselected states. (Since the character picker uses font characters, most fonts don't have appropriate entries for back and end, so you have to draw your own.)

If you're using a DMD, then you can copy the images folder from the built-in *high_score* mode folder into your own *high_score* mode folder.

Source folder to copy:

```
mpf/modes/high_score/images
```

Where to put it:

<your_machine>/modes/high_score/images

If you have an HD display or your using some other font, then you can create these four images on your own, add them to your *high_score/images* folder, and then add the names of the files to your character_picker configuration.

(I) Check out this complete config file

Here's a complete *high_score.yaml* config file. This is what's used in the demo_man sample game.

```
#config_version=3

high_score:
  categories:
    - score:
      - GRAND CHAMPION
      - HIGH SCORE 1
      - HIGH SCORE 2
      - HIGH SCORE 3
      - HIGH SCORE 4

slide_player:
  high_score:
    - type: text
      text: PLAYER %player_num%
      font: medium
      v_pos: bottom
      h_pos: center
      x: -27
      y: -21
    - type: text
      text: "%award%"
      font: small
      v_pos: bottom
      h_pos: center
      x: -27
      y: -12
    - type: character_picker
      #width: 50
      name: high_score
      slide_name: high_score
      clear_slide: true
      persist: no
      height: 9
      font: medium
      v_pos: bottom
      selected_char_color: 0
      selected_char_bg: 15
      char_x_offset: 1
      char_y_offset: 1
      char_width: 7
      char_list: "ABCDEFGHIJKLMNOPQRSTUVWXYZ - "
      back_char: back_arrow_7x7
      end_char: end_11x7
```

```

back_char_selected: back_arrow_7x7_selected
end_char_selected: end_11x7_selected
image_padding: 1
shift_left_tag: left_flipper
shift_right_tag: right_flipper
select_tag: start
max_chars: 3
timeout: 30s
return_param: award
- type: entered_chars
character_picker: high_score
cursor_char: _
v_pos: bottom
h_pos: left
x: 90
y: -12
cursor_offset_x: 0
cursor_offset_y: 0
cursor_decorators:
  type: blink

high_score_award_display:
- type: text
  text: "%player_name%"
  color: 0
  bg_color: 15
  v_pos: center
  y: 2
  decorators:
    type: blink
    on_secs: .05
    off_secs: .05
- type: text
  text: "%award%"
  font: medium
  v_pos: top
  y: 2

```

Note that there are additional settings you can configure in the *high_score*: section of your config. Refer to the [high_score:page](#) in the config file reference for details.

(J) Add high scores to your attract mode display show

The final thing you'll probably want to do if you're adding high scores to your machine is to configure your attract mode display show to include the various high scores. (Creating an attract mode display show is [one of the steps](#) in the getting started tutorial.)

Creating slides for your attract mode slide show with high scores is pretty straightforward. High scores are saved as machine variables, so you access them via text display elements showing machine variables just like any machine variable. (The how to guide which shows you [how to add the scores from the last-played game](#) has more details on this.)

The exact names of the machine variables you'll use are dynamically created based on the player variable of the award as well as position in the list. They have the format like this:

- `<player_variable_name><position_in_list>_name`
- `<player_variable_name><position_in_list>_value`
- `<player_variable_name><position_in_list>_label`

For example, for the high score based on the player variable "score", if the highest score in the machine is "BRI" with a value of 7050550, then the machine variables will be `score1_name = BRI`, `score1_value = 7050550`, and `score1_label = GRAND CHAMPION`. The second-highest "score" will be `score2_name`, `score2_value`, with `score2_label = HIGH SCORE 1`. If you also tracked a high score entry for "ramps" then the highest scoring "ramps" will be `ramps1_name` and `ramps1_value`, etc.

Since these machine variables are just accessed in a slide like any regular display element, you can set the font, size, position, number grouping, etc. like any [text display element](#).

Here are some examples from the *demo_man* sample game you can use as a starting point for your own machine:



The entry for this slide in the attract mode YAML file looks like this:

```
- tocks: 2
  display:
    - type: text
      text: "%machine|score1_label%"
      v_pos: top
      y: 4
    - type: text
      text: "%machine|score1_name% %machine|score1_value%"
      v_pos: bottom
      number_grouping: true
      y: -3
    - type: shape
      shape: box
      width: 128
      height: 32
    - type: shape
      shape: box
      width: 126
      height: 30
      shade: 8
```

Note that the value of the score is *7050550*, so in order for it to include commas, we add *number_grouping: true*. (Again, just like any other text display element.)

Here's an example showing high scores 1 and 2:



And the related entry for the show YAML file:

```
- tocks: 2
  display:
    - type: text
      text: "1. %machine|score2_name% %machine|score2_value%"
      v_pos: top
      h_pos: left
      number_grouping: true
      y: 3
      x: 12
    - type: text
      text: "2. %machine|score3_name% %machine|score3_value%"
      v_pos: bottom
      h_pos: left
      number_grouping: true
      y: -3
      x: 10
    transition:
      type: move_out
      duration: 1s
      direction: top
```

Notice in this case that we don't use the *score3_label* machine variable. That's because the label value is *HIGH SCORE 1*, but in this case that would be too long for the slide. We don't want the slide to show *HIGH SCORE 1: BRI 93,060*, rather, we want it to show *1. BRI 93,060*. So we just manually enter the "1. " in the text field and use the machine variables to dynamically provide the content for the player's name and their score.

Of course you can add any high score result to your attract mode. Here's an example slide showing the loops champion:



And here's the config you'd add for that in your attract mode display show:

```
- tocks: 2
  display:
    - type: text
      text: MASTER LOOPER
      v_pos: top
      font: medium
    - type: text
      text: "%machine|loops1_name%"
      font: tall title
    - type: text
      text: "%machine|loops1_value% LOOPS"
      v_pos: bottom
      font: medium
      number_grouping: true
      transition:
        type: move_in
        duration: 1s
        direction: bottom
```

How To: Add Coins & 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!

Features of the credits mode include:

- 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.

Here are some screen shots of the credits mode in action:



Now let's dig in to actually getting this set up:

(A) Create your 'credits' mode folder

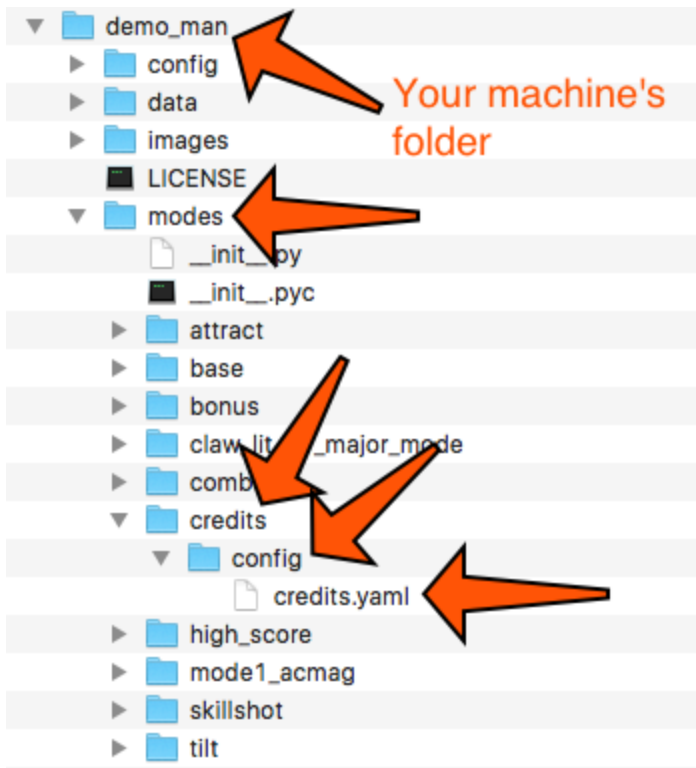
The credits mode works like any other mode in MPF. You'll create a folder called *credits* in your machine's *modes* folder, and that folder will contain subfolders config files, images, etc.

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*.)

Your folder structure should look something like this:



(B) Configure options for the credits mode

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=3` to the top line.

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 `credits.yaml` file should now look like this:

```
#config_version=3

credits:
```

Next you need to add the settings for the credits system. You can use the following as a starting point:

```
credits:
  max_credits: 12
  free_play: no
  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

```

Full details of what each of these settings does is outlined in the [credits:section](#) 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
  - some_existing_mode
  - another_mode_you_might_have
  - bonus
  - credits
```

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:
  code: credits.Credits
  priority: 11000
  start_events: machine_reset_phase_3
  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

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:

```
slide_player:
  credits_added:
    type: text
    text: "%machine|credits_string%"
    expire: 2s
  not_enough_credits:
    - type: text
      text: "%machine|credits_string%"
      expire: 2s
      font: small
      v_pos: bottom
```



```

- type: text
  text: INSERT COINS
  decorators:
    type: blink
    repeats: -1
    on_secs: .1
    off_secs: .1
  enabling_free_play:
    type: text
    text: ENABLING FREE PLAY
    expire: 2s
  enabling_credit_play:
- type: text
  text: ENABLING CREDIT PLAY
  expire: 2s
- type: text
  text: "%machine|credits_string%"
  expire: 2s
  font: small
  v_pos: bottom

```

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, like this:



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 Demo Man](#) for details. (Also check out [Demo Man's DMD slide show](#) for more examples.)

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) 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`):

```
#config_version=3

credits:
```

```

max_credits: 12
free_play: no
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: token
    value: 1
  - switch: s_fourth_coin
    value: 1
    type: money
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

slide_player:
  credits_added:
    type: text
    text: "%machine|credits_string%"
    expire: 2s
  not_enough_credits:
    - type: text
      text: "%machine|credits_string%"
      expire: 2s
      font: small
      v_pos: bottom
    - type: text
      text: INSERT COINS
      decorators:
        type: blink
        repeats: -1
        on_secs: .1
        off_secs: .1
  enabling_free_play:
    type: text
    text: ENABLING FREE PLAY
    expire: 2s
  enabling_credit_play:
    - type: text
      text: ENABLING CREDIT PLAY
      expire: 2s
    - type: text
      text: "%machine|credits_string%"
      expire: 2s
      font: small
      v_pos: bottom

```

How To: Show scores in the attract DMD mode

One of the things "real" pinball machines do is show the scores from the last game on the DMD during the attract mode. So that's what we're going to cover in this tutorial.

Also note that this tutorial is just for showing the old scores of the last game in the attract mode. You can also show the live scores of other players while a game is in progress, but that's a different tutorial which we'll write soon.

To do this, you need to add "machine variables" into your attract mode DMD show file.

(A) What are machine variables?

You might already be familiar with the concept of player variables. Player variables are variables that are stored on a per-player basis in MPF. A complete game might have hundreds variables per player, ranging from simple things like score and ball number, to modes complete, shot status, logic blocks, timers, etc.

Machine variables are similar in concept to player variables, except instead of being stored on a per-player basis, they're stored for the whole machine. So what kinds of variables are properties of the machine? Things like the number of credits, the high scores, and (in this case), the scores from the last game that was played.

(B) Using machine variables in slides

You might know that you can use the `%variable_name%` format in a `slide_show:` or `slide_player:` configuration to access player variables from that slide. Machine variables are pretty much the same, except you add `machine|` in front of the variable name.

So if you want a slide to display the value of a machine variable called `player1_score`, then you would add an entry `text: "%machine|player1_score%"` to your slide config. It's that simple!

In fact that's all you really need to know. Check out this slide that we added to the *Demo Man* `attract_dmd_loop.yaml` display show: (How do you create this attract mode DMD show? [We have a tutorial for that!](#))

```
- tocks: 3
  display:
    - type: text
      text: "%machine|player1_score%"
      number_grouping: true
      min_digits: 2
      font: small
      v_pos: top
      h_pos: right
```

```

x: -80
y: 1
- type: text
text: "%machine|player2_score%"
number_grouping: true
min_digits: 2
font: small
v_pos: top
h_pos: right
x: -1
y: 1
- type: text
text: "%machine|player3_score%"
number_grouping: true
min_digits: 2
font: small
v_pos: bottom
h_pos: right
x: -80
y: -10
- type: text
text: "%machine|player4_score%"
number_grouping: true
min_digits: 2
font: small
v_pos: bottom
h_pos: right
x: -1
y: -10
- type: text
text: FREE PLAY
font: small
h_pos: center
v_pos: bottom

```

We use the display element positioning (`h_pos`, `v_pos`, `x`, and `y`) to get everything set how we like it. (Details on how to do that are [here](#).) Note that we use `h_pos: right` for everything and then set the because we want all the score to be right-aligned (even the ones on the left where we just have a high negative `y` value).

Here are the results:

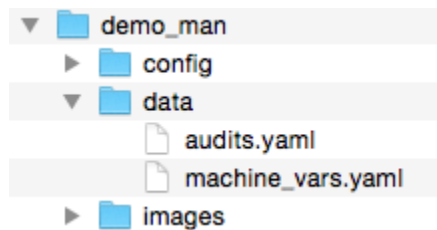


(Note that there are periods instead of commas. That's because the font we're using doesn't have a comma. :(We'll find a new font at some point, or [you can add your own](#).)

One of the cool things with the way machine variables work is that if the machine variable doesn't exist, then MPF will not render it to the display. In other words if the last game you played only had three players, then you would only see three scores.

Another cool thing about machine variables is they have the option to be persisted to disk. That means they can stick around even when MPF is turned off! (This again is something that "real" pinball machines do. When you turn them on, they show the scores from the last game even if the machine was off for a week.) Not all machine variables are persisted to disk, but scores from the last game played are.

When machine variables are persisted to disk, MPF them in a file in your machine folder: `<your_machine_root>/data/machine_vars.yaml`. (Note that in MPF 0.21, we moved `audits.yaml` to that folder also.)



If you take a look at the `machine_vars.yaml` file, you'll see that it looks pretty similar to your YAML config files:

```
player1_score: !!python/object/new:mpf.system.config.CaseInsensitiveDict
  dictitems:
    value: 56473820
player2_score: !!python/object/new:mpf.system.config.CaseInsensitiveDict
  dictitems:
    value: 34586740
player3_score: !!python/object/new:mpf.system.config.CaseInsensitiveDict
  dictitems:
    value: 1209800
player4_score: !!python/object/new:mpf.system.config.CaseInsensitiveDict
  dictitems:
    value: 88759490
```

When MPF boots, it loads the machine variables from disk into memory. When a game starts, MPF erases all the player score entries from the file on disk, so if your last game was a four-person game and then you play a one-player game, the `machine_vars.yaml` file will only contain the score for the first player.

Then when you start a game (and when you add a player), MPF writes the new score value (of 0) to disk. It also updates the file on disk each time a player's turn ends.

How To: Create a Bonus mode

In MPF, there are many situations where you might want to interrupt the flow of progress in order to do something. For example, you might want to hook the ball ending process to run a bonus mode, you might want to hook the ball starting process to play some animation, you might want to hook a ball entering a device to play a show before multiball starts, etc.

At first you might think, "No problem, I'll just create a bonus mode that runs when the *ball_ending* event is posted." Unfortunately that won't work because MPF will just start the next ball, so your bonus mode will be running while the next player's turn has started and the ball is sitting at the plunger. So how do you handle this?

This is done by leveraging MPF's "queue" events. A queue event is just like other event in MPF with one key addition: registered handlers of queue events can respond to the event by telling it, "Hey, I've got some things to do, so don't proceed until I tell you that I'm done." So that's what we're going to do in this tutorial. (More info on queue events is [here](#).)

This tutorial is specifically about how you'd hook a bonus mode into the ball ending process, but the techniques here can be used with any of the queue events in MPF. Here are some examples of queue events:

- `game_starting`
- `game_ending`
- `ball_starting`
- `ball_ending`
- `mode_<name>_starting`
- `mode_<name>_stopping`

Notice that all of these event names end in *-ing*. These events are all posted as queue events, and then when the queue is cleared, the related *-ed* version of the event is posted. (So *ball_ending* is posted as a queue event, and then when that queue is clear, *ball_ended* is posted.)

Registering "waits" via the mode config file

Fortunately it's easy to configure a mode so that it registers a wait. You can do this right in the config file for your mode with the `use_wait_queue: true` setting. For example, here's how we'd do this in a bonus mode config that we want to hook the *ball_ending* event:

```
mode:
  start_events: ball_ending
  stop_events: timer_bonus_complete
  priority: 500
  use_wait_queue: true
```


Since this mode has `start_events: ball_ending`, it will start when the *ball_ending* event is posted. And since it has `use_wait_queue: true`, it will "pause" the ball ending process as long as it's running. To "release" the pause, all you have to do is end the mode.

Here's a very simple example, all done in the config (i.e. with no custom mode code), that will display the word "BONUS" on the display for 3 seconds during the ball ending process. This is the complete mode config file:

```
#config_version=3

mode:
  start_events: ball_ending
  stop_events: timer_bonus_complete
  priority: 500
  use_wait_queue: true

slide_player:
  mode_bonus_started:
    type: text
    text: BONUS

timers:
  bonus:
    start_value: 3
    start_running: true
    direction: down
```

Of course this isn't terribly useful for a ball end bonus since you'd probably want to do some math and add a score, but you get the idea. (And you could use this exact same technique, for example, to play a slide show and some sounds as an introduction to *mode_<name>_starting* event.)

Working with "waits" in custom mode code

So now that you have a useless bonus mode that starts and stops, lets look at adding some custom mode code to actually do something useful.

To do this, we'll add a `code/bonus.py` file into our `bonus` mode folder. (Remember that you also have to add blank `__init__.py` files to your `mode` and `code` folders.)

Then update the config for your bonus mode to specify that you're using custom code, like this:

```
mode:
  start_events: ball_ending
  code: bonus.Bonus
  priority: 500
  use_wait_queue: true
```

(Note that we also removed the `stop_events:` entry from the mode config since we'll stop our mode in code.)

Next let's take a look at the `bonus.py` mode code file. Here's a very simple (and fully functional) file. In this example, we register an event to set the player variable for the bonus multiplier to 1 when a new player is created. We also set it to check for tilt when it starts and to immediately stop if the machine is tilted.

In this example, we're pulling player variables for ramps, modes, and `bonus_multiplier`. Then we spit out a series of events, one-by-one, along with the math to add up the player's score. When we get to the end, we reset the player variable for the bonus multiplier (after making sure the player doesn't have hold bonus), and then we call `self.stop()` which ends the mode. Since we have this mode configured with the `use_wait_queue`, when the mode finishes its stop process the hold will be released and MPF will move on.

Here's the `bonus.py` code:

```
from mpf.system.mode import Mode

class Bonus(Mode):

    def mode_init(self):
        self.machine.events.add_handler('player_add_success',
                                       self.player_add)

    def player_add(self, player, **kwargs):
        player['bonus_multiplier'] = 1

    def mode_start(self, **kwargs):

        if self.machine.game.tilted:
            self.stop()

        self.bonus_score = 0
        self.bonus_start()

    def bonus_start(self):
        self.machine.events.post('bonus_start')
        self.delay.add(name='bonus', ms=2000, callback=self.total_ramps)

    def total_ramps(self):
        self.machine.events.post('bonus_ramps')
        self.bonus_score += self.player['ramps'] * 10000
        self.delay.add(name='bonus', ms=2000, callback=self.total_modes)

    def total_modes(self):
        self.machine.events.post('bonus_modes')
        self.bonus_score += self.player['modes'] * 50000
        self.delay.add(name='bonus', ms=2000, callback=self.subtotal)

    def subtotal(self):
        self.machine.events.post('bonus_subtotal', points=self.bonus_score)
        self.delay.add(name='bonus', ms=2000, callback=self.do_multiplier)
```

```

def do_multiplier(self):
    self.machine.events.post('bonus_multiplier')
    self.delay.add(name='bonus', ms=2000, callback=self.total_bonus)

def total_bonus(self):
    self.bonus_score *= self.player['bonus_multiplier']
    self.player['score'] += self.bonus_score
    self.machine.events.post('bonus_total', points = self.bonus_score)
    self.delay.add(name='bonus', ms=2000, callback=self.end_bonus)

def end_bonus(self):
    if not self.player['hold_bonus']:
        self.player['bonus_multiplier'] = 1
    else:
        self.player['hold_bonus'] = False

    self.stop()

```

Next let's take a look at the bonus.yaml config file. It's pretty straightforward, with slide_player settings that put slides on the display for each event from the code. Note that some of the slides pull their data from player variables (via the %variable_name%), and others pull them from event parameters (via a single percent sign).

```

#config_version=3

mode:
  start_events: ball_ending
  code: bonus.Bonus
  priority: 500
  use_wait_queue: true

slide_player:
  bonus_start:
    type: text
    text: END OF BALL BONUS
  bonus_ramps:
    type: text
    text: "RAMPS: %ramps%"
  bonus_modes:
    type: text
    text: "MODES: %modes%"
  bonus_subtotal:
    type: text
    text: "BONUS SCORE: %points"
    number_grouping: true
  bonus_multiplier:
    type: text
    text: "%multiplier%X"
  bonus_total:
    type: text
    text: "TOTAL BONUS: %points"
    number_grouping: true

```

Extending this code

At this point you should have a fairly basic bonus structure in place, and there are lots of ways you can extend it:

- Add light shows and music.
- Add some code to detect if the player presses both flippers at the same time to cancel or to hurry up the timing of the delays.
- Decide which player variables you want to reset on each ball versus which ones you keep.
- Add more flourish to the slide_player with different fonts, shapes, images, transitions, etc.

How To: Use FadeCandy for LEDs

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 read more about it on the main page of the [FadeCandy software repository](#) in GitHub or on [Adafruit](#) or [SparkFun](#), where you can buy a FadeCandy for \$25.

The FadeCandy is very advanced, offering advanced light processing capabilities (dithering, interpolation, adjustable gamma curves and white balance) that are not available if you just control LEDs directly. If you're not familiar with the FadeCandy, check out this intro from SparkFun:

<https://www.youtube.com/watch?v=-4AUBjV7Y-w>

This How To guide will show you how to setup and use a FadeCandy with the Mission Pinball Framework.

In MPF, you can use the FadeCandy in place of connecting your LEDs to a Multimorphic (P-ROC / P3-ROC) controller, or you can choose to drive some LEDs via your primary pinball controller and some via the FadeCandy.

(A) 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 process that you'll run on your host computer along side of the MPF core engine and the MPF media controller. The FadeCandy server talks to the FadeCandy hardware via a USB driver.
- The FadeCandy server process 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.

(B) 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. It doesn't have to be in your MPF folder. (In fact it shouldn't be in your MPF folder so you can blow away your MPF folder as new releases come out.)

(C) Install the FadeCandy drivers

When I (Brian) 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.

(D) 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.

(D) 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: section](#) 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, there are two ways you can do this.

The first method is used if every LED you have will be connected to a FadeCandy. In this case you can add an entry to the hardware: section of your 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
  leds: fadecandy
```

Or, if you'd like to mix-and-match LEDs from your base platform and the FadeCandy platform, you can add a `platform:` entry to an actual LED configuration entry to override the default. So assuming that you do not have the `leds: fadecandy` entry in the hardware: section of your machine config, you could set up LEDs like this:

```
leds:
  l_some_led:
    number: 0
  l_some_other_led:
    number: 0
    platform: fadecandy
```

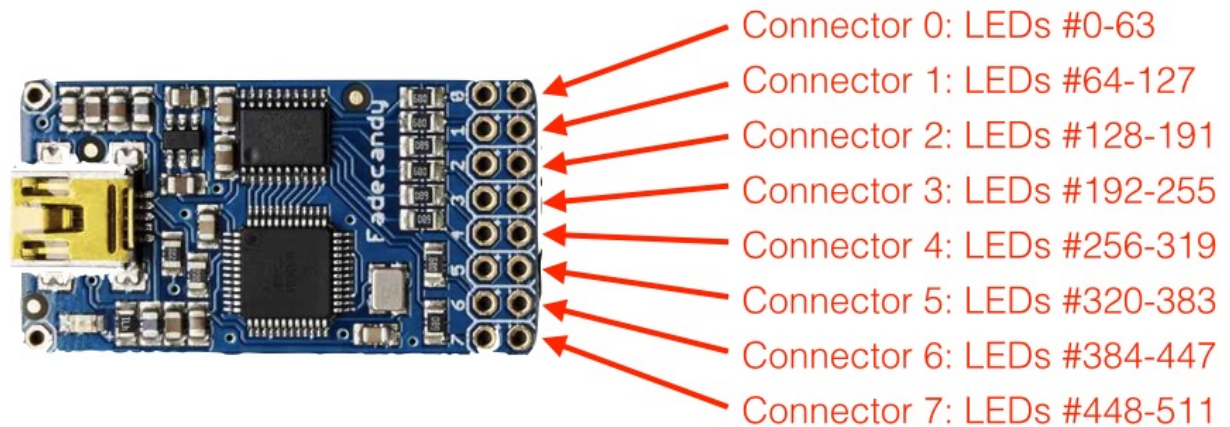
In the above example, `l_some_led` will be run from your base platform (P-ROC or FAST), and `l_some_other_led` will be a FadeCandy. Notice that both of these LEDs are number "0". That's ok, because the first one is LED #0 from your base platform and the second is LED #0 from your FadeCandy.

(If you have mostly FadeCandy LEDs and only a few on your base platform, you could flip that and add the `leds: fadecandy` to your hardware: section and then only override the specific ones on the base platform with `platform: p_roc` or whatever.)

(F) 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 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.



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. (By default this is OPC Channel 0. You can add additional FadeCandy controllers to support more than 512 LEDs which will use other OPC Channels. See the FadeCandy documentation for details.)

(G) Launch the fcserver

In order for MPF to communicate with the FadeCandy, the fcserver has to be running. (On my Windows test machine, this is just `fcserver.exe`.) There are several command line options you can use, but we 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 your `mpf.bat` to run it automatically. (You'll need to figure out how to get it to stop when MPF stops. I'm sure that's possible but I haven't looked into it.)

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.)

(H) Test your LEDs

The easiest way to test your FadeCandy LEDs with MPF is to make a simple attract mode light show with a few of them. We cover this in [this step of our tutorial](#).

(I) Additional FadeCandy LED options

The FadeCandy hardware supports some advanced options which are configured in the [led_settings: 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.

How To: Create a Tilt

This How To guide explains how to configure a tilt in MPF. The MPF package contains a the code for a tilt mode which runs at all times to watch for tilts, so all you have to do to use add some configs to your machine's *modes* folder and you're all set.

Features of the tilt mode include:

- Separate processing of tilt warning, tilt, and slam tilt events.
- Specify that several tilt warnings coming close together only count as one warning.
- Specify "settle time" that the tilt warning switch has to not activate in order to let it "settle" between players (so the next player after a super hard tilt doesn't get a tilt also because the plumb bob was still swinging).
- Configurable options for when a player's tilt warnings are reset (per game, per ball, etc.).
- Flexible events you can use to trigger display, sound, and lighting effects when tilt warnings occur.

Let's dig in to the process of actually getting the tilt setup.

(A) Create your 'tilt' mode folder

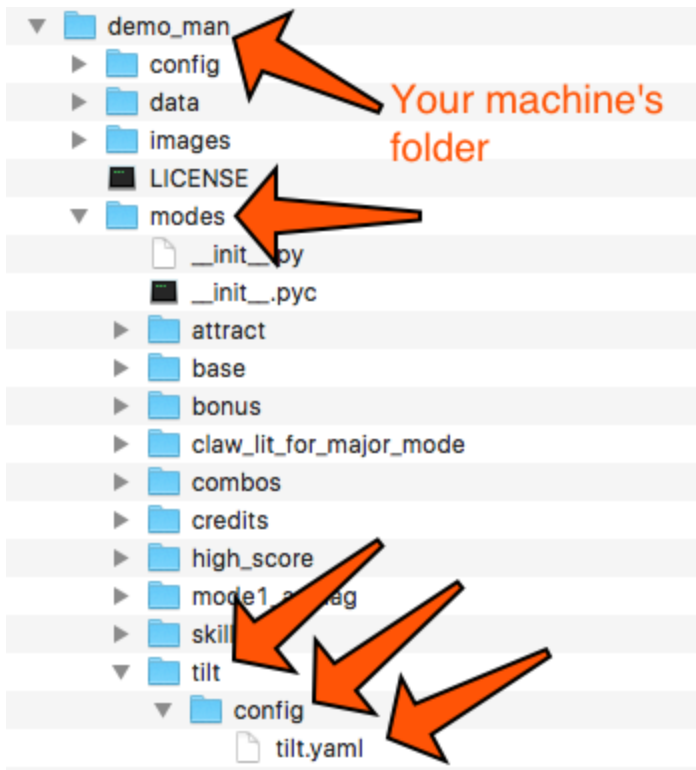
The tilt mode works like any other mode in MPF. You'll create a folder called *tilt* in your machine's *modes* folder, and that folder will contain subfolders config files, images, etc.

So to begin, create a folder called *<your_machine>/modes/tilt*.

Then inside there, create another folder called *config*.

Then inside there, create a file called *tilt.yaml*. (So that file should be at *<your_machine>/modes/tilt/config/tilt.yaml*.)

Your folder structure should look something like this:



(B) Configure options for the tilt mode

Open up the tilt mode's config file that you just copied into your machine folder. It should be at `<your_machine>/modes/tilt/config/tilt.yaml`.

Since this file is totally blank, add the required `#config_version=3` to the top line.

Next, add a section called `tilt;`, 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 `tilt.yaml` file should now look like this:

```
#config_version=3

tilt:
```

Next you need to add the settings for the tilt behavior in your machine. Here's a sample you can use as a starting point:

```
tilt:
  tilt_warning_switch_tag: tilt_warning
  tilt_switch_tag: tilt
  slam_tilt_switch_tag: slam_tilt
  warnings_to_tilt: 3
```

```

reset_warnings_events: ball_ended
multiple_hit_window: 300ms
settle_time: 5s
tilt_warnings_player_var: tilt_warnings

```

Full details of what each of these settings does is outlined in the [tilt: section](#) of the configuration file reference, so check that out for details on anything not covered here. It's all fairly self-explanatory.

First, notice that the switches to activate the *tilt*, *tilt warning*, and *slam tilt* are controlled by switch tags rather than by switch names themselves. (We'll add those in the next step.)

Warnings_to_tilt is how many warning switch activations lead to a tilt. The default value of 3 means that the player will have 2 warnings, since on the third activation the tilt will occur.

The *multiple_hit_window* means that multiple tilt warning events within this time window will only count as a single warning. This is a non-sliding window, meaning that if the window is 300ms and a tilt warning comes in at 0 (the first one), then 250ms after that, then 60ms after that—that will actually count as 2 warnings. (The second hit at 250ms is within the 300ms window, but the third hit 60ms later is actually 310ms after the first one which started the window, so it counts.)

The *settle_time* is the amount of time that must pass after the last warning before the tilt will be cleared and the next ball can start. This is to prevent the situation where a player tilts in an aggressive way and the plumb bob is bouncing around so much that it causes the next player's ball to tilt too. So when a tilt occurs, if a ball drains before the *settle_time* is up, MPF will hold the machine in the tilt state until it settles, then proceed on to the next player and/or the next ball.

The *reset_warnings_events* is the event in MPF that resets the tilt warnings to 0. The sample config from above means that the tilt warnings are reset when the ball ends. In other words with a *warnings_to_tilt* of 3, the player gets three warnings per ball. If you want to make the warnings carry over from ball-to-ball, you could change the *reset_warnings_events* to *game_ended*.

(C) Add switch tags

Since the tilt mode uses switch tags instead of switch names, you need to go into your machine-wide configuration and add the tags to the various switches you want to do tilt-related things.

In most modern games, you'll just use the *tilt_warning* and *slam_tilt* tags, like this (from your machine-wide config):

```

switches:
  s_plumb_bob:

```

```

number: s14
label:
tags: tilt
s_slam_tilt:
number: s21
label:
tags: slam_tilt

```

(D) Add the tilt mode to your list of modes

Now that you have the tilt settings configured, you can add the tilt mode to the list of modes that are used in your machine. To do this, add `- tilt` to the `modes:` section in your machine-wide config, like this:

```

modes:
- base
- some_existing_mode
- another_mode_you_might_have
- credits
- bonus
- tilt

```

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 tilt mode as a mode that your machine will use.

You might be wondering why your new `tilt.yaml` mode configuration file doesn't have a `mode:` section? That's because the `tilt` mode is built-in to MPF (in the `mpf/modes/tilt` folder, so when you add a `tilt` 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 `tilt` mode's config (at `mpf/modes/tilt/config/tilt.yaml`), you'll see it has the following `mode:` section:

```

mode:
code: tilt.Tilt
priority: 10000
start_events: machine_reset_phase_3
stop_on_ball_end: False

```

First is that the priority of this mode is really high, 10000 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 tilt warnings) are displayed on top of the slides from any other mode that might be running.

Also note that the tilt 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 tilt mode to start and never stop. (We even want it to run during attract mode so it can look for slam tilts.)

(E) Add slides and lighting effects

There are several events posted by the tilt mode, including:

- *tilt_warning* – a switch with the tilt warning tag was activated outside of the multiple hit window, and the player’s tilt warnings has just increased.
- *tilt_warning_<x>* – Same as tilt warning, but the “x” is the number of the warning. This lets you put different slides on the display for *tilt_warning_1* versus *tilt_warning_2*, etc.
- *tilt* – The machine has tilted.
- *tilt_clear* – The tilt is cleared, meaning all the balls have drained and the *settle_time* has passed.
- *slam_tilt* – The machine has slam tilted.

You can use this events to tell the player what's going on. For example, the configuration from the tilt mode template includes the following:

```
slide_player:
  tilt_warning_1:
    type: text
    text: WARNING
    expire: 1s
  tilt_warning_2:
    - type: text
      text: WARNING
      y: 2
    - type: text
      text: WARNING
      y: 18
      expire: 1s
  tilt:
    type: text
    text: TILT
  tilt_clear:
    clear_slides: yes
```

These slide player settings put a slide that says *WARNING* on the display for 1 second on the first warning, and a slide that says *WARNING WARNING* for the second warning. They also display a slide that says *TILT* when the player tilts.

Also note the *tilt_clear*: entry which clears out all the slides from the tilt mode when the tilt clears.

Since the tilt mode is running at priority 10,000, these slides should play on top of any other slides from other active modes.

You can change the fonts, placement, text, etc. of these slides or add other display elements as you see fit. You could also add *sound_player* or *light_player* sections if you wanted to play sounds or blink all the playfield lights. (To blink the playfield lights, create a light show with 1 step that turns off all the lights for a half-second or so.)

(F) Check out this complete tilt config file

Here's the complete tilt config file from the Demo Man sample game. (*demo_man/modes/tilt/config/tilt.yaml*):

```
#config_version=3

tilt:
  tilt_warning_switch_tag: tilt_warning
  tilt_switch_tag: tilt
  slam_tilt_switch_tag: slam_tilt
  warnings_to_tilt: 3
  reset_warnings_events: ball_ended
  multiple_hit_window: 300ms
  settle_time: 5s
  tilt_warnings_player_var: tilt_warnings

slide_player:
  tilt_warning_1:
    type: text
    text: WARNING
    expire: 1s
  tilt_warning_2:
    - type: text
      text: WARNING
      y: 2
    - type: text
      text: WARNING
      y: 18
    expire: 1s
  tilt:
    type: text
    text: TILT
  tilt_clear:
    clear_slides: yes
```

How To: Lane Change / Lit Shot Rotation

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.

<https://www.youtube.com/watch?v=4lp60PVe-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 the machine-wide configuration, 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:

```
shots:
  indy_i:
    switch: indy_i
    light: indy_i
  indy_n:
    switch: indy_n
    light: indy_n
  indy_d:
    switch: indy_d
    light: indy_d
  indy_y:
    switch: indy_y
    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:

```
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 our [Base mode config file](#), which is `base.yaml`. So open up `base.yaml` and add the following:

```
shot_groups:
  indy_lanes:
    rotate_left_events: left_flipper_active
    rotate_right_events: right_flipper_active
```

Note that the `indy_lanes:` section of this config is the name of the shot group that we specified in the `shot_groups:` section of the machine-wide config. So really we have the *indy_lanes* shot group configured in two places. We specify the member shots that make up the group in the machine-wide config, and then we specify the rotation events in the mode-specific config.

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:

```
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:

```
shot_groups:
  indy_lanes:
    rotate_left_events: left_flipper_active
    rotate_right_events: right_flipper_active
    reset_events:
      indy_lanes_default_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 `<shot_group_name>_<shot_profile_name>_<state_name>_complete`. In our case, our shot group name is *indy_lanes*, the shot profile we're using is called *default* (since we didn't assign a different profile to these shots), and the state of the shots that we're interested in is called *lit*.

Also notice that instead of adding `indy_lanes_default_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:

```
scoring:
  indy_lanes_default_unlit_hit:
    score: 5000
  indy_lanes_default_lit_hit:
    score: 100
  indy_lanes_default_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. (Brian was confused by this in the video!) 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`:

```
- tocks: 1
  lights:
    indy_i: ff
    indy_n: 00
    indy_d: ff
    indy_y: 00
- tocks: 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:

```
light_player:
  indy_lanes_default_lit_complete:
    show: indy_lanes_complete
    tocks_per_sec: 20
    repeat: yes
    num_repeats: 10
    priority: 1
```

If you've worked with shows before, these settings should be pretty straightforward. Running this show at 20 tocks per second means that it runs really fast. We set `repeat: yes` so the show loops and `num_repeats: 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_default_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_default_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.

How To: Create a Timed Skill Shot

This how to guide will show you how to create a timed skill shot. When the ball is launched, the player will have a few seconds to make a certain shot. If that shot is made before the timer runs out, the player will get a extra points. If they don't make that shot in time, the timer will run out and the skill shot mode will unload.

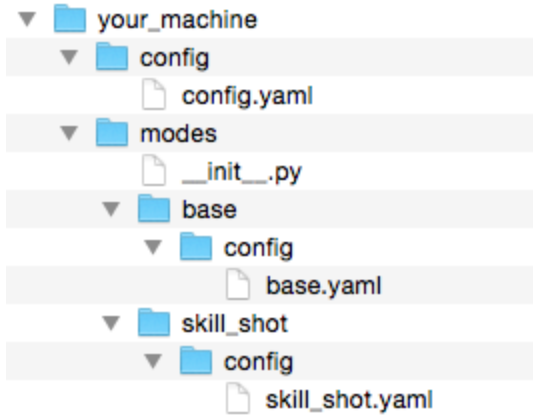
So let's dig in!

(A) Create your skill shot mode folder

The first step is to create the folder structure for your skill shot mode. This will be identical to what you did when you created your base mode and will include:

1. Create a `skill_shot` folder in your machine's modes folder.
2. Create a `config` folder in the `skill_shot` folder.
3. Create an empty file called `skill_shot.yaml` in the `modes/skill_shot/config` folder.

Your new folder structure should look something like this:



(B) Configure your skill shot mode settings

Next open the config file for the skill shot mode (*modes/skill_shot/config/skill_shot.yaml*) and set it up like this:

```
Mode:
  start_events: ball_starting
  stop_events: timer_skill_shot_complete
  priority: 300
```

Just like your base mode, configure the skill shot to start when the ball starts. What's different though is that we're also going to configure the skill shot mode so that it ends when an event called *timer_skill_shot_complete* is posted. (Remember that event name, because we're going to come back to it when we setup our timer which will end the skill shot mode when the time runs out.)

Also note that we're configuring the skill shot so it runs at a higher priority than the base mode. Our best practices are that modes priorities should be multiples of 100. That gives you enough space between the modes to adjust the priorities of specific things up or down within a mode without interfering with other modes. (This will become more apparent as you get deeper into the configuration of your game.)

(C) Configure a switch to be worth some extra points

Next pick a switch that will be the target of your skill shot. To keep things simple right now, let's use a switch called *target1*. Add a scoring entry to your skill shot mode's configuration that makes that switch worth 10,000 points.

In this case you can also add */block* to the end of the value which will prevent lower priority modes from scoring this event:

```
scoring:
  target1_active:
    score: 10000|block
```

Of course in the case of the skill shot, you might not actually want to block the scoring. Maybe you want the player to get the 10,000 points from the skill shot and also to get the 1,000 points from the base mode. In that case then don't add `/block`. But if you want the player to only get the 10,000 points, then you can block the lower priority modes from scoring this event.

(D) Add the skill shot mode to your machine config

Now go back to your machine-wide configuration (`your_machine/config/config.yaml`) and add the skill shot mode to the `Modes:` list. So now it should look like this:

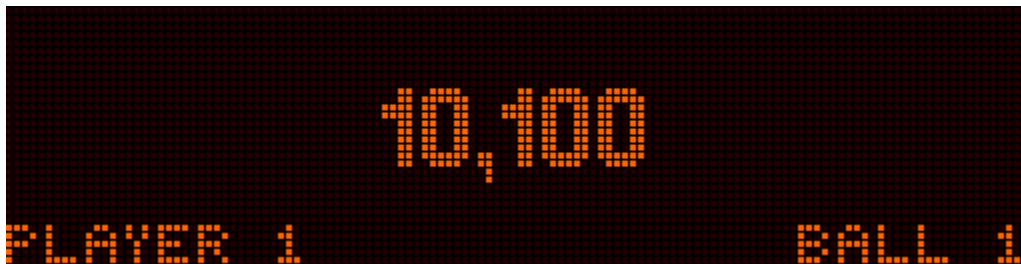
```
Modes:
- base
- skill_shot
```

(E) Run your game to make sure the skill shot is working

Be sure to save both of your config files and go ahead and run your game. Remember that your modes won't load until the first player's first ball starts, but if you push start (or hit the S key), you should see something like this in the console log:

```
INFO : ModeController : ===== ACTIVE GAME MODES =====
INFO : ModeController : skill_shot : 300
INFO : ModeController : base : 100
INFO : ModeController : =====
```

Now if you hit the switch for `target1` (or whatever switch you configured), you should see the score jump by 10,000 points:



Of course what you might have noticed is that yes, the skill shot is working, but it runs forever. How do we make it so it stops after a few seconds? With a timer!

(F) Configure a timer to end the skill shot mode

MPF has timers you can configure via your config files you can use to do all sorts of useful things. If you haven't used MPF timers before, you can read about them in the [timers: section of the configuration file reference](#). Let's set up the a timer like this: (Set this up in the skill shot's configuration file, *skill_shot.yaml*.)

```
timers:
  skill_shot:
    start_value: 5
    direction: down
    control_events:
      - event: balldevice_playfield_ball_enter
        action: start
```

Like most sections of the config file, the `timers:` section is a list of timers. Each sub-entry is the name of a timer, and then all the entries under there are configuration settings for that specific timer.

When a timer completes, it posts an event `timer_<timer_name>_complete`. This is why we configured the skill shot mode's settings with `stop_events: timer_skill_shot_complete`. In other words, this timer ending will post the `timer_skill_shot_complete` event which will cause the mode to stop.

Taking another look at the `skill_shot` timer settings, notice that we specified `balldevice_playfield_ball_enter` as a control event to start the timer. But why *balldevice_playfield_ball_enter*? Why not *mode_skill_shot_started*? In this case we don't want the timer to actually start counting down until the ball has been plunged and is in play. If we set the timer to start on `mode_skill_shot_started` then it's possible the skill shot timer could end and the skill shot would be unloaded before the player even had a chance to plunge!

Of course you could ask, "Then why start the mode with the *ball_starting* event? Why not just configure the mode to start with *balldevice_playfield_ball_enter*?" Good question, of course! The reason we start the skill shot mode on *ball_starting* is because we might want to do something with the skill shot mode before the player has plunged the ball. For example:

- Maybe we want to show a skill shot slide on the display?
- Maybe we want to flash a light indicating a shot or lane that's available for the skill shot. (And if it's a lane, maybe we want the player to be able to use the flippers to rotate it even before they plunge.)
- Maybe we want the skill shot to watch for a flipper button being held in or some other combination of pre-launch setting that would actually change how the skill shot behaves.

Of course if you don't want any of that, then yeah, you can change your skill shot mode settings so it loads on *balldevice_playfield_ball_enter*. (Just be sure to also change your timer so that it starts ticking right away instead of waiting for a start event.)

Anyway, go ahead and test the skill shot mode again. When you push start, you should see both the skill_shot and the base modes show as active in the console log. Then after 5 seconds, you should see the skill_shot mode unload, leaving just the base mode.

Here's a console log of that whole process *without* verbose logging:

```
INFO : SwitchController : <<<<< switch: start, State:1 >>>>>
INFO : SwitchController : <<<<< switch: start, State:0 >>>>>
INFO : Game : Game Starting!!
INFO : Game : Player added successfully. Total players: 1
INFO : Mode.base : Mode Starting. Priority: 100
INFO : Mode.skill_shot : Mode Starting. Priority: 300
INFO : Mode.base : Mode Started. Priority: 100
INFO : ModeController : ===== ACTIVE GAME MODES =====
INFO : ModeController : base : 100
INFO : ModeController : =====
INFO : Mode.skill_shot : Mode Started. Priority: 300
INFO : ModeController : ===== ACTIVE GAME MODES =====
INFO : ModeController : skill_shot : 300
INFO : ModeController : base : 100
INFO : ModeController : =====
INFO : SwitchController : <<<<< switch: target1, State:1 >>>>>
INFO : SwitchController : <<<<< switch: target1, State:0 >>>>>
INFO : Mode.skill_shot : Mode Stopping.
INFO : Mode.skill_shot : Mode Stopped.
INFO : ModeController : ===== ACTIVE GAME MODES =====
INFO : ModeController : base : 100
INFO : ModeController : =====
```

So far, so good... right? We're almost done, but there are a few more things we should tidy up first.

(G) End the skill shot once it's made

You might have noticed that if you keep hitting *target1*, the player will keep getting 10,000 points (at least until the 5 second timer expires). Maybe that's a cool thing that you want, but more likely you want the shot to only be scored once. The easiest way to fix this is to add *target1_active* to the skill shot mode's *stop_events*. So now when a switch with the *points_please* tag is hit while the skill shot is running, it will give the player 10,000 points and also stop the mode. (And stopping the mode will remove the scoring event that was in that mode, meaning additional hits to that target will only score 1,000 points since that's what's configured in the lower priority base mode.)

This also means that we have two *stop_events* for this mode—one which will stop the mode when the timer runs out, and a second which will stop the mode when that switch is hit.

Either one will stop the mode, and if the other event happens later it won't matter because the mode will already be stopped and won't even know about it.

At this point, your entire `your_machine/modes/skill_shot/config/skill_shot.yaml` file should look like this:

```
mode:
  start_events: ball_starting
  stop_events: timer_skill_shot_complete, target1_active
  priority: 300

scoring:
  target1_active:
    score: 10000|block

timers:
  skill_shot:
    start_value: 5
    direction: down
    control_events:
      - event: balldevice_playfield_ball_enter
        action: start
```

(H) Add some lights and display to advertise the skill shot

Really the sky's the limit with what you can do with this skill shot, and you've already learned a bunch of different things you could do in this tutorial so far.

For example, what if you wanted to flash the light at the target while the skill shot was running? How would you do that?

One way would be to create a show file which flashes that light. Put this file in the shows folder in your skill shot mode folder, e.g. `your_machine/modes/skill_shot/shows/flash_target1.yaml`. The file could look like this:

```
- tocks: 1
  lights:
    target1: ff
- tocks: 1
  lights:
    target1: 00
```

Then in your skill shot's mode config file (`skill_shot.yaml`), add a `light_player:` section and configure this show to start and stop with the mode, like this:

```
light_player:
  mode_skill_shot_started:
    - show: flash_target1
      repeat: yes
      tocks_per_sec: 4
```



```
mode_skill_shot_stopped:
  - show: flash_target1
    action: stop
```

Since this light show will run at a priority of 300 (since that's the skill shot mode's priority), it will flash the light "on top" of what any lower priority mode wants the light to do.

You could also configure a countdown timer on the display with a `slide_player:` entry in your `skill_shot.yaml` file, like this:

```
slide_player:
  timer_skill_shot_tick:
    type: text
    text: "SKILL SHOT TIME: %ticks%"
```

Note that "SKILL SHOT TIME: %ticks%" is in quotes because colons have special meaning in YAML files, and since we wanted the color after the word "TIME" we had to wrap the whole thing in quotes.

Also, we knew that running timers post an event each tick called `timer_<timer_name>_tick` with a parameter called `ticks` by searching through the log files for event names.

Now as soon as the ball goes live on the playfield, the display will show a countdown like this: (Assuming you don't hit the skill shot right away.)



Of course this is all just scratching the surface of what you can do, but hopefully you're starting to see how you can use modes and timers to do some pretty cool things—all without any coding!

How To: Create a Kickback Mode

Warning: This How To guide has not yet been updated for MPF 0.21 (the current version). If you're reading this now and you'd like to add this to your game, send me an email (brian@missionpinball.com) and I'll get it updated.

Many games have a kickback, so let's go through the process of setting one up as a mode. Typically, a kickback has a series of things that have to be done to enable it (in our case, completing a set of standup targets), an indicator light to let you know kickback is enabled, a coil to be fired very quickly, and a grace period in case the ball falls back down the outlane too quickly. The good news is that we can do it all in via config files, so here we go!

(A) Set up your kickback coil as an AutoFire Coil

The first thing we need to do is define our kickback coil as an autofire coil. Remember that autofire coils have rules written directly on to the hardware that fire a coil the instant a switch is hit rather than sending the switch event to MPF and letting MPF handle the coil firing. This is done for things like slingshots and flippers that you need to react instantaneously to a switch closure. The actual value of autofire coils when you're running on a super-fast host PC is debatable, but for the kickback we don't want any room for doubt. If that ball goes down the left outlane crazy fast, we want it saved!

To set up a new autofire coil, we need to edit our machine-wide configuration (`your_machine/config/config.yaml`) with this:

```
autofire_coils:
  kickback:
    coil: c_kickback
    switch: s_leftOutlane
    enable_events: mode_kickback_started
    disable_events: mode_kickback_stopped
```

Note that in addition to connecting the `s_leftOutlane` switch to the `c_kickback` coil, we also added enable and disable events. That's because we don't want the kickback to always be functional. You have to earn it!

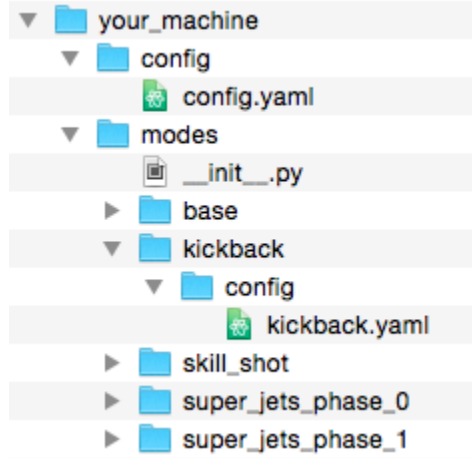
The event we're using to enable the kickback is the `mode_kickback_started` event that is posted when our kickback mode begins. Likewise, we're using the `mode_kickback_stopped` event that's posted when the mode ends to disable the autofire coil. That means we need a mode called "kickback", so let's do that next.

(B) Create the kickback game mode

First, create a folder tree just like you did for the `skill_shot` and `base` modes, except this time substitute "kickback" for the folder and `yaml` file names:

1. Create a `kickback` folder in your machine's modes folder.
2. Create a `config` folder in the `kickback` folder.
3. Create an empty file called `kickback.yaml` in the `your_machine/modes/kickback/config` folder.

You should wind up with something that looks like this:



Then, in your machine-wide configuration (`your_machine/config/config.yaml`), add kickback to the list of modes, like this:

```

modes:
  - base
  - skill_shot
  - super_jets_phase_0
  - super_jets_phase_1
  - kickback

```

Now, let's edit the mode configuration file (`your_machine/modes/kickback/config/kickback.yaml`):

```

mode:
  start_events: targets_left_bank_lit_complete
  stop_events: timer_kickback_grace_period_complete
  priority: 400

```

We gave it a priority of 400 because we already have a base mode running at priority 100, a skill_shot running at a priority of 200, and super_jets running at 300. Since we recommend giving priorities to your early modes in multiples of 100, the kickback became 400. You'll appreciate this approach later on in game development when you want to insert a mode at a priority somewhere in between two other modes you've created.

(In reality, these modes that we've created so far could have with the same priority since they are unrelated to one another. We're showing it here to illustrate that it's best to consider your mode priorities as you create them so you don't run into a major problem later when you have dozens of modes)

As you can see, we've already referenced two things that haven't been created yet: `targets_left_bank_lit_complete` and `timer_kickback_grace_period_complete`. Let's set up our targets first.

(C) Create a Target Group

This tutorial is going to make the player complete a bank of standup targets in order to light kickback. To do this, we need to define our targets in our global hardware configuration, which should already be done. If you haven't set them up as a [Target Group](#) yet, now is the time to do it.

So, in your machine-wide config, create a Target Group like this:

```
target_groups:
  left_bank:
    targets: t_leftBankBottom, t_leftBankTop, t_leftBankMiddle
    reset_events: targets_left_bank_lit_complete
    enable_events: ball_started
    disable_events: ball_ending
```

Note that Target Groups are referencing individual targets, not switches. In MPF, “targets” is the term we use to represent a switch that is paired with a light, like a rollover lane or standup target with an arrow light pointing at it. If you haven't set up your targets yet, you can read more about that in the [Targets section](#).

Inside our Target Group, we defined a group of three targets. When these three targets are all hit, MPF will post an event called *targets_left_bank_lit_complete*. We've specified that event as the reset event for this Target Group, so when you complete all three targets, the lights will go out and you'll have to start again. If you go back and look at our mode config, though, you'll also notice that we're using the same event to start the kickback mode.

The *enable_events* and *disable_events* are simply when we want to turn on/off that Target Group. In this case, *ball_started* and *ball_ending* are appropriate.

(D) Create a Timer for a kickback grace period

Now that we have the TargetGroups set up, let's go back to our mode config. The `start_events:` section is taken care of, but now we need to set up a timer. We'll use this timer to kick off a light show, give a bit of a grace period after kickback fires, and to end the mode when it expires.

```
timers:
  kickback_grace_period:
    start_value: 5
    direction: down
    control_events:
      - event: sw_kickback
        action: start
```

The timer that we've set up is called *kickback_grace_period*, and like the timer we set up in the *skill_shot* mode, it's counting down from 5 seconds. When the kickback mode is running (after the target bank has been completed), rolling over the left outlane switch will fire the

kickback. The easiest way to trigger the timer, then, is to use an event tied to that left outlane switch.

In your machine-wide config file, add a tag to your left outlane switch called “kickback”.

```
switches:
  s_leftOutlane:
    number: S15
    label: Left outlane
    tags: playfield_active, kickback
```

Remember, any time a switch is hit, it posts an event that starts with “sw_” followed by any tags it might have (one event per tag). So in our case, any time a ball rolls over *s_leftOutlane*, we get two events: *sw_playfield_active* and *sw_kickback*. It's that *sw_kickback* event that we're using to kick off our timer.

(E) Give it a shot!

We're ready to try out the kickback. Here's what your config file should look like now:

```
mode:
  start_events: targets_left_bank_lit_complete
  stop_events: timer_kickback_grace_period_complete
  priority: 400
```

```
timers:
  kickback_grace_period:
    start_value: 5
    direction: down
    control_events:
      - event: sw_kickback
        action: start
```

Remember, we don't have to tell the mode to fire a coil because the act of starting this mode enables the kickback coil as an autofire coil.

To test this mode, start a game and hit each of the targets in your Target Group, then drop a ball down the left outlane. It should pop out. If you put it back in within 5 seconds, it should pop out again, but if you wait too long you'll have to re-enable the kickback by hitting your targets again.

(F) Add some effects

That was fun, but we need some sort of indication that the kickback was lit, so let's add some display and light elements.

1) Show something on the display

First, let's add some text to our DMD to let us know that your ball has been saved (as if the fact that it's out on the playfield isn't enough proof). To do that create a SlidePlayer entry in your mode config file:

```
slide_player:
  sw_kickback:
    - type: Text
      text: "Kickback!!!"
      v_pos: center
      transition:
        type: move_in
        direction: top
```

Here, we say that when the event *sw_kickback* happens (which is when the left outlane switch is hit), the game will display a text element on the DMD that says "Kickback!!!" Simple enough.

2) Turn on a light

Next, we need to turn on the kickback light. Most games have this, and most games have a few different states for that light. The first state is to have the kickback light be on to notify the player that kickback is enabled. To do this, we need to create a show file in the "shows" subdirectory under our kickback folder. Name the file "*light_kickback.yaml*" and configure it like this:

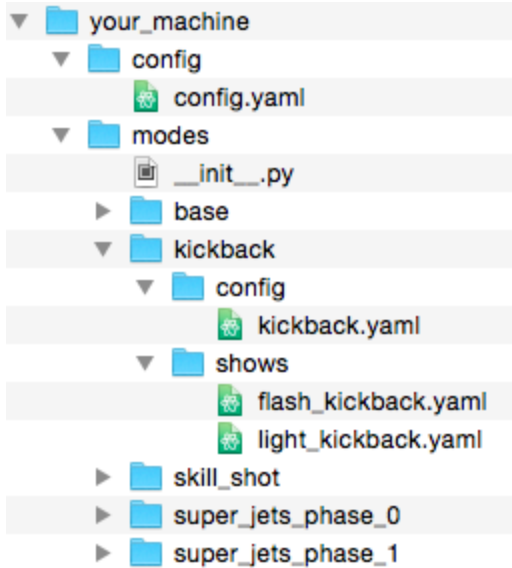
```
- tocks: 1
  lights:
    l_kickback: ff
```

That was easy. We just told it to turn on the light named "*l_kickback*". While you're in that folder, create another show file called "*flash_kickback.yaml*" and fill it with this:

```
- tocks: 1
  lights:
    l_kickback: ff
- tocks: 1
  lights:
    l_kickback: 00
```

In one tock, we're turning on the light, and in the next we're turning it off. Again, nothing too crazy here.

Your folder tree should now look like this:



Back in our mode config, we need to load those show files we created into the ShowPlayer, which we do like this:

```
show_player:
  mode_kickback_enabled:
    - show: light_kickback
      repeat: no
  mode_kickback_stopped:
    - show: light_kickback
      action: stop
    - show: flash_kickback
      action: stop
  sw_kickback:
    - show: flash_kickback
      repeat: yes
      tocks_per_sec: 16
```

Remember that the Show Player is set up to respond to events, so when the event *mode_kickback_enabled* is posted (this is the event that posts when the mode starts), the Show Player knows to play the “light_kickback” show, which simply turns on the light.

When the *mode_kickback_stopped* event is posted after the timer runs out (since the time running out is the stop event for the mode) or the ball ends, it stops running the shows, which turns off the light.

The final entry in our Show Player is for the event *sw_kickback* (man that switch is important!). When the ball rolls over the left outlane switch and that event is posted, in addition to starting the timer and firing the kickback coil, it also tells the Show Player to play the “flash_kickback” show file that we created. In this case, we want the flashing to be very rapid, so we set the `tock_per_sec` to 16, which means that it will flash on and off eight times per second.

(G) Sit back and watch your kickback!

So that's it! We've set up a kickback that is enabled when a certain goal is completed, notifies you that it's enabled by turning on a light, shows some text on the DMD, and gives you a grace period after the kickback has fired. You should have a kickback mode configuration file that looks something like this:

```
# kickback.yaml mode config file

mode:
  start_events: targets_left_bank_lit_complete
  stop_events: timer_kickback_grace_period_complete
  priority: 400

slide_player:
  sw_kickback:
    - type: Text
      text: "Kickback!!!"
      v_pos: center
      transition:
        type: move_in
        direction: top

timers:
  kickback_grace_period:
    start_value: 5
    direction: down
    control_events:
      - event: sw_kickback
        action: start

show_player:
  mode_kickback_enabled:
    - show: light_kickback
      repeat: no
  mode_kickback_stopped:
    - show: light_kickback
      action: stop
    - show: flash_kickback
      action: stop
  sw_kickback:
    - show: flash_kickback
      repeat: yes
      tocks_per_sec: 16
```

How To: Configure a 1980s-style trough

This How To guide explains how you configure a classic style trough which has a dedicated outhole (drain) coil which is separate from where the balls are stored. This was common on classic machines of the 1980s, up through System 11 and in some early WPC machines.

If you're just landing on this page randomly, you might also want to check out our [tutorial on how to configure a trough](#), as it provides some more background information about many of

the things we reference here. Also if you're using MPF with a System 11 machine, we have a full [How To guide on System 11](#).

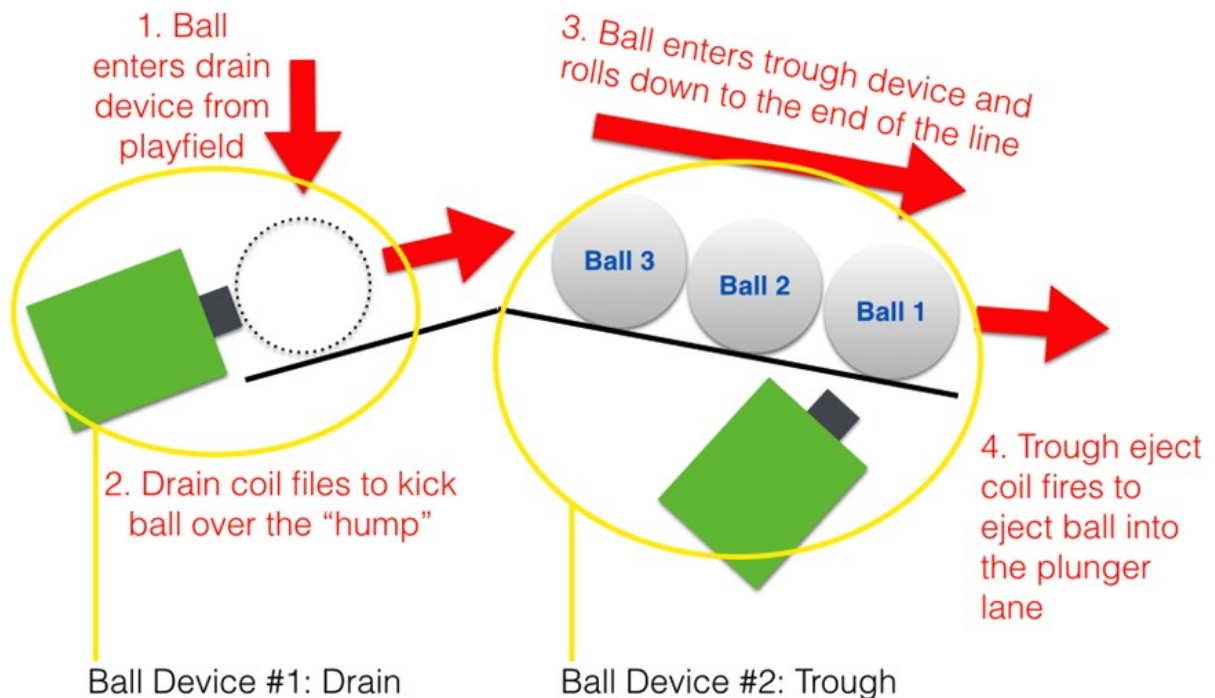
(A) Understand how classic troughs are different from modern troughs

The first step to configuring a classic trough is to understand how it's different than a modern style troughs. Rather than having a single ball device which holds all the balls and then ejects them into the shooter lane, classic machines actually have two devices that work together.

Device #1 can be called the "Drain." (Some manuals call it an "outhole", but we'll call it a drain here.) A ball enters this device when it drains from the playfield. The machine doesn't store balls in this device, rather, it immediately fires the coil to kick the ball up over a hump, with the ball (or balls) are permanently stored on the right side of the hump.

Device #2 can be called the "Trough," as it's more like a modern day trough. It has switches for each ball position and a coil to kick the ball out into the plunger lane.

Take a look at the following diagram which illustrates how the two-coil ball systems work: (This diagram shows three balls on the right-hand trough side, though many classic machines only held one or two. The configuration is the same either way.)



The coil on the left and the switch which detects a ball there will make up the ball device we'll call the "drain", and the coil and three switches on the right will make up the ball device we'll call the "trough".

(B) Add your coils and switches to your config file

In the *switches:* section of your config file, add your drain switch plus the ball switches for your trough. (I think there were either two or three switches on the trough side, but you should enter as many switches as you have.)

Then in the *coils:* section of your config file, add entries for your drain and trough coils.

These new entries in your config file should look something like this: (Note that if you're building this config for a physical machine, your *number:* entries will need to correspond to the actual hardware numbers and formats for your machine. See the [switches:](#) and [coils:](#) section of our configuration file reference for details.)

```
coils:
  drain_eject:
    number: 0
  trough_eject:
    number: 1

switches:
  drain:
    number: 0
  trough1:
    number: 1
  trough2:
    number: 2
  trough3:
    number: 3
```

(C) Configure your drain device

Next, create an entry in your *ball_devices:* section for your drain.

Add *ball_switches: drain* which means this device will read the drain switch to know whether or not this device has a ball.

Add *eject_coil: drain_eject* which is the name of the coil that will eject the ball.

Add *entrance_count_delay: 300ms* (or whatever value you want) to allow for settling time when a ball enters before MPF will process the new ball.

Add *confirm_eject_type: target* to configure MPF so that this device will confirm a ball eject via a new ball entering another ball device.

Add *eject_targets: trough* which tells MPF that this ball device ejects its balls into the device called trough.

Add *tags: drain* which tells MPF that balls entering this device mean that a ball has drained from the playfield.

Your drain configuration should look something like this:

```
drain:
  ball_switches: drain
  eject_coil: drain_eject
  entrance_count_delay: 300ms
  confirm_eject_type: target
  eject_targets: trough
  tags: drain
```

(D) Configure your trough device

The actual trough configuration in a System 11 style machine is similar to the configuration of a modern trough, and you'll configure your entrance count delays, switches, and eject coils as you would any trough. The only difference is your tags. A modern-style trough would have three tags: *drain*, *trough*, and *home*. But with System 11 style troughs, the drain device is the drain, not the trough, so your trough will only have *trough* and *home* tags. (The *trough* tag tells MPF that this device wants to hold as many balls as it can, and the *home* tag tells MPF that it's okay for balls to be contained in this device when the machine is first booted and when games start.)

Your trough configuration should look something like this:

```
trough:
  ball_switches: trough1, trough2, trough3
  eject_coil: trough_eject
  entrance_count_delay: 300ms
  confirm_eject_type: target
  eject_targets: plunger_lane
  tags: home, trough
```

(E) Configure your plunger lane device

You can configure your plunger lane (or "shooter lane" or whatever you're calling it) in a classic machine just like any other machine, as outlined [here](#).

(E) Understanding how this works

When a ball drains from the playfield, it will enter your drain ball device. Since that device is tagged with *drain*, that will trigger MPF's ball drain handler which will remove a ball from play, trigger ball save, etc.

Since the drain doesn't desire to hold any balls, it will immediately eject the ball. It will watch for a ball entering the trough device to confirm its eject.

Then at this point the balls are stored in the trough and it works just like a machine with a modern style trough.

How To: Create an Attract Mode DMD slide show

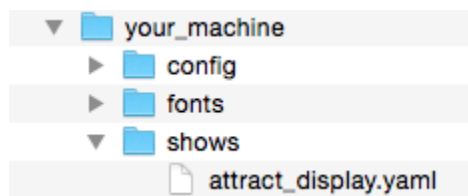
This tutorial will show you how to create a display show which plays during your machine's attract mode, including several types of [display elements](#), [slides](#), and [transitions](#).

Here's a video of the finished product.

<https://www.youtube.com/watch?v=4wo-X8hSwMc>

(A) Create your show YAML file

The easiest way to create a series of different slides and transitions is to create a [show](#). For this example, create a new file called `attract_display.yaml` and put it in the `shows` folder in your machine's root folder. This folder should be at the same level as your `config` file, like this:



(B) Add the entries to your show

You can refer to the "[Creating Shows](#)" page for details on how to actually create a show. In this case we created a show that was strictly for the display. (We'll do lighting effects in their own show which will run at the same time.)

So each step of the show only has a `display:` section, and then in there you'll see one or more [display elements](#) (text, image, animation, etc.) along with the specific settings for each. The show file for the show that's playing in the video above is here:

```
- tocks: 3
  display:
  - type: Text
    font: small
    v_pos: center
    h_pos: center
    text: DRAWING SHAPES
    layer: 1
```

```

- type: Shape
  shape: box
  x: 0
  y: 0
  width: 128
  height: 32
  thickness: 3
- type: Shape
  shape: box
  x: 10
  y: 10
  width: 10
  height: 15
  shade: 9
- type: Shape
  shape: box
  x: 100
  y: 5
  width: 20
  height: 30
  thickness: 0
  shade: 3

- tocks: 3
  display:
    - type: Text
      text: MISSION PINBALL
      transition:
        type: move_out
        duration: 1s
        direction: left
    - type: Shape
      shape: box
      width: 128
      height: 32
      shade: 14
      x: 0
      y: 0

- tocks: 3
  display:
    - type: Image
      image: test
      transition:
        type: move_in
        direction: left
      layer: 2
    - type: Text
      text: TEXT UNDER IMAGE
      layer: 1
      shade: 8

- tocks: 3
  display:
    - type: Text
      text: PRESS START
    - type: Text
      text: FREE PLAY
      v_pos: bottom

```

```

    font: small
    transition:
      type: move_in
      duration: 1s
      direction: right

- tocks: 3
  display:
    - type: Animation
      animation: rolling_ball
      fps: 60
      drop_frames: no
      layer: 1
    - type: Text
      text: TEXT OVER ANIMATION
      layer: 2
      shade: 0
      transition:
        type: move_out
        direction: top

- tocks: 3
  display:
    - type: Text
      text: JUDGE DREDD
      transition:
        type: move_in
        duration: 1s
        direction: top
    - type: Shape
      shape: box
      width: 128
      height: 32
      shade: 9

- tocks: 3
  display:
    - type: Image
      image: p_roc
      layer: 1
      transition:
        type: move_in
        direction: left
    - type: Text
      text: UNDER IMAGE
      layer: 0
      shade: 8

- tocks: 3
  display:
    - type: Text
      text: CENTERED SIZE 10
    - type: Text
      font: small
      v_pos: top
      h_pos: left
      text: TOP LEFT SIZE 5
    - type: Text
      text: BOTTOM CENTERED SIZE 7

```

```

v_pos: bottom
h_pos: center
font: medium
transition:
  type: move_out
  direction: bottom

- tocks: 3
  display:
  - type: Text
    font: small
    v_pos: center
    h_pos: center
    text: DRAWING SHAPES
    layer: 1
  - type: Shape
    shape: box
    x: 0
    y: 0
    width: 128
    height: 32
    thickness: 3
  - type: Shape
    shape: box
    x: 10
    y: 10
    width: 10
    height: 15
    shade: 9
  - type: Shape
    shape: box
    x: 100
    y: 5
    width: 20
    height: 30
    thickness: 0
    shade: 3

```

(C) Configure the show to start and stop by itself

Once we create our show, it just sits there, existing. We have to tell it when to play and when to stop. Fortunately we can do that via our machine config file in the [ShowPlayer: section](#) which we entered like this: (If you already have `machineflow_Attract_start` and `machineflow_Attract_stop` entries because you created an [attract mode light show](#), that's fine. Just add your display show to the list under the existing event entries.

```

show_player:
  attract_start:
    - show: attract_display
      repeat: yes
      tocks_per_sec: 1
  attract_stop:
    - show: attract_display
      action: stop

```

MPF automatically loads show files from the `shows` folder and gives them the name that matches their file name. So just having our show file in the right place means it got loaded. Then we just have to make two entries into our `show_player:` section—one which starts the show when the `attract_start` event is posted, and another which stops the show when `attract_stop` is posted.

You'll notice that we're running this show at 1 `tocks_per_sec`, and that all of our steps in the YAML file are 3 tocks each. So every step is the 3 seconds. We could have just as easily made each step 1 tock and then ran the show at `.33 tocks_per_sec`, but it seems like 1 `tock_per_sec` gives us good flexibility to change the timing of individual steps in our show in the future.

So that's it! Save your config file and run your game, and you should see your show start to play once the attract mode starts up.

How To: Create a "super jets" countdown mode

Warning: This How To guide has not yet been updated for MPF 0.21 (the current version). If you're reading this now and you'd like to add this to your game, send me an email (brian@missionpinball.com) and I'll get it updated.

While we're working on game modes, let's create one more before we move on to other things. In this step, we're going to create a "super jets" mode. This is something that's pretty common, where each hit to a pop bumper is worth a small amount of points initially, but after a lot of hits (usually 75 or so), you get "super jets" which makes each pop bumper hit worth a lot more.

So in our tutorial, let's make our pop bumpers worth 500 points each for the first 15 hits (picking a small number to make testing easier, but you can change it to 75 once everything is working), and then after that, "super jets" will be enabled and each hit will be worth 5,000 points. We also want the count (and the super jet status) to persist from ball-to-ball. (In other words, if our player gets 5 hits their first ball, they'll enter their second ball only needing 10 more hits instead of the full 15.)

If you don't have pop bumpers, you can still follow along with this tutorial. Maybe configure slingshots instead? Or just any standup targets?

(A) Create your pop bumpers

A pop bumper is typically made up of one switch and one coil. As you can imagine, you add your switches and coils to the `switches:` and `coils:` section of your machine config file just like any other switch and coil. For example (just showing the pop bumper-related entries):


```

switches:
  pop_left:
    switch: pop_left
    number: 31
    tags: playfield_active, pop
  pop_right:
    switch: pop_right
    number: 32
    tags: playfield_active, pop
  pop_upper:
    switch: pop_upper
    number: 33
    tags: playfield_active, pop
coils:
  pop_left:
    coil: pop_left
    number: 11
    pulse_ms: 20
  pop_right:
    coil: pop_right
    number: 12
    pulse_ms: 20
  pop_upper:
    coil: pop_upper
    number: 13
    pulse_ms: 20

```

Of course remember that your coil and switch numbers will be different (and might have different formatting) depending on your hardware platform. If everything you're doing is virtual and all your numbers are fake, then just pick the next three numbers for each.

Also if you're virtual, you might want to pick three keyboard keys to map to your pop bumpers so you can test out what we're doing here.

Once you have your switches and coils in your machine configuration, you need to create your pop bumper devices which are what cause the coil to automatically pulse when the switch is activated. In MPF, these are called *Autofire* devices ([more info](#)), and you configure them in the [autofire_coils: section](#) of your machine config file:

```

autofire_coils:
  pop_left:
    switch: pop_left
    coil: pop_left
  pop_right:
    switch: pop_right
    coil: pop_right
  pop_upper:
    switch: pop_upper
    coil: pop_upper

```

Note that autofire coil devices automatically enable and disable themselves when balls start and stop.

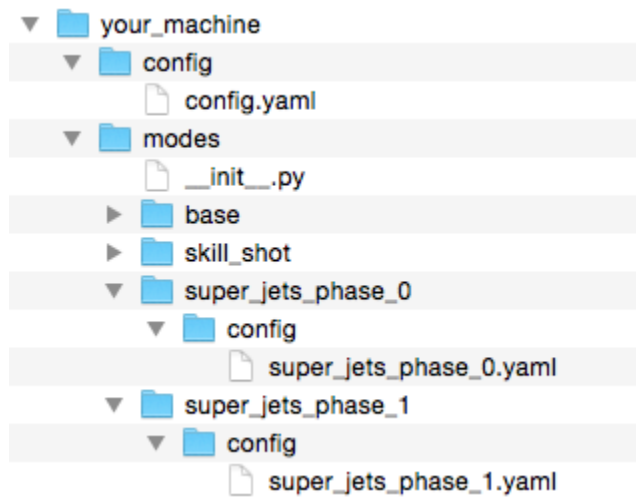
(B) Create two super jets modes

Now that your pop bumpers are all set, let's create a pair of game modes that we'll use to create the super jets functionality. Why two? We'll use one to track the progress towards the super jets, and another which will take over once the super jets are actually active.

You might be wondering if two modes is really necessary? Certainly it's possible to build this with one mode, but really it doesn't matter. Modes don't add any overhead, and they don't slow things down. You can have dozens or hundreds running at once. So yeah, maybe we could put the progress tracker towards super jets in the base game mode, but that feels like the super jets would be polluting that mode. And yeah, maybe we could figure out how to create one mode which tracks the super jets progress and then stops tracking it once super jets are active, but again, meh. Why struggle with that? Two modes makes it easy and it doesn't hurt anything.

Anyway, the two modes we're going to create are called *super_jets_phase_0* and *super_jets_phase_1*. Phase 0 will be the mode that will be enabled by default. It's where the jets will be worth 500 points each, and it's where we will be tracking the progress of our 15 hits. Then once we get that, the phase 0 mode will stop and the phase 1 mode will start. The phase one mode will assign 5,000 points per hit. (And if we wanted to, we could create a phase 2 mode which would be even more points per hit that the phase 1 mode could track... and so on....)

The first step is to add the two folders for the two modes to the modes folder in your game and then to create the empty mode config files as you did when you made your previous modes:



In the super jets phase 0 mode configuration, (*super_jets_phase_0.yaml*), let's set it up like this:

```
mode:
  start_events: ball_starting
  stop_events: super_jets_phase_1_start
  priority: 300
```

Then for your super jets phase 1 mode (`super_jets_phase_1.yaml`), let's configure it like this:

```
mode:
  start_events: super_jets_phase_1_start
  priority: 300
```

(C) Configure a counter to track progress towards super jets

Another MPF component we haven't covered in the tutorial yet is something called *Logic Blocks*. Logic Blocks are like logical "glue" you can add to your config files that can cause certain events to happen based on other events happening. There are three types of logic blocks: sequence, counter, and accrual. (More information is available in the [Logic Blocks section of our Game Logic documentation](#), as well as in the [LogicBlocks section](#) of the configuration file reference.)

In our super jets mode, we'll be using a logic block called *counter* which watches for how many times a certain event happens, and once a threshold is hit, it posts another event. So in our case, we're going to configure the counter to watch for the events posted by the pop bumpers being hit, and after 15 of them, we're going to post an event that starts the super jets.

We'll configure our counter logic block like this (in our `super_jets_phase_0.yaml` mode config file):

```
logic_blocks:
  counters:
    super_jets:
      count_events: sw_pop
      starting_count: 15
      count_complete_value: 0
      direction: down
      events_when_complete: super_jets_phase_1_start
```

(D) Add the two modes to the machine-wide modes list

Don't forget to add these two new modes to the list of modes this game uses in your machine-wide config.yaml file. The Modes: section of that file should now look like this:

```
modes:
  - base
  - skill_shot
```

```
- super_jets_phase_0
- super_jets_phase_1
```

(E) Configure scoring for the two super jet modes

Remember we want a pop bumper hit to be worth 500 points ordinarily (in "phase 0" mode) and 5,000 points in super jut ("phase 1") mode. So let's create those two scoring entries now.

First, in your `super_jets_phase_0.yaml` file, add the following section:

```
scoring:
  sw_pop:
    Score: 500
```

Then in you `super_jets_phase_1.yaml` file:

```
scoring:
  sw_pop:
    Score: 5000
```

(F) Test it out

Save your two config files and then run your game. Once you press start, you should see your score increase by 500 points for the first 15 hits of the pop bumpers. Then on the 16th hit, you should see that you get 5000 points (and 5000 for every hit after that).

You'll also notice in the log file that the `super_jets_phase_0` mode stops and the `super_jets_phase_1` mode starts on the 16th hit.

(G) More to come...

We have a bit more to finish in this step, including:

- Displaying a count down to super jets on the display
- Getting the super jets mode to automatically be active once it's be achieved when a new ball starts

How To: Add TrueType Fonts

The Mission Pinball Framework uses TrueType fonts for text on the DMD and on screen window displays. Picking a good font for your on screen window is pretty easy since the window is high resolution and it has 24-bit color. But when it comes to the DMD, that can be more difficult.

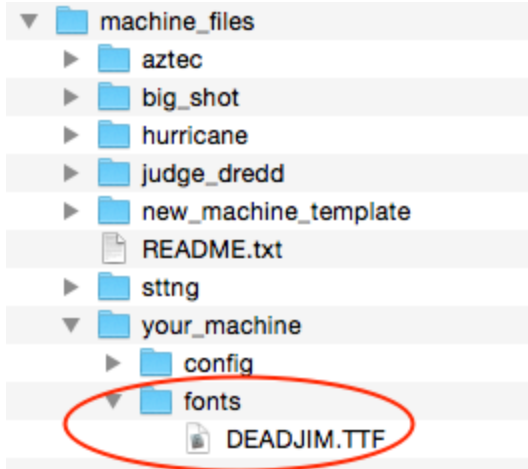
So in this tutorial we're going to show you how to adjust your TrueType fonts so they look good and they're easy to use in MPF. We'll use a font called [He's Dead Jim](#), which lives in a file called `DEADJIM.TTF`. (If you want to follow along, pick a different font because `DEADJIM.TTF` is already included and configured in the MPF package. In other words, we already did this work for this font, and it's ready to go!) Here's an example of what this font looks like:



(A) Copy the font file into your machine's fonts folder

In order to use a TrueType font with MPF, you have to copy the font file to your pinball machine's `/fonts` folder in MPF. We briefly considered allowing MPF to use fonts that were already installed on the host computer, but then we'd run into issues where things would be messed up if you ran your game from a computer that didn't have the same fonts installed. So we decided that each machine would keep its own fonts in its own machine folder in MPF, meaning those machine folders would be 100% self-contained and portable.

So go into your your machine's folder and create a new subfolder called `fonts`, and copy your font file into it:



(B) Add this font to your machine config file

Next you have to tell MPF that you want to use this font by picking a name for it and entering the file location into your machine configuration file. (The reason MPF doesn't just use the actual names of the fonts is because often times you'll have the same .ttf font files used for different sizes, and vice-versa.)

Create a [Fonts: section](#) in your config file, then on the next line, indent 4 spaces (must be spaces, not a tab) and type the name you'd like to refer to this font as in MPF. We'll call it "space_title." Then on the next line, indent 4 more spaces (so 8 total), and add a file setting for the file that you just copied. (MPF will automatically look in your machine's `fonts` folder for files entered here.)

The `Fonts:` section of your config file should now look like this:

```
Fonts:
  space_title:
    file: DEADJIM.TTF
```

Now you can test out your font by adding it into an [Attract mode display show](#) or into a [SlidePlayer entry](#). You'd set it up something like this:

```
- type: Text
  text: MISSION
  font: space_title
  size: 24
```

Now when you run your game, you'll see your new Text element come up on the display, and....



Gross. This does not look good. It's all jagged and weird and kind of small and just pretty much ugly. So now what?

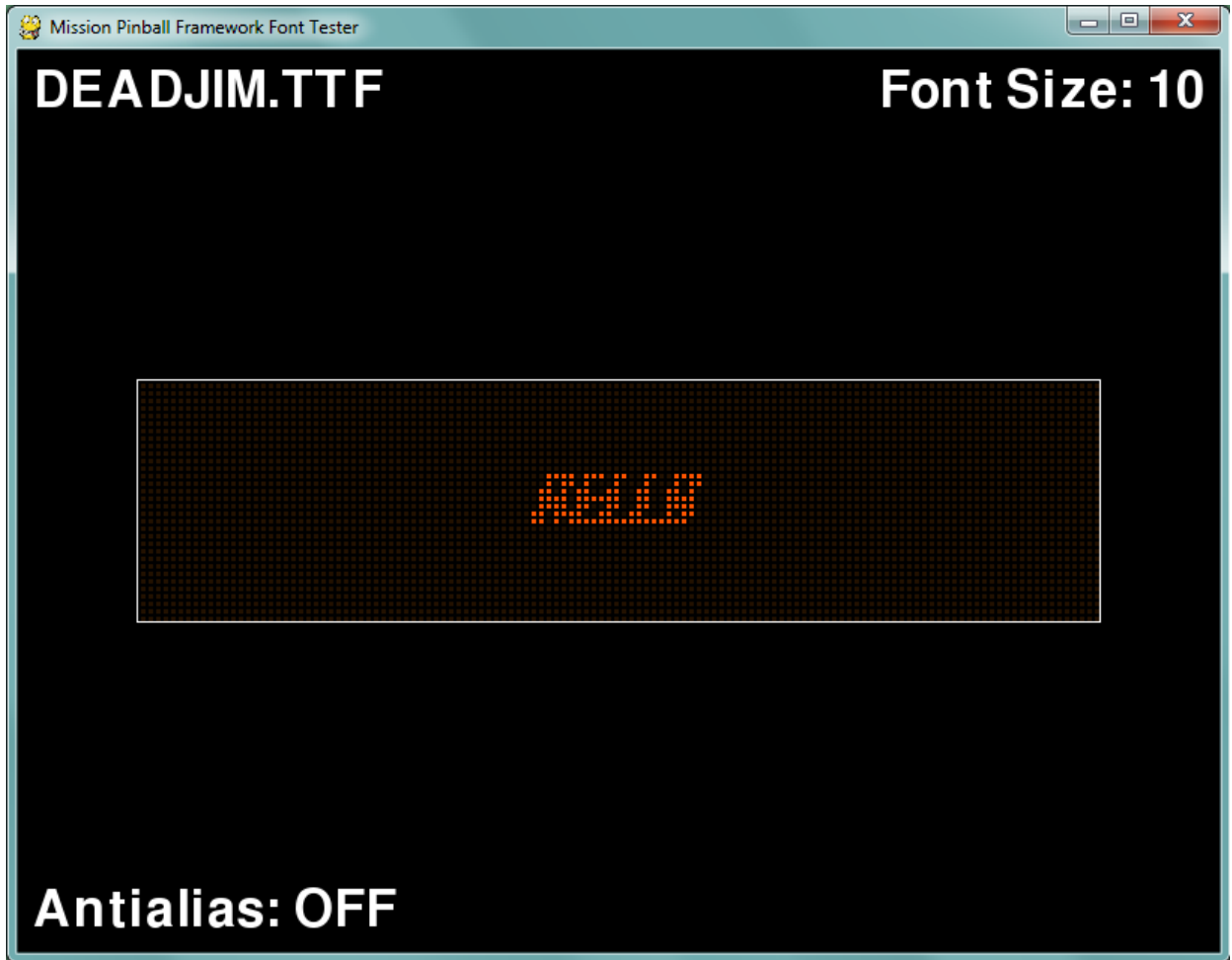
(C) Run MPF's `font_tester.py` tool

The next step to make this font look nice is to play with it with different settings to see how it will look on the DMD. MPF includes a graphical font testing tool to help you do this (and to help you pick which fonts will work best in your machine). The tool's called `font_tester.py` and it's in the `/tools` folder in your MPF package. (Full instructions for the Font Tester tool, as well as a video walk through, is available in the [Font Tester documentation](#).)

Launch the Font Tester from the command line, along with a command line argument which is either (a) a path to a folder which contains `.ttf` font files, or (b) the full path to a specific `.ttf` font file. We can launch the Font Tester with our Dead Jim font like this:

```
python font_tester.py c:\Windows\Fonts\DEADJIM.TTF
```

You should see a window pop up with looks something like this: It has loaded the DEADJIM.TTF font at a default size of 10 point with default demonstration text which says "HELLO":



Next hit the up arrow a bunch of times to increase the font size up to 24, then hit the space bar a few times to clear out the word "HELLO," then type "MISSION." Now you should see this (which is what how your machine displayed it back in Step (B)).



By the way, you can use the left and right arrow keys to step back-and-forth through different fonts in the folder, so if you accidentally switch off of the font you were working with, use the arrows to find it again.

(D) Figure out what font size you like

We rendered the sample text back in Step (B) at a 24 point size. But now that we have this font loaded in the Font Tester with a size of 24, it looks like it's rendering with characters that are 18 pixels tall on the DMD. Is this right? Why's it 18 and not 24?

This happens because TrueType fonts are resolution independent, and there's no rhyme or reason from font-to-font about sizing. Font A might be 24 pixels tall at point size 24, Font B might be 20 pixels tall, and Font C might be 30 pixels tall. It just depends on the font.

In our case we want this font to be a nice and big 24 pixels tall, so hit the up arrow a bunch of times until the font is rendering at the size you want. In this example, it looks good to us when it's at size 32. (This is according to the "Font Size:" text label in the upper right corner of the Font Tester window.)



(E) Add the size to your config file

So now that we have this font showing at a size we like, we have to add the font's "native" size to its configuration. Doing that is simple:

```
Fonts:
  space_title:
    file: DEADJIM.TTF
    size: 32
```

Now when we ask for the "space title" font without specifying a size, we'll get this DEADJIM.TTF font rendered at 32 point. (You can still specify a size when calling it to override the default.)

(F) Decide whether you'll use antialiasing

Of course even though we have the size dialed in, it still doesn't look too great. One option we can try is enabling antialiasing. Antialiasing smooths out all the sharp edges on the font, and it makes a noticeable difference on a low-resolution display such as a DMD. Some people like it, and some hate it. So really whether you use it is up to you.

We have found that some fonts really require it, like the one here in this example, and others don't need it—especially "bitmap" or "pixel" fonts that were designed for low resolution use. Personally we feel that it's probably best to be consistent throughout your machine—either all or none should be antialiased—but again, it's just a matter of taste which is up to you.

Anyway you can try antialiasing in the Font Tester by hitting `CTRL+A`. You should see the text in the lower left corner change to tell you that antialiasing is enabled, and you should see the font smooth out. You can hit `CTRL+A` again and again to turn antialiasing on and off.

In this case it looks like enabling antialiasing is actually pretty good for this font:



We can enable antialiasing in this font's configuration so it will be used by default. (Individual calls to use the font and always disable it if they want.)

```
Fonts:
  space_title:
    file: DEADJIM.TTF
    size: 32
    antialias: yes
```

(G) Trim the extra space off the top and bottom

The final step to fine-tuning this font is to trim the extra pixels off the top and bottom. Of course if you're looking at an orange font on a black screen, you have no idea which dark pixels are part of the font and which are part of the background. So in the Font Tester tool, you can hit `CTRL+B` to make the font's bounding box light up bright green.

When we did that for our Dead Jim font at 32 point, it was immediately clear that they are extra rows of pixels along the top, and at least four extra rows of pixels across the bottom.

At this point you might be wondering why this matters. Isn't this a little nit-picky?

The problem is that all fonts are different. When Pygame (the Python add-in MPF uses for graphics and fonts) renders a given font at a given size, it just produces a rectangle. For example if MPF says, "I want the text 'MISSION' to be rendered with the font 'DEADJIM.TTF' at

'32' point," Pygame's font engine says, "Okay here's a rectangle that's 100 pixels wide and 30 pixels tall."

The problem is we don't actually know where the font is inside that rectangle. Some fonts touch the top and bottom, other fonts are small and centered, and others are small and offset. Again it just depends on the font.

The reason this a problem is that all of the placement MPF does for that rendered text is actually based on the placement of the rectangle. So if you want that text to be vertically centered on the screen, MPF can center the rectangle that contains the font. But if the font is not centered in the rectangle then it won't look centered.

In some cases this doesn't matter, but in others, man, some fonts are crazy and have almost as much extra room on the top or bottom as they are tall!

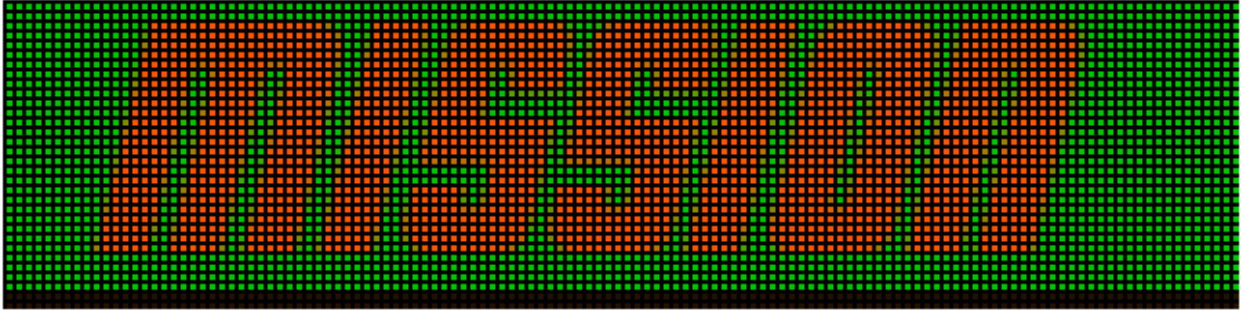
So that's what this `CTRL+B` green box is. It's the actual rectangle that this font renders into at that given size. And you're using it to see how many rows of pixels you have to crop off the top and bottom of this font. (These are size specific, so a font that requires 2 rows to be cropped at 32 point might only require 1 row to be cropped at 28 point.)

By looking at the top of this green bounding box, we can see that we need to crop 2 rows of pixels from the top.



Now for the bottom, notice that the green pixels extend all the way to the edge of the DMD. It's possible for the bounding box to be larger than the on screen DMD area, so we actually don't know how far down they go at this point. You can use the `SHIFT` key plus the up and down arrow keys to shift the font rectangle up and down, so hit `SHIFT+up` a few times to shift the entire image up until you can see where the edge of the bounding box is.

Ahh, there it is. So looking closely it looks like there are 4 rows of green pixels under the bottom of the font, so we want to crop 4 pixels from the bottom of this font when it's rendered at size 32.



If you add the crop values to your config entry, your font config should now look like this:

```
Fonts:
  space_title:
    file: DEADJIM.TTF
    size: 32
    antialias: yes
    crop_top: 2
    crop_bottom: 4
```

At this point you might be wondering why you have to go through the extra step of entering cropping values. Isn't this something that MPF could figure out on its own?

The problem with that is that different text strings have different letter heights. So any kind of "automatic" trimming of white space would change depending on whether the text had characters that stuck up or down, and then you'd end up with positioning that would change depending on what text was showing. So that's why we just go through the process of setting up the fonts ahead of time. At least this is something you only have to do once.

(H) Think about other sizes

While you have the font open in the Font Tester, you might want to think about whether you'll want to use that font at any other sizes. For example we'll create a "space_medium" version of this font which we'll add to our config, bringing our final entry to this:

```
Fonts:
  space_title:
    file: DEADJIM.TTF
    size: 32
    antialias: yes
    crop_top: 2
    crop_bottom: 4
  space_medium:
    file: DEADJIM.TTF
    size: 25
    antialias: yes
    crop_top: 3
    crop_bottom: 3
```

Note that rendering that font a 25 point actually increases the number of rows to crop on top?!? Eh, who knows why. Whatever. (That's why we made this tool!)

One final note, the Font Tester tool has a snapshot function which you can use with `CTRL+S`. When you do that it will save an image (it will try `.png` but will fall back to `.bmp` on some systems) of the window contents into a folder called `font_snapshots`. This is nice because you can spend an hour going through all the fonts on your system and playing with different sizes and settings, then if you take a snapshot of ones that you like you'll end up with a folder full of things you can use as a starting point for your config file.

How To: Configure a Color DMD

So you want a color DMD? No problem. MPF can do that. This how to guide explains the process.



(A) Understand how Color DMDs work

Color DMDs are all the rage now. You're probably heard of the company called [ColorDMD Displays LLC](#) who sells add-on products.

If you look at physical pinball DMDs, you can only buy them in one color. Sure, thanks to today's LED technology, that color doesn't have to be orange, but it's still just one color.

Maybe in the future we'll have RGB LED displays which have a tiny RGB LED for each pixel, but we don't have those yet. (At least not in the sizes we need for a traditional pinball DMD.) So the way these color DMDs work is they replace the physical DMD with an LCD monitor, then on the LCD monitor they show high res images of round dots in any color which look like low res full-color DMD images.

If you want to create a color DMD-style display in MPF, it's pretty easy.

(B) Create your "native" DMD

The first step to creating a color DMD in MPF is to create a [DMD: entry](#) in your machine configuration file. The only difference between configuring a color DMD versus a standard DMD are (1) you have to change `physical:` from `yes` to `no`, and you have to create a setting called `type:` which you set to `color`.

Here's an example:

```
dmd:
  physical: no
  type: color
  width: 128
  height: 32
  fps: auto
```

The `physical: no` setting is needed because when you have a color DMD, you don't actually have a physical pinball DMD connected up to the 14-pin DMD header on your hardware controller. The `type: color` tells MPF that the DMD display you're creating should use 24-bit full color pixels instead of the limited 16-shade single color pixels.

By the way, since you don't have a physical DMD when using color, your color DMD is not limited by a physical resolution. In other words it doesn't have to be 128x32. It can be 192x64, or 256x128, or 512x128, or... you get the idea!

To be clear, the `width:` and `height:` settings in the `dmd:` section of your config file only represent the number of virtual dots in your DMD. The actual resolution you display this at is set up elsewhere.

(C) Create the on screen element for that native DMD

You can refer to the [window: section](#) of the configuration file reference for an example of how to create an on screen window (if you don't already have one) and for an example of how to add your DMD to it. Your settings will probably end up looking something like this:

```
onscreen_dmd:
  type: VirtualDMD
  width: 512
  height: 128
  pixel_color: ff5500
  dark_color: 220000
  h_pos: center
  v_pos: center
  priority: 1
  pixel_spacing: 1
```

(Note the `pixel_color:` setting doesn't have an effect when you have a color DMD.)



When you're ready to physically mount the LCD into your backbox, you can delete all the other display elements except for the `VirtualDMD`, set the window to run in full screen mode, set your `width:` and `height:` settings to they exactly match the opening in your speaker panel, and then use [x: and y: settings to fine tune the positioning](#) of the virtual DMD with pixel-level precision.



(D) Update your display elements so they're color

If you just change your DMD to be color and do nothing else and then run your game, you'll see that all your pixels have turned from orange (or whatever color they were before) to white. So what gives?

This is because the actual display elements are not in color. So your display is showing color, but the content it's showing is mono. (This reminds us of that old [Calvin & Hobbes comic about the world turning into color.](#))

The exact way you update your display elements depends on what type of elements you have. For [Text](#) and [Shape](#) elements, you simply add a `color:` setting followed by a six-digit

hex color code. (So pretty much everywhere you previously specified a `shade:` value, you can now specify a `color:` value instead. In fact you can actually keep both of the entries there, and MPF will use the `shade:` value for traditional DMDs and the `color:` value for color DMDs and LCDs.)

For images in and animations, if you're importing full color files (like `.png`, `.jpg`, `.gif`, `.bmp`, etc.), all you have to do is remove the [target: dmd setting](#). So if MPF sees `target: dmd`, it will convert your color image to 16-shade (or whatever your DMD is set for) mono, and if it doesn't see it, it will keep the image as a full color image.

For DMD files (with the `.dmd` file extension) that are already in their 16-shade format, obviously converting them to color isn't going to do any good. They'll just continue to show up as white, (or whatever `color:` value you set them to be), and if you want them to be in proper full color then you should add the colors and then save them as a full color `.png` or something instead of a `.dmd`.

Let's look at an example real quick of an original non-color DMD, a color DMD with no settings changes, and then a color DMD with changes made to the elements.

Note these snippets of config files are not complete, rather, they just show the relevant areas to the color settings and the elements on this slide.

(1) Original with no color

Here are the DMD and Images sections of the configuration file. Notice that the DMD is configured like a traditional DMD, and the `scary_guy` image has its `target` set to `dmd` which means it will be converted from a full color PNG to a 16-shade DMD image.

```
dmd:
  physical: no
  width: 128
  height: 32
  shades: 16
images:
  scary_guy:
    file: scary_guy.png
    target: dmd
```

Here's the step from the attract mode show which builds this slide. Notice there's a text element, a shape element (a rectangle for the border), and an image element.

```
- tocks: 3
  display:
    - type: Text
      text: JUDGE DREDD
      priority: 1
    - type: Shape
      shape: rect
```



```
width: 128
height: 32
shade: 9
priority: 1
- type: Image
image: death
```

Here's the result as seen in the on screen virtual DMD window with the above settings:



(2) Switch the DMD type to "color"

Now let's simply switch the DMD type to "color" by adding a `type: color` entry.

```
dmd:
  physical: no
  type: color
  width: 128
  height: 32
  shades: 16
images:
  scary_guy:
    file: scary_guy.png
    target: dmd
```

If that's the only change we made, our slide now looks like this.



Notice that both the text and the border rectangle are white, and the image doesn't show up at all.

(3) Properly setting the color settings

To "fix" this for a proper color display, first we need to go into the config file and remove the `target: dmd` setting for our image. Doing so will mean MPF will not convert that image to the DMD 16-shade color format (which is what we want now since we have a full color display).

```
dmd:
  physical: no
  type: color
  width: 128
  height: 32
  shades: 16
images:
  scary_guy:
    file: scary_guy.png
```

Next if we go into the show file, we can add some colors to our text and rectangle elements. Right now they are both white since we don't have any colors defined for either of them, so they're using the default color (which happens to be white).

So let's add `color:` entries to the Text and Shape elements. (No color is specified for the image element since images get their colors from their pixels.)

```
- tocks: 3
  display:
    - type: Text
      text: JUDGE DREDD
      priority: 1
      color: ff0000
    - type: Shape
      shape: rect
      width: 128
      height: 32
      shade: 9
      priority: 1
      color: 0000ff
    - type: Image
      image: death
```

Notice that the rectangle shape element already had a `shade:` specified which is the intensity it was set to on the traditional DMD. You can either keep that setting or remove it. If you keep it then it will be used if you ever run this show on a traditional DMD again.

Now if you rerun MPF, you'll see the properly colored slide in your virtual DMD:



How To: What if your plunger lane has no switch?

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 in older single-ball machines, including many EM and early solid state machines.) If you're just landing on this page randomly, you might also want to check out our [tutorial on how to configure a trough](#), as it provides some more background information about many of the things we reference here.

Here's an example of what we're talking about:

Modern Plunger Lane



Switch which is active when a ball is resting in the lane.

Old Plunger Lane



This switch doesn't count. It's higher up and used in this machine to tell the machine a ball has entered the plunger lane from the playfield via a gate in the outlane. But it is not activated when a ball comes from the trough, so we ignore it when configuring the plunger.

No switch. Machine does not know when a ball is here.

In modern machines, we configure the plunger lane as a [ball device](#). This is possible since there's a switch in the plunger lane, so when that switch is active then the machine knows

there's a ball "in" the plunger ball device, and when that switch is inactive then the machine knows the plunger ball device does not contain a ball. Simple.

But what happens if your plunger lane doesn't have a switch which is activated when the ball is sitting at the plunger? How does it know there's a ball there?

The solution is simple: In machines like this, the plunger lane is *not* 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.

So if you have this type of plunger lane, then you would *not* have a "plunger" ball device configured in your *ball_devices:* section.

There's one other change you have to make to get this all to work. In modern machine where your plunger is set up as a ball device, that's the device that has the *ball_add_live* tag. (This is the tag that tells MPF that it should add a live ball into play from this device.) So in this situation since your plunger lane isn't actually a ball device, you need to add the *ball_add_live* tag to your trough, like this:

```
ball_devices:
  trough:
    tags: drain, home, trough, ball_add_live
    ball_switches: drain
    eject_coil: trough_eject
    entrance_delay_count: 0.3s
    exit_delay_count: 0.3s
```

Then when MPF needs to add a live ball into play, it will eject a ball from the trough and you're all set!

If you have a [classic 1980s-style trough](#) with separate drain and trough devices, you'd simply add the *ball_add_live* tag to the second device in the chain, like this:

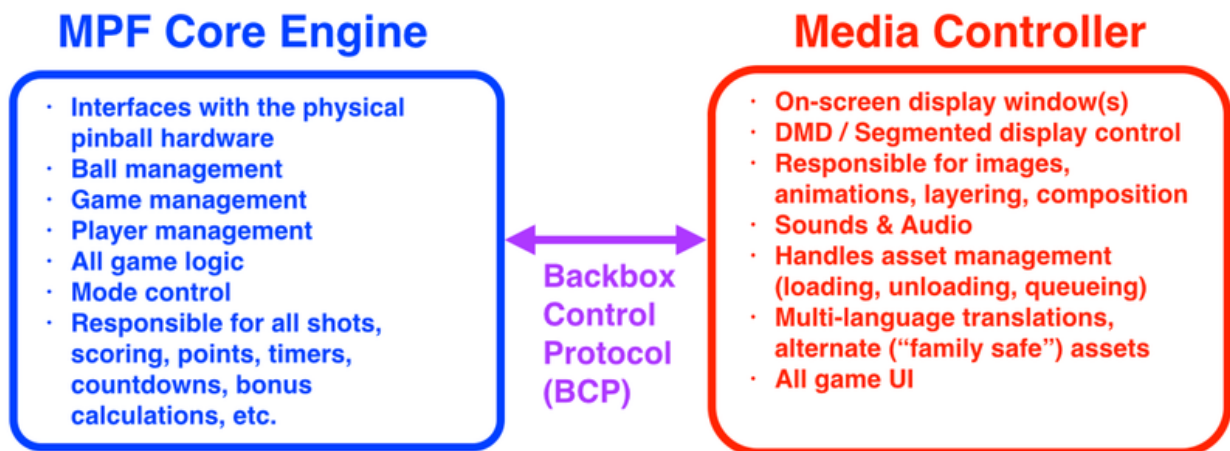
```
ball_devices:
  drain:
    ball_switches: drain
    eject_switch: drain
    eject_coil: drain_eject
    entrance_count_delay: 300ms
    confirm_eject_type: target
    eject_targets: trough
    tags: drain
  trough:
    ball_switches: trough1, trough2, trough3
    eject_switch: trough1
    eject_coil: trough_eject
    entrance_count_delay: 300ms
    confirm_eject_type: target
    eject_targets: plunger_lane
    tags: home, trough, ball_add_live
```

6. MPF Architecture Concepts

This section of the documentation explains all the nitty-gritty details of how MPF works. You don't necessarily have to read this to get started with MPF, as you can get a running pinball machine just by following the tutorial and various how to guides. But as some point you'll probably want to dig into the architecture of the various components of MPF, so that's what we'll do here.

The MPF Core Engine

As we've mentioned elsewhere in this documentation, the MPF core engine is a separate running process on your computer from the code that controls your display and sounds. When you run MPF, you'll typically have two processes running—the core game engine and a media controller, and the two processes talk to each other via a protocol we created called the Backbox Display Protocol (BCP).



The MPF core engine is written in Python 2.7. It lives in the `/mpf` folder in your MPF download package. (If your MPF download package is in a folder on your computer called `mpf`, then the MPF game engine is in a folder called `/mpf/mpf`.)

The MPF core engine is the process responsible for running the pinball machine and communicating with its hardware, including:

- All communication with physical hardware.
- Managing, tracking, and controlling all devices, including switches, coils, lights, LEDs, motors, steppers, servos, and flashers.
- All game logic, including game starting and stopping game modes, rules, players, scores, etc.

You run the MPF core engine from the command prompt like this: `python mpf.py your_machine_folder`.

Platform Interfaces

The Mission Pinball Framework is "platform agnostic," meaning it can be used to control pinball machines via several different hardware pinball controllers, including:

- FAST Pinball Nano Controller
- FAST Pinball Core Controller
- FAST Pinball WPC Controller
- Multimorphic P-ROC
- Multimorphic P3-ROC
- Mark Sunnuck's (Snux) System 11 interface board

MPF can also directly control other types of hardware devices, including:

- Fadecandy (an option to control RGB LEDs)
- OpenPixel Control hardware (for RGB LEDs and DMX lighting)
- SmartMatrix controllers (for RGB LED matrix-based displays)

We'll continue to add support for future platforms as the needs arise. (Most likely the open source Open Pinball Project (OPP) controllers will be next, though in theory MPF can talk to just about anything.)

Control of each of these different types of hardware is done via a platform interface (found in the `mpf/platform` folder) and is completely transparent to you. Often times switching hardware platforms is as simple as making a few lines of changes in your configuration files and MPF does the rest.

MPF can communicate with multiple platforms at the same time. For example, you might use a P-ROC to control your switches and coils, a FadeCandy to control RGB LEDs, and a SmartMatrix controller to control a color RGB LED DMD. You can do this all at the same time and that's fine.

MPF's platform interfaces contain the "low-level" driver code from the various hardware makers which is translated back-and-forth to a format that MPF can use.

For example, pulsing driver #20 for 30 milliseconds via a P-ROC requires a Python command that looks like this:

```
self.proc.driver_pulse(20, 30)
```

On a FAST pinball controller, the same command is done by sending the following string to a virtual serial port:

```
DN:14,89,00,10,1e,ff,00,00
```

Don't be misled by the apparent complexity of the FAST interface. The FAST interface includes all the settings, including pulse strength and recycle time, in a single command, whereas the P-ROC uses separate commands for each. The point is that we read and understand all the details of the hardware interfaces so you don't have to. :)

Smart Virtual Platform

The *Smart Virtual Platform* is based on the [Virtual Platform](#) interface with one key difference: The Smart Virtual platform watches for coil pulse events and simulates balls moving from device to device by activating and deactivating switches as necessary.

The Smart Virtual platform is the default platform that's used if you don't specify a platform in the *hardware: platform:* section of your machine config file.

To understand why the smart virtual platform exists, consider this simple machine configuration for a trough, a plunger lane, and keyboard key mappings to simulate their switches:

```
switches:
  s_trough1:
    number: s31
  s_trough2:
    number: s32
  s_trough3:
    number: s33
  s_trough4:
    number: s34
  s_plunger_lane:
    number: s27

coils:
  c_trough_eject:
    number: c01
    pulse_ms: 25
  c_plunger_eject:
    number: c03
    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
tags: ball_add_live

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 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.

Of course you can 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, but doing this manually is difficult. Why? Because ball devices in MPF are "smart". So when the trough coil fires, if MPF doesn't see the ball leave the trough (by the switches changing), then it will mark the eject as failed, retry the eject, see if the switches changes, retry again, etc. Ultimately it will decide the trough coil is broken and mark the coil as broken.

The same is true for the timing between the ball leaving the trough and the ball entering the plunger lane. If you don't hit the plunger lane key fast enough, MPF will think the ball got stuck somewhere in between the trough and plunger and try to find it.

One workaround is to set your eject timeouts to be really long to give you enough time to deal with things, but that means you'd have to manage two different sets of configurations for real and virtual hardware.

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 used for a ball device's eject coil, then it looks to see if there are any balls in that device. If so, it knows that in a "real" machine, that coil pulse would move the ball out of that device, so it automatically deactivates one of the ball switches from that device.

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. So simply starting a game puts the ball in the plunger lane without you having to mess with the "1" and "P" keys.

The smart virtual platform interface is used automatically by MPF if you do not specify a platform. You can also manually specify the smart virtual interface for scenarios where you're chaining together multiple config files like this:

```
hardware:
  platform: smart_virtual
```

You can also specify the smart virtual platform interface via the `-X` (uppercase *X*) from the command line, like this:

```
mpf your_machine -X
```

FAST Pinball



The [FAST Pinball](#) controllers are created by Aaron Davis and Dave Beecher from Seattle. FAST has three models—the *FAST WPC Controller*, the *FAST Core Controller*, and the *FAST Nano Controller*.

The *FAST WPC Controller* is used when you want to write your own game code for existing Williams WPC machines. It replaces the original Williams WPC MPU board and leverages the existing Williams driver board and wiring harnesses.

The *FAST Core* and *Nano* Controllers are used with new machines. The two controllers are similar. The Core is a bit bigger and has support for physical DMDs, and an attachment for a BeagleBone Black. Choose it if you want to have a physical DMD, otherwise the Nano Controller is fine. (Any of the FAST controllers can be used with an LCD to create a virtual DMD if you want that style of display in your machine.)

If you want to use a FAST Pinball controller for your MPF machine, our [all-in-one installers](#) (as well as our manual installation instructions) will install the driver you need.

Multimorphic P-ROC / P3-ROC



The P-ROC and P3-ROC are hardware controllers created by Gerry Stellenberg of Austin, Texas-based [Multimorphic](#). The P-ROC controller is a "drop in" replacement in existing WPC and Stern machines, whereas as the P3-ROC controller is designed for new machines.

Multimorphic also has custom driver and interface boards that can be used for new machines with either controller.

MPF supports both the P3-ROC and the P-ROC. (MPF supports the P-ROC operating with P-ROC driver boards as well as with Williams WPC and Stern S.A.M. driver boards. We haven't tested with a Stern Whitestar machine yet but it should work. Contact us for details if you'd like to help us test!)

More information about these controller boards (and ordering information) is available at [pinballcontrollers.com](#). They also have an active [forum](#) and [wiki](#).

Using a P-ROC or P3-ROC with the MPF

If you want to use a P-ROC or P3-ROC for your MPF machine, our [all-in-one installers](#) (as well as our manual installation instructions) will install all the drivers and support libraries you need. You don't need to deal with building and compiling libpinproc and pypinproc as outlined on the pinballcontrollers.com website because we have done that for you with our installers.

Virtual Platform

MPF's virtual platform interface is a software-only platform you can use if you don't have a physical pinball controller attached.

Note that MPF now includes an improved virtual platform interface called the [smart virtual platform](#) which is the new default for MPF and most likely the platform you'll want to use for testing. The traditional virtual platform interface described here still exists, but it's mostly

used now for testing MPF itself since the smart virtual platform is much better for game testing.

When you use the virtual interface, you can "run" your game by using keyboard keys to simulate switches, the OSC interface to send switches from a desktop app or mobile device, or the switch player which lets you write scripts that automatically send switch events based on a timer. (The switch player is great for automated testing!)

Note for P-ROC and P3-ROC users: P-ROC's pyprogame includes a virtual P-ROC interface called *FakePinPROC*. We don't use that in the MPF because doing so requires that pyprogame is installed, and it's likely that people using MPF won't have pyprogame. Using MPF's virtual hardware interface is conceptually similar to *FakePinPROC*.

Snux

MPF's Snux platform interface is an overlay platform interface that lets you use a Multimorphic P-ROC or FAST WPC controller with Mark Sunnucks's (a.k.a. "Snux") System 11 driver board. Details about how this platform interface works are in our [How To guide for System 11](#).

FadeCandy

MPF's FadeCandy interface is used when you want to control RGB LEDs connected to a FadeCandy LED controller. You can mix-and-match the FadeCandy platform with others (e.g. you can use a P-ROC for coils and switches and a FadeCandy for LEDs), and you can even mix-and-match LEDs in a machine (e.g. if you want 512 LEDs, you can use the FAST Controller to drive 256 of them and a FadeCandy to drive the other 256).

Details on how to configure MPF for the FadeCandy are in [this How To guide](#).

Open Pixel

MPF's Open Pixel interface is used when you want to control RGB LEDs connected to a Open Pixel Controller LED controller. You can mix-and-match the Open Pixel platform with others (e.g. you can use a P-ROC for coils and switches and a Open Pixel for LEDs), and you can even mix-and-match LEDs in a machine (e.g. if you want 512 LEDs, you can use the FAST Controller to drive 256 of them and a Open Pixel to drive the other 256).

Details on how to configure MPF for the Open Pixel are in [this How To guide](#). (The linked How To guide is for the FadeCandy, which is a type of Open Pixel Control LED controller. MPF has a separate platform interface for the FadeCandy since the FadeCandy has some advanced features, but that How To guide can be used for general Open Pixel Control devices too.)

SmartMatrix

MPF includes a SmartMatrix platform interface for controller RGB LED matrices which can be used to create "real" color DMDs. Details on how to use this platform interface are covered in [this How To guide](#).

System Modules

The MPF game engine contains several core system components which are included in the `/mpf/system` folder. Some of the components do OS-like things (system timers, an event manager, task scheduling, etc.), while others are more pinball-specific (a switch controller, platform drivers, lighting effects controllers, etc.).

There's really not too much you need to know about the core components other than (1) they are all in the `/mpf/system` folder, and (2) they are all required, so if you delete or change any of them, bad things will happen. :)

We designed MPF to be flexible and extensible, so anything that is not absolutely 100% required for every machine is considered a "plugin" and located in the `/mpf/plugins` folder.

The rest of this documentation section will detail the OS-like core system components. Then we have separate documentation sections which cover the pinball-specific stuff as well as the plugins.

Asset Manager

The *Asset Manager* is responsible for loading and unloading game assets from disk. It's in the `mpf/system/assets.py` module.

- In the MPF game engine, the asset manager is used to manage hardware shows (lights, LEDs, and drivers).
- In the MPF media controller, the asset manager is used to manage media shows (displays, images, videos, and sounds).

Asset loading is done in a separate thread (called the *Asset Loader*), so assets can be loaded in the background without slowing down your game. You can configure assets to be preloaded when MPF starts, to be loaded when the mode that needs them loads, or to be loaded on-demand. The exact decision for how each type of asset will load depends on several factors, like how long the asset takes to load, how fast your computer is, and how much memory your computer has.

Tracking the asset manager loading progress with events

When MPF first starts up, it loads all the assets configured for preload from disk. This happens while MPF is starting, and the attract mode of your machine won't actually begin until the all the assets are loaded. Since this can take awhile on a complex machine, asset loading status events have been created which you can use to show a loading status on the display.

As MPF loads its assets, it sends updated (via [BCP](#)) to the media controller which the media controller can use for the progress update on the display. (Since the media controller has to load its own assets, it will track the combined progress of both the MPF and the media controller assets.)

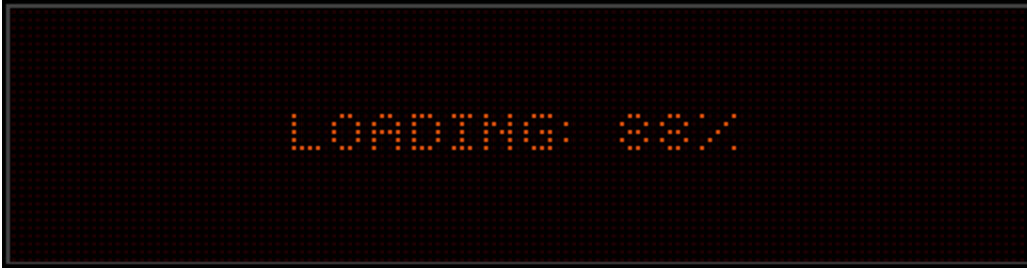
Whenever there's a status update change (on either side), the media controller will post an event called *asset_loader* with parameters *total* (total assets that still need to be loaded), *pc* (number of assets the pinball controller still needs to load), *mc* (number of assets the media controller still needs to load), and *percent* (an integer from 0 to 100 representing the percentage complete the loading process is).

The default config file for the media controller (`mcconfig.yaml`) now includes the following sections in its `slide_player`:

```
asset_loader:
  type: text
  text: "LOADING: %percent%"
  font: small
asset_loading_complete:

waiting_for_client_connection:
- type: text
  text: WAITING FOR
  font: small
  v_pos: center
  y: -4
  slide: waiting
- type: text
  text: CLIENT CONNECTION...
  font: small
  v_pos: center
  y: 4
```

This will create an asset loader screen like this:



You can customize this by creating your own `slide_player:` entry for the `asset_loader` event and doing whatever you want with the `total`, `pc`, `mc`, and/or `percent` parameters.

You might notice that the complete percentage suddenly drops. 88%, 89%, 91%, 92%, 55%... What?!? This is because the complete percent is the combined percentage of MPF and the media controller. So when the media controller first starts, it will start loading its own assets and that's the percentage you'll see. Then when MPF connects, the media controller will merge in the count of its assets, and MPF might have more to load than the media controller, so the overall percentage goes down. (In the future we can fix this by having the media controller cache the number of assets MPF needs to load so the media controller can give an accurate number from the start.)

Once the media controller loads its assets, if it doesn't have a connection to MPF, it will post a `waiting_for_client_connection` event that you can also hook, like this:



Also note that there's an `asset_loading_complete` event posted when the asset loading process finishes. In the sample config above, there's an empty `asset_loading_complete:` section which will post a blank slide (to clear out the slide with the percentage so it doesn't stay at `LOADING: 100%` forever).

Ball Controller

The Ball Controller is responsible for knowing where all the balls in the machine are at any given time. It lives in the `mpf/system/ball_controller.py` module.

The ball controller keeps track of which balls are in which ball devices, how many balls are known and are missing, and tracking balls that drain from the playfield.

The ball controller also makes sure that all the balls are in their "home" positions before a game can start, and it's responsible for moving balls into position as needed.

BCP Interface

MPF's BCP interface is responsible for connecting the MPF game engine to other processes via the [Backbox Display Protocol](#) (BCP). The BCP interface code is in the `mpf/system/bcp.py` module.

MPF's BCP interface can communicate with multiple BCP servers at the same time, and it's responsible for receiving messages via BCP and translating them to MPF actions as well as for watching MPF for actions that should be sent out to BCP servers and building and sending those messages.

The BCP interface runs as two standalone threads—one for sending and one for receiving—for each BCP server that MPF is attached to. (MPF can dynamically connect and disconnect from BCP servers while its running.)

How MPF's BCP interface works

BCP 'get' and 'set' commands

- Incoming *get* commands are turned into events in the format `bcp_get_<name>`. It's up to an event handler to register for that event and to send the response BCP *set* command.
- Incoming *set* commands are turned into events in the format `bcp_set_<name>` with a parameter `value=<value>`. It's up to an event handler to register for that event and to do something with it.

Note that BCP *set* commands can contain multiple key/value pairs, and separate events will be posted for event for each pair

Config Manager

MPF's *config manager* is responsible for loading, parsing, processing, and validating the settings in config files which MPF uses for your machine-wide and mode-specific configurations.

Even though we make a big deal about how you can do 90%+ of your game "programming" via config files, it's not actually necessary to do so. (If you want to use MPF but want to setup everything in Python code instead of using the config files, that's fine by us!) But if you do choose to use config files, MPF's config manager is what makes that happen.

Delay Manager

Often times in pinball programming, you need to set a "delay" for something to happen in the future. You can't just tell MPF to sleep() because that would mean that all of MPF would stop running and your game would stop, so instead MPF contains a Delay Manager which is responsible for scheduling and servicing delays for actions which need to happen at some future time.

MPF's delay manager lives in the *mpf/system/tasks.py* module.

Delays are used for one-time things you want to happen at some point in the future. You can create, reset, and delete as many delays as you want. Example uses of delays in the framework include:

- The ball search feature sets a delay to start the ball search anytime there's at least one uncontained ball. (The length of the ball search delay is specified in the configuration files, typically 10 seconds.) Then once that delay is set, any time a playfield switch is hit, the delay is reset to start that many seconds in the future again. When there are no more balls uncontained, the ball search delay is removed.
- Delays are used with ball devices any time a ball switch changes state. This allows for the balls to settle so the device can get an accurate count. For example, when a ball enters the trough and rolls down to the end, all the switches along the way will quickly change between "active" and "inactive" as the ball rolls by. Each switch change in the trough sets (or resets) a ball count delay for 1/2 second in the future (also configurable), so the trough device doesn't actually count the the switches until the delay expires.
- You can use delays to display information on the DMD for a short duration. For example, when a ball is saved you could write the words "Ball Saved!" to the DMD immediately, and then schedule a delay for two seconds in the future to remove those words.

Device Manager

MPF's device manager is responsible for loading and unloading MPF devices. (Most devices in MPF can be loaded on a machine-wide or a per-mode bases.) The device manager lives in the *mpf/system/device_manager.py* module.

The device manager is also responsible for maintaining collections of devices (via the DeviceCollection base class).

You can connect to the device manager to see all the devices in MPF (as well as their properties), and the device manager can dump a configuration of all devices to a file for troubleshooting purposes.

Data Manager

The MPF Data Manager is used to load and save data to files on disk. The data manager lives in the *mpf/system/data_manager.py* module.

There are many different types of data that are written to disk, including audits, configurations, high scores, credits, and anything else that MPF needs to "persist" when MPF is not running. The data manager provides a centralized service for reading and writing data which is used by several components of MPF.

By default, machine-specific data is saved in the *<machine_folder>/data* location. The current data files you'll find are:

- audits.yaml (audits)
- earnings.yaml (earnings, if you're using the credits mode and your machine is not set to free play)
- high_scores.yaml (high score names and values)
- machine_vars.yaml (machine variables that are set to persist)

Event Manager

The Event Manager is responsible for all aspects of MPF's event-driven control system. It lives in the *mpf/system/events.py* module.

The event manager handles registering and removing event handlers, receiving commands to post events, the processing and calling of handlers for events that have been posted, and for calling callbacks once events have been completed.

The event manager maintains a queue of events that are in-progress.

The event manager is also responsible for setting up and maintaining event monitors which are ways that other processes can be notified of all events that are posted. (The BCP interface uses this, as well as tools that plug in to show the state of the event system.)

File Manager

MPF includes a *file manager* that provides a common interface for various configuration and data files that MPF uses. The file manager is in the *mpf/system/file_manager.py* module.

Examples of the types of files and MPF reads from and writes to include config files (MPF config, your machine-wide configs, and your mode configs) as well as data files like audits, operator settings, high scores, etc.

The file manager uses file interface plugins to read and write different formats of files. By default, everything MPF does with files is in a YAML format, though it's possible to use other formats if you prefer (XML, JSON, etc.)

Hardware Show Controller

MPF's hardware show controller is responsible for playing hardware shows. It lives in the *mpf/system/hw_show_controller.py* module.

A hardware show is a scripted series of actions that happen at certain times. These used to be called "light shows" though we changed the name to "hardware shows" since they can also include LEDs, flashers, and coil/driver actions. Shows can be played, paused, stopped, sped up, or slowed down. Each show plays at a certain priority, and lights or LEDs in higher priority shows will take precedence over those in lower priority shows.

The hardware show controller is responsible for managing all of this.

Logic Blocks Manager

Logic Blocks are used to configure flowchart-style game logic via the machine configuration files.

Logic Blocks are one of the most powerful features of MPF. You can use them to implement simple game logic. It's probably easiest to understand them with some examples:

- If either banks of drop targets is complete and the ball is shot into the center hole, light the special.
- If the lit special is hit, award a special and then reset the special.
- If the tilt switch is activated more than 3 times within a single ball, process a tilt.
- If the player locks three balls, start multiball
- If the player shoots three ramp shots within 10 seconds, light a hurry-up shot to a specific target
- etc.

There are currently three different types of logic blocks in MPF:

- **Accrual** logic blocks are used when you want to track progress towards some goal where the individual steps can be completed in any order. (You just have to "accrue" all the steps.)
- **Sequence** logic blocks are used when you want to track progress towards some goal, but where each step has to be completed in order.

- **HitCounter** logic blocks are used when you want to track a certain event happening again and again, with optional support for decay, counting multiple hits within a certain window as one hit, etc.

The real power of logic blocks is that you can chain together individual blocks to create fairly sophisticated game logic.

Logic Blocks for your game are configured in the machine configuration file, and you can read the config file reference for each block type for detailed implementation instructions and examples.

Do you have to use Logic Blocks?

We love the concept of Logic Blocks and use them extensively for the games that we program. Even though we know Python and could implement game logic in pure Python code, we don't want to reinvent the wheel each step of the way. We designed the Logic Blocks to meet our needs as game programmers, and we use them a lot.

That said, just about every aspect of the MPF is optional, so if you don't like the idea of trying to figure out how logic blocks work and you'd like to write your game logic in Python, then go right ahead! That's totally fine. And if you're not that good with Python, or if you want to quickly build-out your basic game logic, take a look at Logic Blocks.

Of course it's unlikely that you'll be able to build 100% of your game logic with logic blocks alone, but you can definitely get a lot done. (If you think about it, pinball game logic really is just a giant flowchart of "if this then this, then this, this, and this, then that, or if you do this instead, then that..", etc. Logic Blocks can handle most of those scenarios for you.

Machine Controller

If MPF is a pinball operating system, then the machine controller is its kernel. The machine controller lives in the *mpf/system/machine.py* module and is responsible for almost everything, including:

- Loading and processing all the core system modules, plugins, and devices.
- Loading and processing the machine configuration files.
- Loading and processing the machine-specific scriptlets.
- Setting up the platform interfaces which communicate with the physical hardware.
- Resetting the machine.
- Running the main game loop.

If you're writing Python code, the machine controller is typically accessed via `self.machine`.

One quick side note: The pinball community tends to use the terms "game" and "machine" interchangeably. (We're certainly guilty of this too. We would say that "Road Show is our favorite game", not "Road Show is our favorite machine.") That's fine, and we're not on a mission to change peoples' terminology.

But in the context of the Mission Pinball Framework, a "game" and a "machine" are not the same thing. In MPF, "machine" is a physical machine, and "game" is an actual game that's in progress. So that's why you build your "machine" code and your "machine" configuration. The "game" for us is what happens when players are actively playing a game. (So when an MPF-powered machine is in Attract mode, your machine is running, but you don't have a game.)

Mode Controller

The mode controller is responsible for starting, running, and stopping modes in an MPF machine. It lives in the `mpf/system/mode_controller.py` module.

In traditional pinball machines, you might think of a "mode" as a "game mode." (e.g. the missions in *Star Trek*, the mansion modes in *Addams Family*, the cities in *Road Show*, etc.) In MPF, the in-game modes are "modes" too, but so is the attract mode (and even the game itself is a mode called "game"). MPF uses modes for the bonus, the high score entry, tilt monitoring, credit monitoring, and lots of other little things you might not think of as actual "modes." (Don't worry. Running modes don't really take up any memory or CPU if they're not doing anything, so if you have 20 modes running at once, that's fine.)

Individual modes have access to switches, the display, timers, delays, logic blocks, devices, light shows, and all sorts of other things, and the mode controller keeps everything straight. The mode controller is also responsible for watching for events that start modes and for making sure that modes start and stop correctly.

Score Controller

The *Score Controller* in the Mission Pinball Framework is responsible for adding (or subtracting) points from a player's score based on what's happening in the game. (Actually, the score controller can add/subtract any values from any player variable, so you can use it to automatically track a player's points, the number of modes completed, extra balls, aliens vaporized, etc.)

The score controller lives in the `mpf/system/scoring.py` module.

The score controller performs its actions based on events, and you can set any player variable to change value based on any event that's posted. See the [scoring: section of the configuration file reference](#) for details on how to set this up.

Scoring-related events that are posted

Events from scores are automatically created via the [player variable events functionality](#) which you can use to update your slides. If you just want to update the value of the score on the display, you don't have to do anything special since the player variable change will update on the display automatically. But if you want to do a little +5000 animation or something, you can hook the player variable change event which is posted like this:

player_<variable> with parameters *value*, *prev_value*, and *change*.

For example, if the player's current score was 1,250,000 and then you have a scoring event with score: 5000, then an event called *player_score* will be posted with parameters *value:125500*, *prev_value: 1250000*, *change: 5000*.

Mode-based scoring

The score controller is tightly integrated with MPF's modes. It automatically tracks the changes from each mode of any player variable from your scoring section and posts events for them.

For example, if you have this section in a mode called frenzy:

```
scoring:
  any_target_hit:
    score: 5000
    targets: 1
```

Then whenever the *any_target_hit* is posted, in addition to the regular *player_score* and *player_target* events you would expect, you'll also get *mode_frenzy_score_score* and *mode_frenzy_targets_score* events, each with *value*, *prev_value*, and *change* parameters that have the values from that mode only. (The formula for the event names for mode scoring is *mode_<mode_name>_<player_var_name>_score*.)

A few other notes on scoring:

- Scoring now **MUST** be done in a mode config. It is not valid to have scoring entries in your machine-wide config.
- Scoring can be used for much more than just scores. Since bonus, high score, and scoring is all tied to player variables, you can get creative. Use it to track progress, jets, ramps, aliens zapped, modes complete, etc.
- You can enter negative values to subtract scores or to "count down" scores.
- In the future, we'll add the ability to specify scoring multiplier events.
- Scoring entries only work when a ball is in play.

Shot Profile Manager

The shot profile manager is responsible for loading, parsing, and registering shot profiles. It also figures out which profiles should be applied to which shots.

The shot profile manager lives in the `mpf/system/shot_profile_manager.py` module. The actual shots (and shot groups) that the shot profile manager applies profiles to are implemented as devices and are covered in the devices section of the documentation.

Switch Controller

The Mission Pinball Framework includes a core system component called the *Switch Controller* which 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 them that a switch action happened, rather than all different parts of the game all talking to hardware.

MPF's switch controller is based on the `SwitchController` class located in the `/mpf/system/switch_controller.py` module.

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 "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 tablet (via the OSC plug-in). 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.

System Timing & Timers

MPF includes a timing module (`mpf/system/timing.py`) which includes the *Timing* class (used for the main system timer) as well as a *Timer* class (used to set up periodic system timers).

Machine Tick Speed (Hz)

One of the configuration options that you specify when using the Mission Pinball Framework is a constant called *HZ* which is how many "ticks" the framework software runs per second. Basically the machine does some action, then goes to sleep, then wakes up to do more actions, then goes to sleep, etc. as many times per second as what you set your HZ rate to be.

The timing module controls that. (For a detailed discussion on this topic, read the [Machine HZ & Loop Rates document](#).)

Periodic Timers

In addition to the system timer that "ticks" along at the machine's Hz rate, the Mission Pinball Framework also has periodic timers. A periodic timer is simply a method that you register to call on a regular basis (every x seconds or milliseconds). You can have as many periodic timers as you want and they can all be different rates and be started, stopped, created, or deleted at any time.

For example, countdown modes employ periodic timers that are called once per second to do their countdown and to update a variable which holds the number of seconds remaining in the mode. The DMD code uses a periodic timer set to the frame rate to update the DMD, and the light controller uses a periodic timer to update all the lights multiple times per second.

Tasks

The Mission Pinball Framework includes a Tasks module (`mpf/system/tasks.py`) that lets you create, start, sleep, and kill tasks. This module uses the coroutines design pattern. If

you're not familiar with this, check out David Beazley's excellent tutorial "[A Curious Course on Coroutines and Concurrency](#)." (Actually if you want to take your Python skills to the next level, [his book is awesome too](#).)

MPF makes heavy use of tasks for the stuff it does internally. You can use tasks for modes and features you write.

Devices

A *Device* is a logical piece of pinball hardware. Every pinball machine is made up of hundreds of devices. Devices are hierarchical, meaning that certain devices can be made of up other devices.

For example, there are low-level devices are like switches, coils, and lights. Each of these is a device with certain properties (name, status, etc.) and methods (turn on, fire now, are you open or closed, etc.)

Then for the next level up, you can group these low-level devices to create higher-level, logical devices. For example you might group an eject coil and seven optical switches into a ball device you call the "trough," or you might group a switch and a coil together into a logical level device you call a "pop bumper."

Even these logical devices can be made up of other devices. A single switch low-level device might be used to create a logical device which is a single drop target. Then you might combine five individual drop targets, plus a reset coil, into an even higher level logical device called a drop target bank.

Here's a list of all the devices that exist in the MPF:

Low-level devices

- Switch
- Coil
- Matrix Light
- RGB LED
- Flasher
- GI String

Logical devices

- Autofire Coil
- Flipper

- Ball Device (anything that holds balls, like the trough, plunger lane, playfield lock, etc.)
- Autofire Coil
- Score Reel
- Score Reel Group
- Drop Target
- Drop Target bank
- Shot
- Shot Group
- Diverter

Abstract Devices

- Ball Lock
- Ball Saver
- Multiball

Low-Level Devices

Low-level devices in MPF are the lowest-level hardware devices. These are the things that the pinball controllers interface with, including lights, LEDs, switches, and drivers / coils.

Low-level devices are tied to hardware platforms and send & receive hardware commands.

Coil (Solenoid)

MPF supports coils (also called solenoids or drivers).

There are several actions you can perform on coils, including:

- **Pulse:** A predefined 'pulse' of a certain number of milliseconds is sent to the coil. This is how most coils in pinball machines work, including slingshots, pop bumpers, the trough eject coil, ball poppers, drop target reset and knockdown coils, ball launch plungers, etc. Pulses can be from 1 to 255 milliseconds.
- **Enable:** A coil is enabled (i.e. held in the "on") position. This is used for some types of coils that are designed to be held on, such as the flipper "hold" windings as well as certain diverters and coil-controlled gates. Note that you should never "enable" a coil that isn't designed to be enabled. If you just turn on a regular coil and hold it on, it will overheat and burn out. So make sure you only enable coils that are meant to

be enabled. (In fact to protect against this, you have to specifically configure an "allow enable" setting for each coil you want to enable. We put this into MPF as a safety precaution to make sure you don't accidentally enable a coil that isn't designed for it.

- **Timed Enable:** This is like a pulse that lasts longer than 255ms. It's useful for things like diverters (where you may want them to enable for a few seconds at a time). Under the hood this actually implements an *enable* command with a delay scheduled to automatically disable the device when the time ends.
- **Disable:** This is simple. It just disables (i.e. "turns off" a coil)

Configuring your coils

Coils are configured in the [coils: section](#) of the machine configuration file.

Step-by-step tutorial for coils

The step-by-step tutorial covers coils in the following sections:

[Step 4. Get flipping!](#)

Driver-Enabled Device

A driver-enabled device is a type of device that has two states (enabled and disabled), and it's enabled by holding a driver on. (e.g. when the driver is on, the device is enabled. When the driver is off, it's disabled.)

These are rare in modern machines but were common in early solid state machines (up through System 11 and some early WPC machines).

An example of a driver-enabled device is the flippers in System 11 machines. In those days they didn't have the computing power to process flipper button activations in software, yet they still needed to be able to disable the flippers when a game wasn't in progress or when the player tilted. So they were implemented in a way where they had a driver which they turned on to enable the flippers. (That driver was connected to a relay.) When it was on, the flippers were enabled. When it was off, they were disabled.

MPF has a special device type called a driver-enabled device which is used for these types of devices. They're configured in the `driver_enabled:` section of your config files.

Flasher

A flasher is just a really bright light that's controlled by a driver circuit instead of a light circuit. (Driver circuits are used since they are controlled by MOSFETs that can support more voltage than the low-voltage lighting circuits.) Most flashers run at 24vdc.

Even though flashers are really no different than coils under the hood, MPF treats flashers as their own types of devices so you can use more "light-centric" commands for them.

You can also include flashers in shows, meaning you can coordinate complex flasher actions with your existing light shows.

Configure your flashers in the [`Flashers:` section](#) of your machine configuration file.

GI String

The Mission Pinball framework includes support for GI (general illumination) strings which are common in existing Williams and Stern machines. We allow you to specify GI String devices which you can then enable, disable, or (if the hardware supports it) dim. (GI Strings tend to be a relic of existing machines. Most new machines with RGB LEDs just use more RGB LEDs for their GI strings which means that each GI LED is individually controllable and color-able. So when you use a P-ROC or FAST controller with a new custom machine, you wouldn't use this GI string device type.

GI Strings are actually kind of complex. Many of them are AC (even in new 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.

The MPF hides all this complexity from you. You just define your GI strings in your machine configuration file and then you can enable, disable, and dim the dimmable ones as you wish.

LED

The Mission Pinball Framework supports RGB LEDs, like those found in Jersey Jack's Wizard of Oz or Stern Star Trek Premium or LE. RGB LEDs are *not* controlled via a traditional lamp matrix, rather, they're driven by LED driver boards. MPF supports several types of LED driver boards, including the PD-LED (for use with a P-ROC or P3-ROC), serially-controlled LEDs driven by the FAST Pinball controller (which has four built-in channels which support up to 64 RGB LEDs each or a [FadeCandy](#) from Adafruit.

MPF can set an RGB LED to any color, and it can include RGB LEDs in shows, scripts, playlists, and layers, and it can smoothly fade from one color to another and "blend" LEDs with colors set in lower-level layers.

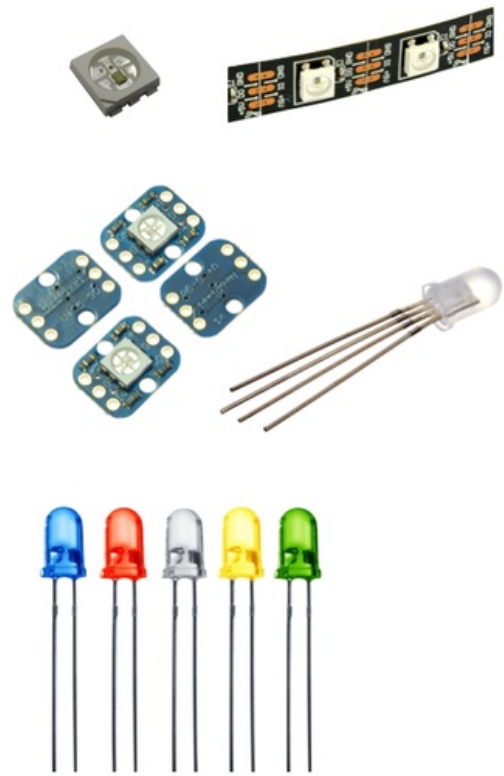
MPF also supports the concept of "brightness_compensation" which lets you configure the maximum brightness for an RGB LED. Brightness compensation is a floating point value between 0 and 1, and its used as a multiplier for all commands that are sent to an LED. For example, if you turn on an RGB LED to be bright white, its color would be `ffffff` (or, in integers, 255, 255, 255). If you set the brightness compensation to `0.85`, then MPF would actually send that LED a value of `[217, 217, 217]`. ($255 * 0.85$). You can set brightness compensation on entire RGB LEDs or on individual elements, allowing you (or your customers) to "fine tune" LEDs to their liking (e.g. white point adjustment), to "turn down" an entire machine so it doesn't blow out their eyes in a dark bar, or to compensate for inconsistent elements in a single LED package.

Note that MPF can support new-style direct controlled LEDs (which are typically RGB) as well as incandescent replacement LEDs added into a lamp matrix. (Though the latter are actually seen as "matrix lights" as far as MPF is concerned, and you can't control the color or fade them as smoothly as direct-controlled LEDs.)

“MatrixLights”



“LEDs”



Matrix Light

In MPF, the "matrix light" device is anything that's connected to a traditional lamp matrix, regardless of whether they're incandescent lamps or "drop in" replacement LEDs like those from CoinTaker. (If you're using RGB LEDs, those are covered [here](#).)

MPF lets you turn on and turn off individual matrix lights, as well as set levels of brightness and fade rates. (Due to the realities of how lamp matrixes strobe through all their columns, we can only get 12 levels of brightness (between off and full on) for lamps or LEDs connected to a lamp matrix, whereas we can get 255 levels per channel on LEDs connected to an LED controller.

You set up all your lights in the [matrix_lights: section of your machine configuration file](#), and they can driven, pulsed, flashed, and added to shows via the lighting commands in the Hardware Show Controller module.

Switch

A switch in a pinball machine is like any switch you'd expect. It's used for all switches, including cabinet buttons, rollovers, targets, optos, trough switches, DIP switches, etc. MPF supports matrix switches, direct switches, and Fliptronics switches.

They only have two properties: active or inactive. (Note we don't say "open" or "closed" because sometimes switches are normally-closed which mean they're actually active when they're open.) In the MPF, you [specify your switches in your machine configuration file](#), as well as whether each switch is normally open or normally closed. (Then the framework translates those into "active" and "inactive" events for the machine.)

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.

MPF Events posted by Switches

Event	Type	Description
<code>sw_tagname</code>	Standard	When a switch moves from the "inactive" to "active" state, the Switch Controller will post one event for each tag the switch is configured for. The event name starts with "sw_" followed by the tag name.

Configuring your Switches

Switches are configured in the [switches: section](#) of the machine configuration file.

Tutorial for step-by-step switch setup

The step-by-step tutorial covers switch setup in the following sections:

[Step 4. Get flipping!](#)

Logical Devices

A logical device in MPF is a device that is composed of multiple low-level devices.

Remember that low-level devices are used to represent the lowest-level, physical hardware in a pinball machine, including switches, lights, and drivers (coils). Logical devices are based on groups of low-level devices which work together.

For example, a flipper is a logical device because it groups a switch (a low-level device) with a coil (another low-level device) to become a flipper which can be enabled or disabled or have its properties changed based on what's going on in the game. The trough is a logical device that combines a bunch of switches and an eject coil into a ball device which knows how to receive and count balls, how to eject them, whether the eject has failed, etc.

Ball Device

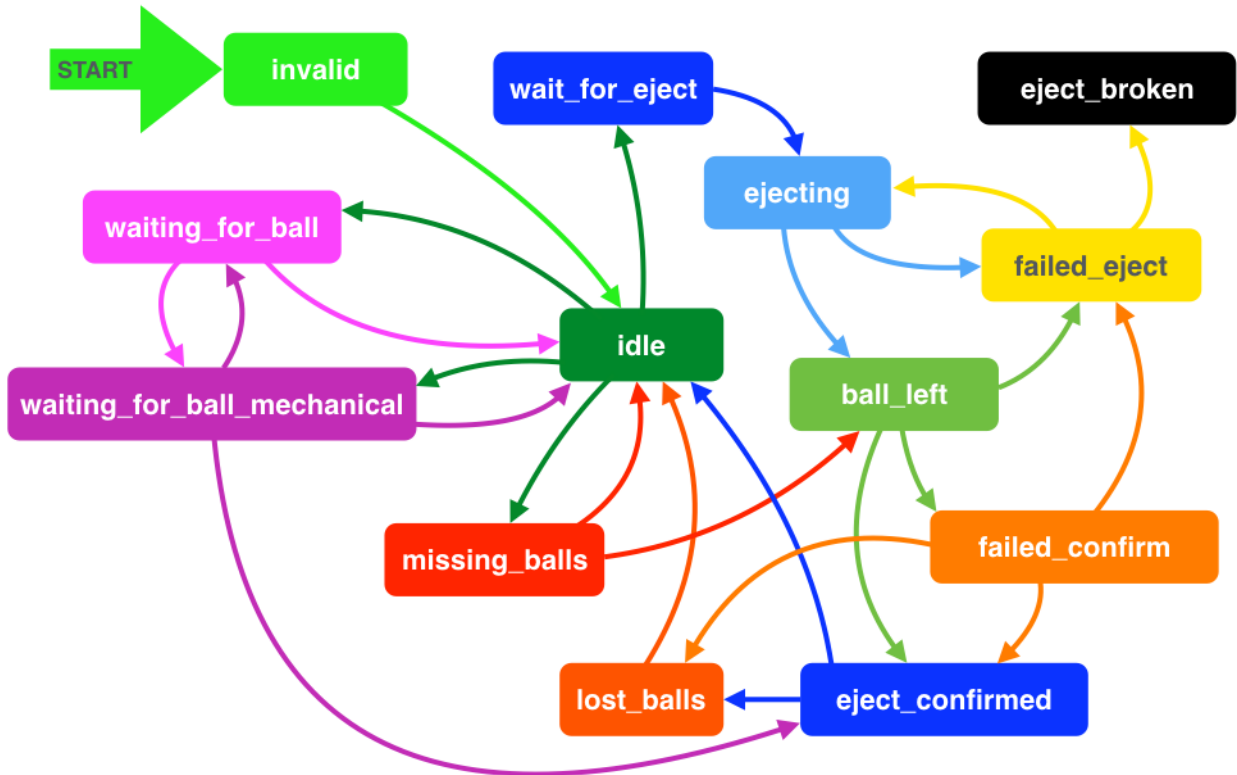
A *ball device* is any device in a pinball machine which is able to hold (i.e. "capture") a ball and then release it. (Either automatically or based on some action by the player.) Examples of ball devices include the trough, the plunger lane, VUKs, poppers, playfield locks, etc.—basically anything that can hold a ball. (Even the playfield is 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. (In MPF, the [playfield](#) is a type of ball device.)

Ball devices are very intelligent in MPF and 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.



*There's an unshown valid path from all states back to *invalid* since calling the device's `stop()` method will reset them back to *invalid*, recount the balls, and then move them to *idle*.

Ball devices know where they fit in the "chain" (so they know which devices feed them and which devices they feed), and they work with the ball controller to help MPF know where all the balls are at any given time.

Diverter

The Mission Pinball Framework includes specific device drivers to handle diverters. We consider a diverter to be anything that alters the path of the ball when it's enabled, including:

- A traditional diverter which is a metal flap at the end of a rod (typically used on ramps).
- 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.

- Something else completely custom, such as the Ringmaster in *Cirque 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???" If you think about it from the perspective of pinball software, yeah, trap doors and the Ringmaster *are* diverters because when they are not active, a ball shot to them goes towards one place, and when they're active it's "diverted" to go somewhere else.

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, so nothing is "active" to hold it on.)

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 module.

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.)

Our 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.

Diverters are configured in the [Diverters: section of your machine configuration file](#). You can see the full details of options (and how you configure them) there.

Drop Target

Drop Target devices in the Mission Pinball Framework inherit all the features and functions of [regular targets](#) with additional functionality for reset and knockdown coils.

You can group multiple drop targets together into a Drop Target Bank, giving you functionality to query how many of the targets are currently complete, to reset the entire group, and to post events and run shows when the bank is complete.

Autofire Coil

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, BeagleBone, etc.) which runs your game, controls the DMD, 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...). The MPF Python code tells the hardware controller to fire a coil or turn on a light. When a switch changes state in the pinball machine, the hardware controller sends the state change to the MPF game code running on the computer.

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 write 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!

How your Python game code 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 your python game code 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. I mean 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 second 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 the Mission Pinball Framework 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 our framework 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.

Flipper

Flippers are pretty important to pinball machines (obviously) and there are a lot of different types and different ways they can be setup. The Mission Pinball Framework supports all the different options.

Understanding flipper concepts

We know you might be thinking, "Why is this a section? I push the button, the flipper flips. What's so hard about that?" Well, we'll show you.

At the most basic level, yes, a flipper is just like any other coil which is tied to a switch. When the flipper button is active, the flipper coil is engaged. When the switch is deactivated, power to the coil is cut. Simple!

Unfortunately flipper coils are more complex than that. A normal coil in a pinball machine that fires a popper or slingshot is only ever active for a few tens of milliseconds at a time. But a flipper coil can be "held" for multiple seconds or even minutes at a time, and if the machine kept the full current flowing through the coil that whole time, the coil would quickly burn up. (Remember that nostalgic smell of model railroad motors from your childhood? Yeah... you don't ever want to smell that in the pinball world. :)

So pinball flippers need a way to stay active without burning up. There are two different ways that pinball manufacturers solve this problem. (It varies depending on the company and the year.)

Option 1: The flipper has two windings, a "power" and a "hold"

Some pinball machine flipper coils are actually two coils in one. They literally have two sets of windings. One is strong and high powered to give the flipper the strength it needs to do the initial stroke, and the second winding is much weaker and draws less current—strong enough to hold the flipper up but weak enough that the flipper doesn't burn up.

You can tell if you have this kind of flipper because the coil will have three terminals instead of two. There's terminal for the high power winding, another terminal for the weaker hold winding, and a third terminal which is the common shared by both.

Typically the pinball machine enables both flippers when the flipper button is first pushed, and then the power to the main winding is cut when the flipper first makes it into its "up" position. Then when the flipper button is released, the power is also cut to the hold winding.

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 far 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.)

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 starting pulsing the power.

How important are 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. They 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.
- 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.)

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 in that cause 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 room in your switch matrix, 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.)

Configuring your flippers

So given all that, how should you actually configure your flippers in your game? The answer depends on how you use each of the options:

Design Decision 1: Dual-wound coils or single-wound?

Dual-wound means you use the traditional flipper coils with high power and low power windings. The only real downside to these is that you need two driver outputs per flipper.

If you're reprogramming an existing pinball machine that already exists, then you'll probably just use whatever types of flipper coils the machine already has. The only way we could see not doing that is if you wanted to "free up" some driver outputs to control additional accessories you'd like to add into the game. For example, maybe you want to add some kind of spinning toy and you need an extra driver output. If the machine is currently using all the outputs, you could disconnect the hold coils from your flippers and hook those outputs to your new toy. Then you'd configure the flippers in the Mission Pinball Framework so they only used the power coil with the short full power then pulse modulated holds.

Personally we feel this pulse-then-modulated approach with a single winding on the flipper coil is fine. The dual-wound coils are really a relic of the EM days when they didn't have the fine control needed to quickly pulse the coils in the hold position. (And in fact Stern has been doing this for years.)

The only downside we hear about this is that some people just don't like it. If you ask why, they can't really give a reason other than saying they don't like it. So as far as we're concerned, why not go this route and free up those additional driver outputs?

Then again, if you have the room, eh, why not stick with dual wound?

Design Decision 2: Pulse timings or EOS switch to indicate "up" position?

Next you have to figure out how your 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?

Finally, you have to decide whether you're going to use EOS switches to notify the machine when a flipper has lost its hold while the flipper button is still engaged. (And if so, what you're going to do about it.)

We believe the chances of a ball breaking the hold are generally slim, and if it's something that happens often that indicates that your hold power is not strong enough. (Assuming you're holding the flipper with the pulse modulation to the power winding rather than using a dual-wound coil.) We also believe that if a ball breaks a flipper hold, automatically reapplying full power to restore the hold can be confusing to the player.

That said, all machines are different, and tastes are different, so you should go with whatever you want. The nice thing about not using EOS switches is again, you can free up those switch inputs for other things if you're running low. But even if you don't use them to automatically correct for a broken hold, we like the idea of still connecting the EOS switches and using them for audit logging purposes. (e.g. using them to record any instances of a flipper hold being broken by a fast moving ball, a broken hold winding, or a broken flipper.)

Applying these design decisions to your game

Once you make these three decisions, you have to actually apply them to your game. In the Mission Pinball Framework this is done via the [Flippers second of the machine configuration files](#), and then based on your settings a series of hardware rules are written to the hardware controller.

Need to add:

enabling & disabling flippers

fancy modes like inverted flippers, no-hold flippers, etc.

Note about how we can detect and notify the operator of broken flippers.

Playfield

The playfield in MPF is actually a special type of [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. 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.

Playfields are configured in the `playfields:` section of the configuration file.

Playfield Transfer

A *playfield transfer* device in MPF is a device that is used to transfer balls from one playfield to another in machines that have multiple playfields. This device is needed when the playfields are directly connected without a proper ball device in between. For example, sometimes a tube connecting two playfields will have a switch in it that's activated as the ball rolls through, letting MPF know that the source playfield has lost a ball and the destination playfield has gained a ball.

Playfield transfers are configured in the `playfield_transfers:` section of the machine config file.

Score Reel Group

A *Score Reel Group* is a logical grouping of multiple [Score Reel devices](#) that are used grouped together to show a multi-digit value in an EM pinball machine. For example, there

might be a Score Reel Group called "Player1" that's a grouping of 4 Score Reel devices, one each for the thousands, hundreds, tens, and ones digits.

Score Reel Groups have properties including which Score Reels are in which positions in the score (thousands, hundreds, etc.), whether there are any fake "blank" positions (like the plastic "0" inserts some machines use to make the scores seem bigger), and how many individual reels in the group may be firing simultaneously. Score Reel Groups have methods that let you do things like add to the score, and reset the reels.

The Score Reel Group functionality in the Mission Pinball Framework is somewhat advanced. Check out [this blog post that we wrote](#) for more information.

Details for configuring Score Reel Groups can be found in the [Score Reel Groups section of the machine configuration file reference](#).

Shot & Shot Group

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.

Many shots in MPF have lights or LEDs associated with them that indicate what "state" the shot is in. (e.g. "shoot the flashing targets" or "hit the lit ramp" could mean "hit the target that's in the *flashing* state", or "hit the ramp that's in the *lit* state".)

Shots in MPF are similar to other devices we've worked with (like autofires and ball devices) where you group together switches, lights, and/or LEDs into the things we call *shots*. (Not every shot has a light or LED associated with it.)

You can even configure shots that are 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.)

You can also group multiple shots together into a shot group. For example, you might have three lanes at the top of your playfield, each with a rollover switch and a light. In that case you'd configure each of them as a separate shot. Then you could create a shot group that grouped all three of the shots together into a logical group. A shot group lets you do cool things, like trigger an event when all of the member shots are in the same state (increase Bonus X when all three shots are "lit"), and you can setup things like "shot rotation" that rotates the state of the member shots left or right (lane change via the flipper buttons, for example).

Score Reel Controller

The Mission Pinball Framework supports EM-style mechanical score reels for anyone brave enough to convert an EM machine to modern digital control. ([We're doing this](#) with a 1974 Gottlieb Bit Shot.) Since this is something that most people won't care about, we won't waste too much time talking about it here in the documentation. But if you're interested, [check out the blog post we wrote](#) with the full details.

Mechanical score reels are managed via the Score Reel Controller (in the `score_reel.py` module) and configured in the [Score Reels section of the machine configuration files](#).

One final note: The score reel support is built into the framework and available on any machine running MPF, so there's nothing stopping you from building an EM score reel topper for a modern machine like Attack From Mars and giving it some mechanical scoring love. (You'd need about 12 digits of course, but it would work! :) Anyone? Anyone??

Score Reel

A *Score Reel* in the Mission Pinball Framework is an electromechanical (EM) score reel unit from an EM pinball machine. This device represents a single reel, and multiple Score Reels can be grouped together (into a [Score Reel Group](#)) to make up a complete grouping that might represent a player's score. The credit unit in an EM machine is also a score reel device.

Maybe we should take a step back. Did you know you can use this framework to control EM pinball machines? It's very cool, but also a lot of work because you have to strip out the EM guts of the machine and completely rewire the machine from scratch for your modern digital control environment. We did this for a 1974 Gottlieb Big Shot machine, and we estimate the effort to be about 100 hours. ([Details of that project are here.](#))

Individual score reels have properties like the number of digits they have, the name of the coil that advances them, positional switches that let the machine know what position the reel is in, and (optionally) whether there is also a coil to decrement the reel (like with a credit reel) and whether or not the reel can "roll over." (So a typical reel in a score unit can roll over from 9 to 0, while a credit reel cannot.)

More details about score reels are available in the [Score Reel section of our machine configuration file reference](#).

Abstract Devices

Abstract devices are logical devices in MPF that are based on concepts rather than physical devices, though they are created and managed just like low-level and logical devices, so they're included here for completeness.

For example, a ball saver in MPF is a "ball saver" device. You might actually have several ball savers in your machine, each running based on different conditions and with different settings. (And maybe multiple ball savers will run at the same time.) You might have a regular ball saver which is active for 10 seconds after a ball starts which flashes the shoot again light, but you might also have one shot that always drains straight down the middle which could have a phantom (unadvertised) ball saver attached to it.

There are also abstract devices for things like multiball and ball locks (whether physical or virtual).

Ball Lock

You can use MPF's *ball lock* device to configure ball locks. Ball lock devices work with ball devices to lock balls for the player, and they automatically keep track of how many balls a player has locked (even if another player causes the ball device to physically release its balls).

Configuring ball locks

See the `ball_locks`: section of the configuration file reference.

Events posted by ball lock devices

The ball locks in MPF will post the following events. Each ball lock you set up has a name, and that name will be used in the `<name>` section of the event names below:

- *ball_lock_<name>_balls_released* - The ball lock has released one or more balls. This event includes a *balls_released* parameter which is the number of balls it released.
- *ball_lock_<name>_locked_ball* - The ball lock has just locked another ball (or balls). This event includes a *balls_locked* parameter which is the number of balls it just locked, and a *total_balls_locked* parameter which is the total number of balls it has locked.
- *ball_lock_<name>_full* - The ball lock is full. A parameter *balls* is included which is the number of balls that are locked.

Ball Save

MPF's *ball save* abstract device is used to configure ball saves. You probably know what a ball save is, but if not, it's where the ball that drains doesn't count and the player gets their ball back. Typically it's used to give the player their ball back if they drain right after their ball starts, or it's used to 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 the latter if possible, and instead, as Lyman Sheets would say, "Fix your f-ing game layout!")

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.

Configuring ball saves

See the [ball_saves: section](#) of the configuration file reference.

Events posted by ball save devices

The ball saves in MPF will post the following events. Each ball save you set up has a name, and that name will be used in the *<name>* section of the event names below:

- *ball_save_<name>_enabled* - The ball save has been enabled.
- *ball_save_<name>_disabled* - The ball save has been disabled.
- *ball_save_<name>_hurry_up* - The active time remaining has decreased into the "hurry up" period.
- *ball_save_<name>_grace_period* - The active_time remaining is up, but the ball save will stay active for the grace_period time.
- *ball_save_<name>_saving_ball* - The ball save is saving a ball.

Multiball

MPF includes a *multiball* abstract device which can be used to automatically start and stop multiballs.

Configuring multiballs

See the `multiballs:` section of the configuration file reference.

Events posted by multiball devices

The multiballs in MPF will post the following events. Each multiball you set up has a name, and that name will be used in the *<name>* section of the event names below:

- *multiball_<name>_started* - This multiball has just begun. A parameter *balls* is the number of balls in this multiball.
- *multiball_<name>_shoot_again* - A ball has drained while this multiball's "shoot again" period is active. A parameter *balls* indicates how many balls will be added back into play.
- *multiball_<name>_ended* - This multiball has ended. This event is posted when there is only one live ball left on the playfield.

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.

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. Most typical 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.

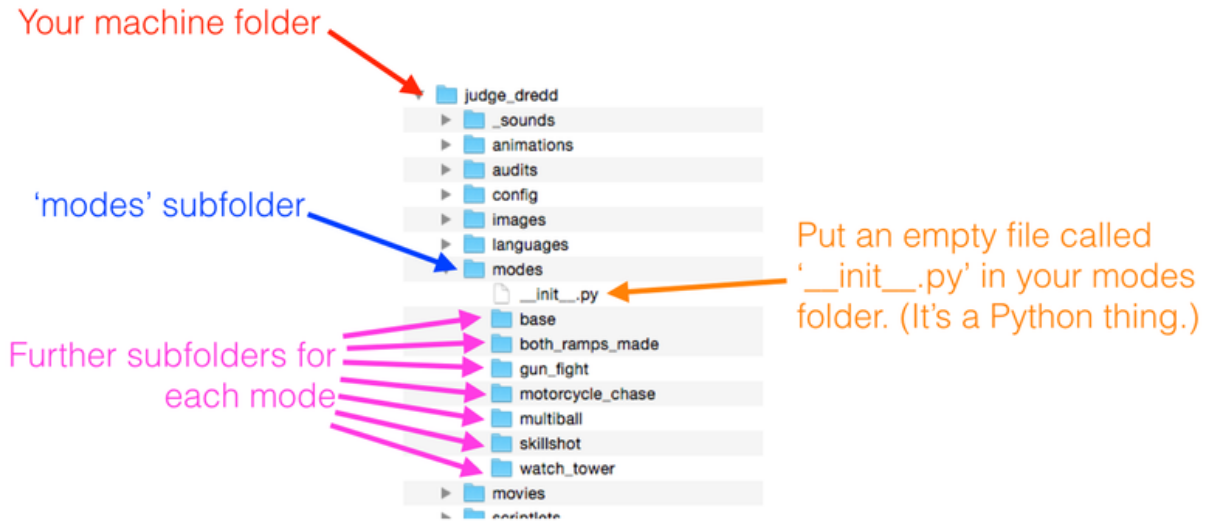
MPF takes a slightly different approach to modes.

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.

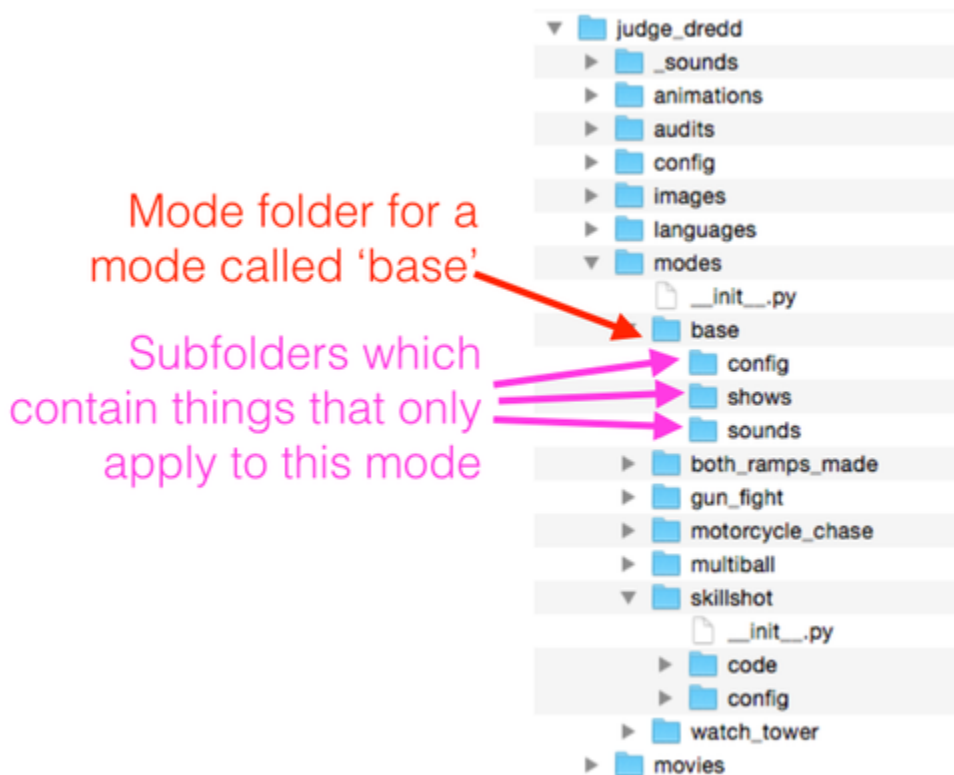
How modes work in MPF

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. (Note that if you want to run custom Python code from a mode, you also have to put a blank file called `__init__.py` in your modes folder. That's two underscores before and after "init". More on that later.):



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:

- Animations
- Images
- Logic Blocks
- Movies
- Scoring
- Shots
- Show Player entries
- Slide Player entries
- Light Player entries
- Timers
- 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:

```
scoring:
  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 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.)

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 specific 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 optional, but it's 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 100l 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 all MPF elements. 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 be 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.)

These are the modes you're looking for...

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.

Built-in Modes

The MPF package 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:

[subpages]

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 list in the *mpf/modes/attract* folder. It is 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.)

Game Mode

MPF's built-in game mode 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 attract mode starts again.

The code and configuration for the built-in game mode lives in the *mpf/modes/game* folder. It is 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.)

Credits Mode

The MPF package 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.

High Score Mode

The MPF package 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.

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.

Tilt Mode

The MPF package includes built-in tilt mode that can be used to track manage tilt warnings, tilts, and slam tilts. See the [How To: Create a Tilt](#) guide for details.

Events

The concept of events is one of the most important concepts in MPF. MPF is an event-driven framework, which means that almost everything that happens generates an event, and then things that need to react to certain events receive notification that an event occurred and can act on it.

Understanding how Events work in MPF

It's easiest to understand the concept of events by going through some examples. (These examples are not actual Python code, rather, they're just some pseudocode-like example.)

The main two actions with events are "posting" events (which is where some component posts an event to the Event Manager), and "handling" events (where some component registers to do some action when an event happens).

So let's apply this to an event like a new ball starting. You can imagine that lots of things would need to happen, which means that lots of components need to register (or "add a handler" in MPF parlance) for that event. For example, when a new ball goes live:

- The ball save timer needs to start.
- The skill shot need to start.
- The DMD needs to switch over to show the current player score.
- The music system needs to start playing the background music.
- The auditing system needs to start the timer to record average ball times, etc.
- The tilt system needs to start watching for tilts.
- Etc.

So when a new ball is starting, the game mode might tell the Event Manager a post an even called "ball_starting." Then the Event Manager will look through its list of registered handlers and look to see if any have been registered "starting." If it finds them, it will contact them one-by-one so they can do whatever they have to do for that event.

The real beauty of using events for certain modules to notify other modules is that your notifying module doesn't know or care whether any (if any) handlers have been registered. This allows the game code to be extremely modular. You can add your own components which can listen for events to "insert" themselves into the game, or you can remove built-in modules without breaking anything. (It also makes it really easy to "program" your game with config files—essentially all your config files are is big lists of what happens when certain events are posted.

Event names in MPF are not case sensitive. If you post an event called "My_Event", it will actually be posted as "my_event." Similarly if you register an event handler to list for an event called "My_Event", it will actually register itself to listen for "my_event".

The Event Manager Queue

The Event Manager only processes one event at a time. So when it gets an event to post, it will call the handlers that have registered for notification of that event one-by-one. If one of those handlers wants to post another event of its own, that's fine. But when it posts the event since the Event Manager is busy, it will add the new event to an event queue. Then after the first event is done (meaning after all the event handlers have been called for the first event), the Event Handler will call the next event in the queue. The event queue can hold multiple events, and they will be called one-by-one in the order they were posted.

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.)

Event Callbacks

When you post an event, you have the option to pass a *callback* as a parameter. A callback is a function or method that you want to be called once the event is done processing. (i.e. it's called once all the handlers that have registered for that event have been called. If no handlers are registered, the callback is called immediately.)

One "gotcha" with callbacks is they're called after the event is done processing. If the event manager is busy (because another event is in progress), then the callback won't actually be called until the actual event is processed, which might not be immediate.

In most cases you combine callbacks with special types of events. So to understand how this all works, we need to look at the different types of events you can call.

Types of Events

There are several different *types* of events in the Mission Pinball Framework, including:

- Basic events
- Boolean events
- Queue events
- Relay events

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.

Basic Events

The basic event is just a simple event name. Your code just calls the Event Manager and tells it to post an event with a given name, like `events.post('your_event_name')`. Then the event manager calls each of the registered handlers one-by-one. If you have specified a callback, like `events.post('your_event_name', callback=self.some_function)`, that callback will be called after the last registered handler has returned after handling the event.

Boolean Events

Boolean events are used when you want to get some feedback from all of the handlers that have registered for that event. When you post a boolean event, if any of the registered handlers return `False` then the event manager will stop processing the event. (i.e. the remaining handlers are not notified.) Boolean events are typically used with callbacks, where the event manager will pass a value of `False` to a callback if one of the handlers returns `False`.

For example, you might have an event called *request_to_start_game* that's a boolean event. When that event is posted (perhaps because the player pushed the start button during the attract mode), the event manager will receive that event and contact all the registered handlers one-by-one. You'd typically post that event with a callback of something like `self.result_of_start_game_request`. Then if any registered handler returns `False`, the event manager will call the callback and pass the `False` result, like `self.result_of_start_game_request(False)`. Then your `result_of_start_game_request()` method might choose to do nothing if it gets a result of `False`, or it might choose to actually start a game if it's called without the `False` value.

What types of handlers might you register for an event called *request_to_start_game*? There could be many. The ball controller might want to make sure all the balls are in their home position. The tilt module might want to make sure the plumb bob tilt is settled and not swinging. If the game is not set to free play, the credits module has to make sure there's at least one credit in the game. Any one of these modules can deny a game start by registering itself as a handler for the *request_to_start_game* event and then returning `False` if it doesn't want to allow the start.

This, by the way, is a great example of the power and flexibility of using events for this kind of thing instead of manually hard coding each of these modules into the game code. If the game is set to free play, then the credits module does not load, so it's not part of the process of watching for a request to start a game. This means your game starting code doesn't have to know anything about a credits module or whether or not it's active. The game starting code just posts the event and will start the game as long as no one denies it. (Once the game start request is approved, then a second event is posted which actually starts the game.)

That's the one that the credits module will register for to actually decrement a credit from the machine.)

This extensibility is how you can add functionality to your own game that might need to approve or deny a game start. For example maybe you have some complex playfield toy that has to be in a known position in order for the game to start. So you could have your game code register a handler for the *request_to_start_game* event which you could deny if your toy wasn't ready to go. That's how you can inject yourself into the game starting process without having to hack any of the core Mission Pinball Framework code.

Note: you can see an example of the *request_to_start_game* boolean event in action in our [MPF Game Start Sequence](#) documentation.

Queue Events

Queue events are used when an event handler wants to temporarily "pause" the event processing while it finishes up some task. This is called a queue event because the event manager literally creates a little queue of events it's waiting for, and then when that queue is cleared it calls the callback.

An example of this might be after a tilt. When that happens the game controller will post a *ball_ending* event (since the tilt ends the ball), but the ball controller might not actually want the game to move on until the ball has drained into the trough. So the *ball_ending* event is posted as a queue event, like this:

```
events.post_queue('ball_ending', callback=self.ok_to_end_ball)
```

When a queue event is posted, the event manager will create an event queue instance and pass it as a parameter to all the registered event handlers. So if your ball controller wants to make sure all the balls have drained before the game moves on, it will register a handler for the *ball_ending* event. In that handler code, if the ball controller is not ready for the ball to end then it can call a `queue.wait()` command to tell the event manager that it would like it to wait before finishing. Then after the ball drains, the ball controller can call a `queue.clear()` to remove it's hold request from the queue. Once that event's queue is totally clear, the event manager will call the callback that was originally included with the event posting.

Here's an example of all this in action. (This should probably move to the Advanced Programming section of this documentation.)

Add a handler for your event as normal:

```
self.machine.events.add_handler('ball_starting', self.block)
```

In the handler method, give it a parameter named “queue”. Also save queue so you can access it later. Do whatever you need to do then call `queue.wait()`. Your handler will be called immediately.

```
def block(self, queue):
    self.queue = queue
    ...
    self.queue.wait()
```

Then in your code that clears the wait:

```
self.queue.clear()
```

Note if none of the registered event handlers call `queue.wait()`, then the callback will be called immediately.

If you want to kill a queue event (i.e. without just waiting forever), then in your registered handler, do two things:

```
queue.kill() # Clears the queue and does not call the callback
return False # Causes future (lower priority) handlers not to be called
```

Relay Events

Relay events are used when you want to pass kwargs from one event to the next. In this case the handler literally takes whatever one event returned and passes them as kwargs to the next event. The idea is you can pass some kwargs around that each event can modify. For example, if a ball drains, the game calls a ball drain event with kwargs `balls=1`. Then if there's some other module that wants to save that ball, it can receive `balls=1` and change it to `balls=0`. Then when the event gets back to the original caller, it has new data.

Note a handler must return a dictionary that will later be packed via `**`. So a handler would do:

```
return {'balls': 1}
```

to have the next handler be called like:

```
handler(balls=1)
```

Relay events tend to work well with callbacks since you aren't guaranteed they'll fire right away. To use a relay event, add `ev_type='relay'` to your event post.

Best Practices for Using Events

When a handler responds to an event, the "flow" of the code goes into that handler. This means that you do **not** want a handler to take too long to return. If there's something that a handler needs to do that takes a long time, it should set up a task, a timer, or register to do work based on the "timer_tick" event. In other words, your handlers should return quickly.

FAQs on events

We've received several questions from users about events, so we're sharing a list of questions that have been asked as well as our answers:

The documentation states, "One 'gotcha' with callbacks is they're called after the event is done processing. If the event manager is busy (because another event is in progress), then the callback won't actually be called until the actual event is processed, which might not be immediate." Does this mean that the callback is called after the event has been sent to all registered handlers or until the current handler is complete?

The callback is called after all the handlers for that event have been called.

When an event is posted, if there's another current event in progress (meaning that the new event was actually posted by a handler from some prior event), then the new event is added to the queue. (The queue is essentially a list of events that still need to be called). So all the handlers for the current in-progress event are called, then the callback is called (if a callback was specified). Then when that callback is done, that event is "done" and the Event Manager checks the queue list to see if another event should be posted.

Technically speaking only the Event Manager can post an event. All the other code bits that post events are really saying, "Hey event manager, can you please post this event?" And the event manager is like, "yeah yeah, I'll do it when I'm not busy."

You can see this in action with verbose logging enabled where the event manager receives an event at one point, but the actual "post" of that event might not happen until hundreds of lines later.

How do boolean results factor into this? This stops the event from being sent to the remaining handlers?

Correct. If any handler returns False, that event is not sent to the remaining handlers. (The order the handlers are called can be set by specifying a priority when a handler is registered.)

If a boolean event has a callback and one of the handlers returns False, the callback is still called with a special parameter `ev_result=False`. This lets you take some action (if you want) on that event failing.

How does the event caller know when all handlers have completed processing?

When you call any method in Python, when that method gets to the end of its code, it will "return" to whatever called it. (Even if that method calls another method, that second method will get to the end of its code and return back to the point that called it in the first method, then the first method will finish and return to whatever called it, etc.)

The Event Handler is essentially just a mapping of event names to handler methods and priorities, so when it sees an event called "foo", it will see there are three registered handlers, so it will call the first one, and when that one returns it calls the second one, and when that one returns it calls the third one, and when that one returns the event method is over and then it returns and the game loop continues.

If you add an infinite loop (or just any loop that takes a long time) into one of your handlers, then MPF will get "stuck" there. So it's up to each handler to do what it needs to do quickly and then return.

The event manager is a big queue. First In, First Out. For example, we have 5 handlers for the event "foo". "foo" will be sent to all 5 before discarding the event and popping the event off the queue in order to send out the next event. But what I am trying to figure out is when the event manager must send to all 5 or when it can terminate early. In other words, if handler #2 returns a False for a boolean event, then handler #3,#4 and #5 never see the event? Correct?

Correct.

Now if it's not a boolean event, is there anything that can also stop/ suppress the event from being seen by all the handlers? Or is it sent to all '5' regardless of the handlers results?

Correct, if it is **not** a boolean event, then the event is sent to all 5 handlers regardless of the results. Nothing can stop it. If you don't want this behavior, then post a boolean event instead of a regular event.

Player Management

MPF has a robust system for managing the players in an active game.

Individual player objects are created (based on the Player class in the *mpf/system/player.py* module), with one instance of that class per player in the game. Players are automatically created as they're added to a game and removed when the game ends.

A key concept of player management in MPF is *player variables*. A player variable is a name/value pair of some attribute you want to store on a per-player basis. (The most obvious example is *score*.)

There are two types of player variables—*tracked* and *untracked*.

Tracked player variables

Tracked player variables automatically post events when they change, and the event they post contains the name of the player variable, the new value, the previous value, and the change in value. You can then use the event generated from the player variable change to do anything you want. (Start or stop a mode, update a timer, show something on the display, play a show, etc.)

Tracked player variables are also sent to the media controller via BCP.

The events posted for tracked player variable changes are posted automatically in the form of `player_<variable_name>`. For example, if player one's score is currently 10,000 and then they get 500 points, an MPF event will be posted called `player_score` with parameters `value=10500`, `pre_value=10000`, `change=500`, `player_num=1`.

Default tracked player variables include *score* (the player's score), and *number* (which player number they are), and of course you can create lots more of them as you build-out your game. (You can configure logic blocks, shot progress, and timers all to use tracked player variables to store their settings.)

Untracked player variables

Untracked player variables are also stored on a per-player basis, but they don't automatically post the events when they change, and they are not sent out via BCP. Untracked player variables are used for the "internal" things you might need to save on a per-player basis but that you don't need to broadcast everywhere. For example, the states of logic blocks are automatically stored on a per-player basis as untracked player variables.

Accessing player variables in code

(This is a thing for people writing Python code. If you're only using config files, you can skip it.) Tracked player variables are used like this:

```
player.ball = 1
```

or

```
player['ball'] = 1
```

Sometimes you don't want to go through all that gunk. Sometimes you just need to store something (or several things) on a per-player basis for your own use, but you don't need it being broadcast everywhere (and running into BCP size limits, etc.). It was impossible to create your own untracked attributes before because the Player class's `__setattr__` method would store whatever you set as a tracked player variable.

Untracked player variables are accessed like this:

```
player.uvars['some_name'] = whatever
```

You can have as many of these as you want per player, all with separate names. You can store whatever you want (values, lists, dicts . . . any Python object).

Use them like you use a Python dictionary. Loop, set, read, iterate, etc.

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 [tracked 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).

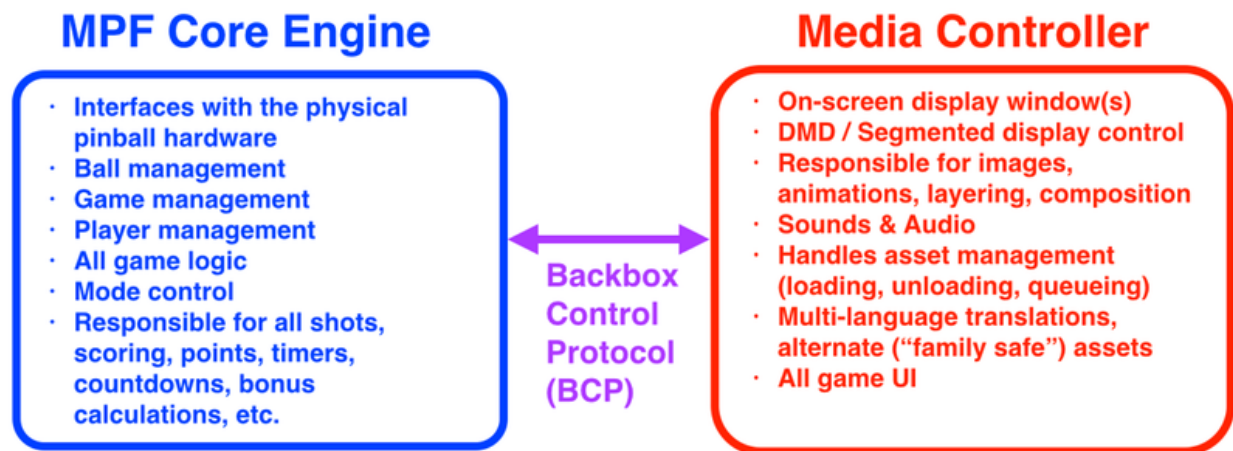
Machine variables can be set to "persist", meaning they are saved to disk and available to MPF the next time it boots up. (For example, if you first turn on a pinball machine, it will still show the scores of the last game played in the attract mode.) These machine variables are stored in the `<your_machine_folder>/data/machine_vars.yaml` 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.)

Media Controllers

The MPF project separates out the "[core engine](#)" from the "media controller." These are literally two completely separate processes that run. (So when you "run MPF" on your computer, you'll actually have two programs running at the same time—the MPF core engine and a media controller.)

The game engine and the media controller talk to each other via a TCP socket with a protocol we invented called "BCP" (for "[Backbox Control Protocol](#)"). Here's a diagram that shows what the MPF game engine is responsible for versus what the media controller is responsible for:



Media Controller Options

When you use MPF, there are currently two media controller options.

- There's a built-in Python-based media controller that we call "The MPF Media Controller." This is the default option that most people use. You can use this for DMD or LCD-based games, and it can play high def video and work with HD displays.
- There's an advanced media controller based on Unity 3D which you can use if you want complete control over the display and audio experience. Using this option requires programming and knowledge of Unity 3D.

The MPF Media Controller

The MPF media controller is included in the core MPF package. You run it from the command line via `python mc.py your_machine`. The files that make up the built-in media controller are in the `/mpf/media_controller` folder.

Everything we cover in the tutorial, how to guides, and this documentation is based on this built-in media controller. It can drive LCD and DMD-based displays and play multi-track audio. All of its configuration is done via the config files.

What's new in the MPF Media Controller (mpf-mc) in v0.30?

One of the big changes for MPF in version 0.30 is that we're changing the MPF media controller (mpf-mc) architecture. Previous versions used a multimedia library called [Pygame](#), and starting in 0.30, we're removing Pygame and replacing it with a library called [Kivy](#).

(Check out the [release notes](#) for details about the other stuff that's new in MPF 0.30, as well as our [migration guide](#) if you're migrating from MPF 0.21 to 0.30.)

What's a multimedia library?

Python at its core is a console-based programming environment. In other words, there's nothing built-in to Python that handles graphics and sounds. Pretty much all it can do is write text to the console and beep. So in order for a Python app to control graphical windows and audio, you have write your Python app so that it can talk to the computer's graphics and sound subsystem.

Doing this directly would be a huge pain, because the way that graphics and sounds work in Windows is different than Mac which is different than Linux. To address this, there are cross-platform multimedia libraries that expose OS-specific multimedia commands in a generic way.

For example, these cross-platform libraries let a programmer write commands like "open a graphical window that is 800x600 pixels", and then the cross-platform library on Windows will translate that to the DirectX calls to make that happen, on Mac it will translate it to the Core Graphics commands, etc. The same is true for everything graphics- and sound-related. We can simply say, "put this image in the window," or "put this text here," or "play this sound," and it's done.

So that's what we used Pygame for up until this point.

Why change?

Pygame has served us well for MPF versions 0.9 - 0.21. But there are many limitations in Pygame that, when taken together, caused us to want to find a different multimedia library to use.

The limitations in Pygame that have been holding back MPF include:

- Pygame is not maintained anymore. The last release was in 2009, which means it only works with the versions of Python that were current in 2009.

- Bugs that have been found since 2009 have not been fixed. (For example, the Mac version of Pygame has a memory leak which is why we can't support MPF on a Mac.)
- The audio system in Pygame is very basic. It only allows 2-channels (no surround sound), 8 simultaneous voices, and a single music track.
- Pygame does all of its graphical processing in the CPU, meaning it does not leverage the GPUs in modern computers. This leads to both lower performance and higher CPU usage. (Things that a GPU can do almost instantly take a lot of processing power if they're done on a CPU.)
- Pygame only supports video if it's encoded into MPEG-1 format with very obscure settings which does not lead to high quality videos. (And Pygame on the Mac doesn't support video at all!)
- Pygame cannot precisely control where the graphical window pops up on the screen, which is a problem for machines that have an LCD in the backbox that is partially obscured.

Introducing Kivy

After looking at many different options, we've decided to use a Python library called [Kivy](#). Kivy is modern, well-supported, and actively maintained. It's licensed with the MIT license (which is what MPF uses). Kivy leverages OpenGL, SDL2, and the GPU (if a system has it) and is easy to use for us as developers.

Kivy is also designed to be a complete application framework (rather than just a multimedia library), which means that Kivy has a lot of functions built-in that we had to build from scratch in the previous version of our MPF media controller.

(As a side note, remember that MPF's architecture separates the MPF core engine from the media controller. So everything we're talking about here with Pygame and Kivy only applies to the media controller. The MPF core engine remains the same.)

The MPF Kivy-based Media Controller architecture & concepts

Much of the general concepts of the media controller do not change in MPF 0.30. However we are updating a few terms to make them more logical.

A **display** is (usually) a physical thing that shows content, like an on screen window, a DMD, or an RGB LED color DMD.

Each display has a **slide frame** which is responsible for showing slides and running transitions between slides. (A transition is just an animated effect as the display flips from one slide to another, like push, move, slide, fade, flip, etc.)

The main concept in the new media controller is the **slide**. A slide is like a container which holds widgets.

Widgets are the individual elements you put on the slide. (These were called "display elements" in previous versions of MPF.) There are different types of widgets for different types of content: text, image, video, shape, etc. You can **animate** widgets over time by changing their properties to make them change color, size, rotate, etc. Any mode can put any widget on any slide. (Or you can put a widget on top of the current slide, for example if you want the credits mode to show a quick pop up message that plays an animation when a coin is inserted on top of whatever slide is current.) You don't have to create a new slide for a mode, but you have the option to if you want. When a mode stops, its widgets and slides will be removed.

The following diagram shows how it all fits together. Most of the time in your game, you'll be focused on slides and widgets. Widgets and slides are all tied back to modes in MPF.

One of the new features in the new mpf-mc is that you can add "sub slides" to a slide. You do this by adding a second slide frame as a widget (which you can size and position however you want), and then that sub-slide frame can manage its own slides and transitions, each with its own widgets. This is how you would do something like the "4-up" screen in *The Wizard of Oz* or how you'd let a game mode "own" a section of the slide. (In the WOZ example you'd have 4 slide frame widgets, each controlling its little portion of the screen.)

You can also add a widget to a slide which shows the content from another display. For example, this is how you can add a "virtual DMD" to an on screen window.

Here's the overall architectural diagram:

Changes to terminology with Kivy

As part of this move to Kivy, we're going to rename many of the concepts and components in the media controller so they match the names that Kivy calls things. The names we used previously were essentially made-up, and since people may need to search for how to do things in Kivy, it makes sense that MPF calls things by the same names that Kivy uses.

Old name	New name
Display	Display
Slide	Slide
none	Slide Frame
Display Element	Widget

Transition	Transition
Decoration	Animation
Movie	Video
Animation	Image (but an animated type, like GIF or zip file of image frames)

How display stuff was configured in the PRIOR version

We want to minimize the impact of changes to config files in MPF. That said, the new mpf-mc will address several shortcomings in the prior version, and in doing so, we have to make some changes.

As a reminder, here is how display content was controlled in the PRIOR version of MPF:

(1) Via `slide_player`: entries, like this:

```
slide_player:
  some_event:
    - type: text
      text: foo
    - type: shape
      shape: line
    - type image
      image: hello
      (etc)
```

(2) Via `show_player`: entries, like this:

```
show_player:
  some_event:
    show_name:
      tps: 5
      repeat: true
      (etc)
```

And then show yaml files would include slide settings for each step in the show, like this:

```
- tocks: 1
  display:
    - type: text
      text: foo
    - type: shape
      shape: box
      (etc)
```

Limitations of the PRIOR version

In the past year, we've realized there were several limitations with the way MPF handled the display, including:

- Slides were "owned" by a mode. If you wanted to use the same slide in multiple modes, you had to copy+paste the same slide settings in multiple areas. This was annoying and meant that you had to change things in lots of places.
- There was no way for slides to be reused.
- The `slide_player` concept meant that events could only trigger complete slides. You couldn't just add a display element to an existing slide.
- It was just about impossible to make compound slides with various parts of the slide getting content from different modes.
- In general, it was all "slide-centric"

Changes for the NEW version

The big change is that you'll be able to define named slides like a pool of assets. So you end up with all these slides. You can define them in machine-wide config or mode configs.. whatever is easiest. Like this:

```
slides:
  slide_name_1:
    - type: text
      text: foo
    - type: image
      image: banana
  single_player_score:
    - type: text
      text: %score%
  (etc)
```

At this point you're just defining slides. They're not even tied to modes yet. It's just slide names and widgets (& positioning & layering).

Then you can show slides via events, like this:

```
slide_player:
  event_name:
    slide: screen_name_1
    transition:
      type: slide
      direction: right
      duration: 1s
```

This gives the advantage of being able to reuse slides from any mode.

Then you can also define widgets where you give them names. Again this can be in mode config for machine wide config:

```
widgets:
  some_widget_name:
```

```
- type: text
  text: foo
```

You can also define groups of widgets:

```
widgets:
  common_bottom:
    - type: text
      text: BALL %ball%
    - type: text
      text: PLAYER %num%
```

The idea there is you can apply those widgets to any slide based on any event, like this:

```
widget_player:
  some_event:
    - widget: common_bottom
      slide: some_slide_name
```

Or you could apply those widgets to whatever screen was current

```
widget_player:
  some_event:
    - widget: common_bottom
      slide: %current%
```

All of these (*slide_player:* and *widget_player:*) can have additional settings for expiration time, layer, etc.

You have the option of defining widgets as either part of a slide definition (in the *slides:* section) or as standalone widgets (or groups of widgets) in the *widgets:* section. It really doesn't matter which. The only reason there are two options is if you want to define entire slides at once or more standalone widgets you can reuse and add to any slide. (And even if you define an entire screen, you still have the option to dynamically add widgets later.)

todo:

Adding sub-slide frames

Can you just add screen/widget definitions in *slide_player* and *widget_player* sections?

Unity Media Controller Architecture

Instead of using the Python-based MPF media controller to driver the graphics and sounds in your pinball machine, you can use a Unity-based media controller. This is very much a work-in-progress (as is MPF), but you can read more about and download the Unity Media Controller [here](#). (This project is called the "Unity BCP Server" since it talks to MPF via BCP.)

Backbox Control Protocol (BCP) 1.0 Specification (DRAFT)

This document describes the Backbox Control Protocol, (or "BCP"), a simple, fast protocol for communications between an implementation of a pinball game controller and a multimedia controller.

BCP transmits semantically relevant information and attempts to isolate specific behaviors and identifiers on both sides. i.e., the pin controller is responsible for telling the media controller "start multiball mode" and it is not directly concerned with how that happens; it's "fire and forget." Configuration or implementation in the media controller knows how to handle that mode, but it doesn't necessarily know what's going on inside the pin controller as it happens.

The protocol is versioned to prevent conflicts. Future versions of the Backbox protocol should be designed to be backward compatible to every degree possible.

The reference implementation uses a raw TCP socket for communication. On localhost the latency is usually sub-millisecond and on LANs it is under 10 milliseconds. That means that the effect of messages is generally under 1/100th of a second, which should be considered instantaneous from the perspective of human perception.

It is important to note that this document specifies the details of the protocol itself, not necessarily the behaviors of any specific implementations it connects. Thus, there won't be details about fonts or sounds or images or videos or shaders here; those are up to specific implementation being driven.

N.B. Since the pin controller and media controller are both state machines synchronized through the use of commands, it is possible for the programmer to inadvertently set up infinite loops. These can be halted with the "reset" command or "hello" described below.

Protocol Format

- Commands are human-readable text in a format similar to URLs, e.g. `command?parameter1=value¶meter2=value`
- Commands characters are encoded with the utf-8 character encoding. This allows ad-hoc text for languages that use characters past ASCII-7 bit, such as Japanese Kanji.
- Commands and parameter names are whitespace-trimmed on both ends by the recipient
- Commands are case-insensitive

- Parameters are optional. If present, a question mark separates the command from its parameters
- Parameters are in the format name=value
- Parameter names are case-insensitive
- Parameter values are case-sensitive
- Parameters are separated by an ampersand
- Parameter names and their values are escaped using percent encoding as necessary; see <https://en.wikipedia.org/wiki/Percent-encoding>
- Commands are terminated by a line feed character (\n). Carriage return characters (\r) should be tolerated but are not significant.
- A blank line (no command) is ignored
- Commands beginning with a hash character (#) are ignored
- If a command passes unknown parameters, the recipient should ignore them.
- To accommodate Backbox's asynchronous nature, commands may include an optional 'id' parameter. This allows subsequent responses to be tied back to a specific command. Most situations do not warrant this level of tracking, but it may be important in some scenarios, e.g. the pinball controller requesting a 'wave_your_hands_in_the_air' show, but no such show exists. The value of an id may be any string up to 32 characters. When the id parameter is not present in a command, that command's value is used for any response id (just the command itself, not the parameters). Any subsequent response from commands such as a show ending or triggers should send the corresponding id back that the show was started with.
- The pinball controller and the media controller must be resilient to network problems; if a connection is lost, it can simply re-open it to resume operation. There is no requirement to buffer unsendable commands to transmit on reconnection.
- The pinball controller is responsible for initiating the connection to the media controller, never the other way around.
- Once initial handshaking has completed on the first connection, subsequent re-connects do not have to handshake again.
- An unrecognized command results in an error response with the message "unknown command"

In all commands referenced below, the \n terminator is implicit. Some characters in parameters such as spaces would really be encoded as %20 in operation, but are left unencoded here for clarity.

Initial Handshake

When a connection is initially established, the pinball controller transmits the following command:

```
hello?version=1.0
```

...where *1.0* is the version of the Backbox protocol it wants to speak.

The media controller may reply with one of two responses:

```
hello?version=1.0
```

...indicating that it can speak the protocol version named, and reporting the version it speaks, or

```
error?message=unknown protocol version
```

...indicating that it cannot. How the pin controller handles this situation is implementation-dependent.

Commands

The following BCP commands have been defined (and implemented) in MPF:

- ball_end
- ball_start
- config
- dmd_frame
- error
- external_show_frame
- external_show_start
- external_show_stop
- get
- goodbye
- hello
- mode_start
- mode_stop
- player_added

- player_score
- player_turn_start
- player_variable
- set
- shot
- switch
- timer
- trigger

Here are the details for each:

ball_end

Parameters: None

Origin: Pin controller

Response: None

The ball has ended.

TBD does this post before or after the bonus?

ball_start

Parameters: player_num, ball

Origin: Pin controller

Response: None

Indicates that a ball has started. It passes the player number ("1", "2", etc.) and the ball number as parameters. This command will be sent every time a ball starts, even if the same player is shooting again after an extra ball.

config

Parameters: variable1=value1&variable2=value2&etc=etc

Origin: Pin controller or media controller

Response: None

Config is effectively the same as “set”, with the additional expectation that the value will be stored to disk so as to be available at next start or reset.

A game may use any set of config variables it wants, but here are some examples of what they could be:

Name	Description
credits	Set the number of credits in the system. This would be a decimal number such as 1 or 1.3, or it might be “free_play”
custom_message	Set the custom system message. Newline values must be percent-encoded.
language	This allows the pin controller to request a specific flavor of the presentation.
grand_champ	Set info about the grand champion. Value format would be initials,score. Initials may not contain commas.
high_score_N	Set info about the one of the high scores. Value format would be initials,score. Initials may not contain commas.
rating	This allows the pin controller to specify a “movie rating” for the machine. An example would be controlling “pg” versus “r” ratings for games such as Sopranos, which can include risqué language, sounds, images, etc.
volume_master	Set the master audio volume.
volume_sfx	Set the volume of the sfx track.

dmd_frame

Parameters: data (*see note)

Origin: Media controller

Response: None

Used by the media controller to send a DMD frame to the pin controller which the pin controller displays on the physical DMD. Note that this command does not use named parameters, rather, the data is sent after the command, like this:

```
dmd_frame?<raw byte string>
```

This command is a special one in that it's sent with ASCII encoding instead of UTF-8.

The data is a raw byte string that is exactly 4096 bytes. (1 bytes per pixel, 128x32 DMD resolution = 4096 pixels.) The 4 low bits of each byte are the intensity (0-15), and the 4 high bits are ignored.

error

Parameters: message

Origin: Pin controller or media controller

Response: None

This is a command used to convey error messages back to the origin of a command.

Parameter options:

```
message=invalid command&command=<command that was invalid>
```

external_show_frame

Parameters: name, led_data, light_data, flasher_data, gi_data

Origin: Media controller

Response: None

Sends updated device values (LED colors, light intensities, flasher pulse times, GI brightness) for an externally-controlled [hardware show](#) that is currently running. All of the data parameters are optional, but at least one must be included in each **external_show_frame** command.

Parameter	Description
name	The name of the external show (must be currently running).
led_data	A concatenated list of hex RGB color values that correspond to the list of LED names in the <i>leds</i> parameter when the external show was started (ex: led_data=0000009999990000FF).
light_data	A concatenated list of hex brightness values (00 to FF) that correspond to the list of light names in the <i>lights</i> parameter when the external show was started (ex: light_data=0099FF).
flasher_data	A concatenated list of values (0 = off, 1 = flash) that correspond to the list of flasher names that in the <i>flashers</i> parameter when the external show was started (ex: 0010011).
gi_data	A concatenated list of hex brightness values (00 to FF) that correspond to the list of GI names in the <i>gis</i> parameter when the external show was started (ex: 0099FF).

external_show_start

Parameters: name, priority, leds, lights, flashers, gis

Origin: Media controller

Response: None

Starts an externally-controlled [hardware show](#) (including LEDs, lights, flashers, and/or GI effects) with an associated show name and priority. Externally-controlled shows provide real-time device control via **external_show_frame** BCP commands. All devices that will be managed by the show must be included in the device list parameters (leds, lights, flashers, gis). The order in which the devices are listed in the device list parameters is important as all subsequent device data value updates will correspond to the order established in the show start command. The device list parameters are optional, but at least one must be included in order to start a valid hardware show.

Parameter Description

name	The name of the external show
priority	The priority of the external show relative to all other hardware shows in the pin controller.
leds	A comma-separated list of LED device names to include in this show.
lights	A comma-separated list of light device names to include in this show.
flashers	A comma-separated list of flasher device names to include in this show.
gis	A comma-separated list of GI device names to include in this show.

external_show_stop

Parameters: name

Origin: Media controller

Response: None

Stops an externally-controlled [hardware show](#) that is currently running.

get

Parameters: names=variable1,variable2,...,variableN

Origin: Pin controller or media controller

Response: set

Asks the other side to send the value of one or more variables. Variable names are to be stripped of leading and trailing spaces and lower-cased. The other side responds with a "set" command. If an unknown variable is requested, its value is returned as an empty string. For sanity reasons, all variable are to be lower case, must start with a letter, and may contain only lowercase letters, numbers, and underscores. Variable names should be lowercased on arrival. Variable names can be no more than 32 characters.

See "set" for a list of common variables.

goodbye

Parameters: None

Origin: Pin controller or media controller

Response: None

Lets one side tell the other than it's shutting down.

hello

Parameters: version=xxx

Origin: Pin controller or media controller

Response: See below

This is the initial handshake command upon first connection as described above. It sends the protocol version that the origin controller speaks. When received by the media controller, this command automatically triggers a hard "reset", described below.

If the pin controller is sending this command, the media controller will respond with either its own "hello" command, or the error "unknown protocol version." The pin controller should never respond to this command when it receives it from the media controller; that would trigger an infinite loop.

mode_start

Parameters: name, priority

Origin: Pin controller

Response: None

A game mode has just started. The mode is passed via the name parameter, and the mode's priority is passed as an integer via the priority. For example:

```
`mode_start?name=base&priority=100`.
```

mode_stop

Parameters: name

Origin: Pin controller

Response: None

The mode as stopped.

player_added

Parameters: player_num

Origin: Pin controller

Response: None

A player has just been added, with the player number passed via the *player_num* parameter. Typically these commands only occur during Ball 1.

player_score

Parameters: value, prev_value, change, player_num

Origin: Pin controller

Response: None

Sent to the media controller any time the player's score changes. It's possible that these events will come in rapid succession. Also note the parameter *player_num* indicates which player this score is for (starting with 1 for the first player). While it's usually the case that the *player_score* command will be sent for the player whose turn it is, that's not always the case. (For example, when a second player is added during the first player's ball, the second player's score will be initialized at 0 and a *player_score* event for player 2 will be sent even though player 1 is up.

player_turn_start

Parameters: player_num

Origin: Pin controller

Response: None

A new player's turn has begun. If a player has an extra ball, this command will *not* be sent between balls. (However a new *ball_start* command will be sent when the same player's additional balls start.

player_variable

Parameters: name, value, prev_value, change, player_num

Origin: Pin controller

Response: None

This is a generic "catch all" which sends player-specific variables to the media controller any time they change. Since the pin controller will most likely track hundreds of variables per player (with many being internal things that the media controller doesn't care about), it's recommended that the pin controller has a way to filter which player variables are sent to the media controller.

Also note the parameter *player_num* indicates which player this variable is for (starting with 1 for the first player). While it's usually the case that the *player_variable* command will be sent for the player whose turn it is, that's not always the case. (For example, when a second player is added during the first player's ball, the second player's default variables will be initialized at 0 and a *player_variable* event for player 2 will be sent even though player 1 is up.

set

Parameters: variable1=value1&variable2=value2&etc=etc

Origin: Pin controller or media controller

Response: None

Tells the other side to set the value of one or more variables. For sanity reasons, all variable are to be lower case, must start with a letter, and may contain only lower case letters, numbers, and underscores. Variable names should be lowercased on arrival. Variable names can be no more than 32 characters. Variable values are of unbounded length. A value can be blank.

Setting a variable should have an immediate effect. For example if the system audio volume is set, it is expected that audio will immediate take on that volume value. Or if the high score is currently being displayed and its variable it set, it should immediately update the display.

shot

Parameters: name, profile, state

Origin: Pin controller

Response: None

Indicates that a shot was just hit. Parameter *name* is the name of the shot, *profile* is the name of the shot profile that was active when it was hit, and *state* is the name of the state that the profile was at when it was hit.

switch

Parameters: name, state

Origin: Pin controller or media controller

Response: None

Indicates that the other side should process the changed state of a switch. Two parameters are required, name which is the name of the switch, and state which is "1" for active and "0" for inactive.

When sent from the media controller to the pin controller, this is typically used to implement a virtual keyboard interface via the media controller (where the player can activate pinball machine switches via keyboard keys for testing). For example, for the media controller to tell the pin controller that the player just pushed the start button, the command would be:

```
switch?name=start&state=1 followed very quickly by switch?name=start&state=0.
```

When sent from the pin controller to the media controller, this is used to send switch inputs to things like video modes, high score name entry, and service menu navigation. Note that the pin controller should not send the state of every switch change at all times, as the media controller doesn't need it and that would add lots of unnecessary commands. Instead the pin controller should only send switches based on some mode of operation that needs them. (For example, when the video mode starts, the pin controller would start sending the switch states of the flipper buttons, and when the video mode ends, it would stop.)

timer

Parameters: name, action, ticks

Origin: Pin controller

Response: Varies

This command allows the pin controller to notify the media controller about timer action that needs to be communicated to the player. There are many timers in MPF (configured via the [Timers: section](#) of a config file). You can enable a timer to send its details to the media controller by adding a `bcpl: yes` setting to a timer's settings.

trigger

Parameters: name

Origin: Pin controller or media controller

Response: Varies

This command allows the one side to trigger the other side to do something. For example, the pin controller might send trigger commands to tell the media controller to start shows, play sound effects, or update the display. The media controller might send a trigger to the pin controller to flash the strobes at the down beat of a music track or to pulse the knocker in concert with a replay show.

BCP Command Flow (Reference Order)

If you want to get an idea of the order events are sent in (and the exact types of events), the easiest way to do that is to run MPF's sample game Demo Man with verbose logging enabled. Then open the MPF log file (not the MC one) and filter it based on "bcplclient" and you'll see all the BCP commands that are sent from MPF's BCP client.

Demo Man includes a config file called "autorun" which uses the switch player plugin to automatically play through a game, so you can run that to generate a full log file that includes lots of different thing happening, like this:

```
mpf demo_man -v -c autorun
```

Here's the high-level order the BCP commands will be sent from MPF. This process starts with MPF boot and then follows through a game starting. The . . . sections are where stuff has been left out, but you get the idea. Note that in many cases (such as this one), the game mode actually begins before the attract mode ends. These two events were sent 7ms apart, so it's quick, but just FYI to be ready for the game to start whenever the attract mode is running.

```
hello?version=1.0
reset
mode_start?priority=10&name=attract
...
mode_start?priority=20&name=game
mode_stop?name=attract
player_added?number=1
```

```

player_turn_start?player=1
ball_start?player=1&ball=1
...
ball_end

```

Credits

The Backbox Control Protocol is being developed by:

- Quinn Capen
- Kevin Kelm (responsible for the initial concept and first draft of the specification)
- Gabe Knuth
- Brian Madden
- Mike O'Rourke

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* (located in the `/mpf/system/ball_controller.py` module) which manages everything.
- Individual *Ball Devices* (troughs, locks, etc.) which track how many balls they're currently holding, request new balls, eject balls, etc.
- The *playfield* device which is a special type of ball device that represents 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.

'Playfield' balls versus 'balls in play'

One important concept for ball tracking to understand is that there's a difference between "playfield" balls and "balls in play."

Playfield balls are any balls that are loose on the playfield, while *balls in play* is an in-game concept which represents how many balls a player has in play at any moment. In most cases, the number of playfield balls and balls in play will be the same, but not always. 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, so the ball controller still has to track them and wait for them all to drain. (Why? Well, the machine has to wait for all the balls on the tilted playfield to drain before it moves on to the next ball, and ball search has to keep running in case a ball gets stuck on its way to drain.)

Actually there are many scenarios where you have more playfield balls than live balls. If the player shoots the ball into a lock, at that moment you won't actually have any playfield balls but that player still has a ball in play.

By the way, the ball controller has nothing to do with the number of balls in play. It literally doesn't know what that concept is. All the ball controller cares about is tracking where all the balls are at all times. Whether a live ball is actually in play or not is the responsibility of the game logic—not the ball controller.

Lighting & Display Shows

MPF uses the concept of “shows” to control a sequence of actions that take place. Since MPF is actually two separate processes (the core game engine and the media controller), there are actually two different kinds of shows that run in each process.

- Hardware Shows are sequences of hard actions, including lights, LEDs, flashers, drivers, and/or GI effects.
- Media Shows are sequences of slides, videos, movies, images, and/or sounds.

The general concepts and file formats are similar for both kinds of shows.

For example, hardware shows are sequences of lights, LEDs, coils, flashers, and GI actions.

Shows can be long or short, simple or complex. You can have multiple shows running at the same time, and you can start and stop individual shows while others are running. You can create playlists of shows so that multiple shows are linked together to play back-to-back.

Hardware Shows

A hardware show is a show that takes place using pinball machine hardware, including lights, LEDs, drivers (coils), GI, and/or flashers.

You'll ultimately have dozens (or maybe even hundreds) of shows in your machine. Some of them might be simple, like when you complete a final shot in a shot group then all of the group's lights flash in sequence. Other shows will be complex, like when the player hits wizard mode and everything is rocking out like crazy.

The key to remember with shows is that you can have multiple shows playing at once. So something that might seem super complex at first could actually be ten little shows all running at the same time rather than one huge show. (This is also nice because then all your

little shows become sort of like "building blocks" you can combine in different ways for future shows.)

For example, the attract mode in a lot of machines might look really complex at first, but if you break it down, it's not too bad. For example, a typically attract mode might be built of up lots of individual shows, such as:

- A bunch of lights in a row cycle through in a pattern. That's one show.
- A different group of lights in your lower lanes might pulse back and forth in their own pattern. That's another show.
- A "wheel" of lights in the center of the playfield could rotate in their own show.
- The DMD might display a series of text frames and animations (high scores, push start, logo of the machine and your company, etc.) That could be its own show.
- You might have some sound effects that play during the attract mode every so often. Those could be their own show with two steps that repeat forever. (Step 1. Wait ten minutes. Step 2. Play sound. Repeat.)

You create show files in a YAML format (similar to your machine configuration files) which you then put in a "shows" folder inside your machine's folder. Then when you run MPF, all those shows are loaded so you can play them at any time.

Features of shows include:

- A show controller handles playing, pausing, stopping, and resetting shows. One show controller for hardware shows runs in the MPF game engine, and another show controller for media shows runs in the MPF media player.
- Add matrix lights, RGB LEDs, display information, sounds, events, and coil actions to each step of your show.
- Play multiple shows at the same time. Play/pause/stop/restart individual ones, etc.
- Play show files at any speed, including changing the playback speed of a show on the fly while it's playing. This means if you have a "wheel" of lights that you want to rotate faster and faster, you only have to make one show yaml file and then you can keep on changing the playback rate.
- Set repeats, like whether the show should play forever, or whether it repeats a fixed number of times and then stops.
- Configure playlists (lists of shows). Each step in the playlist can be made up of one or more shows, and you have precise control over how it moves to the next step. (After x seconds, after one of the shows plays through x times, etc.)
- Set whether a show "blends" with whatever's running below it. So imagine you have an light that is on solid red, and then you play a show at a higher priority that flashes that same light between blue and off. If you have blend enabled, then when the light in the higher priority show is off, whatever the light was set to in the lower

priority show will "show through." (So you'd have the effect of the light alternating between red and blue.) If you have blend to False, then when the higher priority show turns that light off, it will be "off." In that case the light would flash between red and off.) If you're fading an light between off and a color, and you have enabled blending, then the fade will blend between whatever color is below it. It's very cool!

- Specify the relative priority of a show, so if there are ever two running shows that want to set the same light at the same time, the higher priority one will win. (The priorities only affect lights and LEDs, not show events or coils.)
- Specify what happens when a show ends. Do the lights stay on in their last setting? Are they reset?

So now let's look at how you actually create a show file.

Show Settings

This page discusses all the settings and options you can use when playing shows. (If you're looking for details about how to actually create shows, look [here](#).)

All of the settings and options discussed here are available to you regardless of the technique you use to play or stop the show, whether it's via the machine configuration file, via the action event, or manually in code. (Details of the three of these options are [here](#).)

Options for Playing shows

When you play a show, you can use the following options. All of them are optional (i.e. the show player will use the default option if you don't specifically set it to be something else).

tocks_per_sec

Integer of how fast your show runs. ("Playback speed," in other words.) Your show files specify action times in terms of 'tocks', like "make this light red for 3 tocks, then off for 4 tocks, then a different light on for 6 tocks." When you play a show, you specify how many tocks per second you want it to play. Default is 30, but you might even want tocks_per_sec of only 1 or 2 if your show doesn't need to move than fast.

Note this does not affect fade rates. So you can have tocks_per_sec of 1 but still have lights fade on and off at whatever rate you want. Also the term "tocks" was chosen so as not to confuse it with "ticks" which is used by the machine run loop.

priority

Integer value of the relative priority of this show. If there's ever a situation where multiple shows want to control the same item, the one with the higher priority will win. ("Higher" means a bigger number, so a show with priority 2 will override a priority 1.) Default is 0. You don't have to worry about this unless you have overlapping shows playing at once.

repeat

True/False as to whether the show repeats (i.e. 'loops') when it's done playing. Default is False.

num_repeats

Integer of how many times you want this show to repeat before stopping. A value of 0 means that it repeats indefinitely. Note this only works if you also have `repeat=True`. Default is 0 which means if you set repeat=True then this show will play indefinitely (until you stop it).

blend

True/False which controls whether this show "blends" its lights with lower priority shows and scripts. For example, if this show turns a light off, but a lower priority show has that light set to blue, if you have blend=True then the light will "show through" as blue while it's off here. If you don't want that behavior, set blend to be False. Then off here will be off for sure (unless there's a higher priority show or command that turns the light on). Default is False.

hold

True/False which controls whether the lights or LEDs remain in their final show state when the show ends.

start_location

Integer of which position in the show file the show should start in. Usually this is 0 but you can specify a start point of whatever you want to start part way through a show. This is also used for restarting shows that you paused. The default is "None" which means the show will start playing wherever it last ended (or at the beginning if it hasn't been played yet). Use a value of 0 if you want to force the show to start playing at the beginning.

callback

A callback function that is invoked when the show is stopped. This setting is not available via shows that you play via the `ShowPlayer:` config file settings, but you can use this for shows you play with action events and manually via code.

Options for Stopping Shows

The options available when you stop shows are pretty simple:

reset

True/False which controls whether the show is reset back to the beginning when it stops. If False, you can play the show again and it will pick up where it left off (unless you override it by specifying a `start_location` when you play it). Default is True.

hold

True/False which controls whether the lights (matrix lights and LEDs) in this show stay on or not when the show ends. If True, any lights that this show turned on in the last step will stay on. If False, when the show ends, it will release control of the lights and they can be reset to whatever other state they should be via other shows or sections of the game.

Light Show file format

MPF light shows are created with [yaml files](#) which contain instructions for what what different lights or LEDs should do at different times. (Note that light shows today only support matrix lights and LEDs. In the future we'll add support for flashers and GI strings.)

You can also add event postings and coil firing to light shows.

Details of the light show YAML file

A light show file is a sequential listing of what things you want to do at various time points. Each instruction in a show file plays for a number of "tocks." So roughly speaking, your show file says, "Make this light red for 2 tocks, then off for 3 tocks, then green for 8 tocks, then post event *foo*, then make this light blue, then fire this coil then play this sound . . ."

The term "tocks" was used because we didn't want to use "ticks" because in MPF, the term "ticks" is used to represent each "tick" of the machine loop. So we choose "tocks" instead. We also chose "tocks" instead of "seconds" or "milliseconds" because when you create a show.yaml file, the show file has no sense of real world time. (So your show files don't say,

"do this for 100ms, then do this for 100ms," rather, they say "do this for 1 tock, then do this for 1 tock, then do this for 3 tocks, etc.")

It's not until you play your show that you specify your "playback" speed in terms of "tocks per second." This gives you the flexibility to play back a show at any speed and to change the speed of playback while a show is playing. (And if you want to use tocks as ms, that's fine—just always play back your shows at 1000 tocks per second and you're all set.) The key is that when you're creating a show, what's important is that the ratio of tocks between your steps is what you want them to be.

Let's look at a simple show yaml file. This file "sweeps" left-to-right-to-left between several RGB LEDs on the playfield, turning each one on red (ff0000), then off.

```
- tocks: 1
  leds:
    left_lane: ff0000
    center_lane: 000000
- tocks: 1
  leds:
    center_lane: ff0000
    left_lane: 000000
- tocks: 1
  leds:
    right_lane: ff0000
    center_lane: 000000
- tocks: 1
  leds:
    center_lane: ff0000
    right_lane: 000000
```

The line that begins with `- tocks:` is the beginning of each step in the show. The show controller stays on that step for the number of tocks specified (1, in this example), before moving to the next step. Again, how long a tock is doesn't matter here (because you specify how many tocks per second you want at runtime when you play the show). In this case because each step is the same duration as all the other steps, each step will always play for the same amount of time.

Another important thing to note is that each step represents the instructions that are actually sent to the hardware during that step. So in this above example, if we only want one light on at a time, that's we're using "000000" to turn "off" the light that was turned on red in the prior step. If we didn't have all those "000000" entries then all the lights would end up staying on.

Also note in the above example that we disable the `center_lane` light in the first step. We do that so that this show is "loopable", since we want the `center_lane` to be turned off in the first step but it's enabled red in the final step.

A note about YAML formatting

MPF's show files are in the YAML file format. Since this is the same file format that's used for your machine config files, you're probably at least somewhat aware of it. You're also probably aware that YAML files can sometimes be finicky in terms of syntax, so make sure that you keep in mind the following:

- All indents *must* be done with spaces. No tabs.
- The start of each step is `- tocks: .` The dash must be left-justified with no spaces in front of it, then you need a space before the word "tocks". So it's `<dash><space><t><o><c><k><s><colon>`.
- The next line down (`leds: .`) in our example must have its first letter lined up with the first letter in the row above it. So you have to have two spaces, then the word "leds:".

If you run into any YAML formatting issues, you can always copy and paste the shows from the Judge Dredd sample game as a starting point.

Controlling different types of items in your show

MPF show files can currently control four different types of things at each step, including:

- Matrix lights
- RGB LEDs
- Flashers
- GI strings
- Coils
- Events

Let's look at each of these types of items one-by-one:

Matrix Lights

You control matrix lights via a `lights: .` entry in your show file. Under that you specify each light name as well as a 2-byte hex representation of its value.

```
- tocks: 1
  lights:
    l_pick_a_prize: ff
    l_extra_ball: ff
    l_right_start_feature: ff
- tocks: 1
  lights:
```

```
l_pick_a_prize: 0
l_extra_ball: 0
l_right_start_feature: 0
```

At this point the MPF does not support variable brightness for matrix lights, so the only meaningful values here are 0 for off, and anything non-zero for on.

You can also control matrix lights based on their "tag". Adding a tag (along with the tag name) to a step in the show will apply the setting from that step to all the matrix lights with that tag. You specify tags by adding an entry "tag|<tag_name:" into the step of the light show. (That's the text "tag", followed by the vertical bar character, followed by the tag name, with no spaces.) For example:

```
- tocks: 1
  lights:
    tag|some_tag: 0
```

When you enter a tag name into a light show, MPF internally expands it to all the lights that have that tag. So if you have previously turned on a light by name in a step, and then you later use a tag to turn off lights, then if that light has that tag it will also turn off even though it was turned on individually.

RGB LEDs

MPF lets you control RGB LEDs via the PD-LED board for P-ROC or the built-in RGB LED channels on the FAST controller board. (We'll add support for Fade Candy controllers too, but we haven't gotten to that yet.)

To add RGB LEDs to a show, add a `leds:` entry to each step of your show and specify your LEDs like this:

```
- tocks: 1
  leds:
    beacon: ff0000-f1 # hex color with fade extension
- tocks: 1
  leds:
    beacon: ffcc00 # hex color with no fade
- tocks: 1
  leds:
    beacon: 0, 255, 0, 2 # integer colors with fade
- tocks: 1
  leds:
    beacon: 0, 255, 255 # integer colors with no fade
```

If you just specify a color, the show controller will change that LED to that color instantly. You can also specify that the LED should fade from its current color to the new color over some period of time by adding a fade time to each entry. (This fade time is in tocks, with the fade

beginning where it's entered in your show file and ending after the number of tocks has expired.

Note that LEDs will always respect their own default fade rates and your global fade rate, so if you want to force an LED switch to that color instantly and ignore its default fade, specify a fade time of 0.

Finally, note from the example above that there are two alternate notations you can use to specify LED color and fade values. One is a hex string (with optional -f extension for fades), and the other is a list of integers (three items if you're just specifying the color in red, green, and blue, and four items if you want to specify a fade). It doesn't matter which notation you use, so go with whatever you're comfortable with.

You can also set colors of and fades of LEDs via tags. (See the note on using tags in the Matrix Lights section above.) For example:

```
- tocks: 1
  leds:
    tag|upper_playfield: 0
```

Flashers

You can add flashers to your light show by adding a `flashers:` section and then the name of each flasher. Flashers don't have any additional options since you can't specify their color or duration of flash:

```
- tocks: 1
  flashers:
    flasher1
    flasher2
```

GI Strings

You can control GI strings in your light shows by adding a `gis:` section to your show and then adding the name of each GI string as well as an intensity value from 0-255. For GI strings that are not dimmable, just use `ff` for on and `00` for off. For example:

```
- tocks: 1
  gis:
    gi_back_panel: ff
    gi_upper_right: ff
    gi_upper_left: ff
    gi_lower_right: ff
    gi_lower_left: ff
```

Coils

You can include coil pulses in your shows which means the show controller will pulse a the coil at that step of the show. Coils are specified like this:

```
- tocks: 1
  coils:
    c_knocker: pulse
```

Coil options are pretty simple. You add a `coils:` section to the step in your show file, then add one or more coils under it in the format `coil_name: pulse`. At this time MPF only supports pulsing coils, but we will add the other methods (enable, disable, pwm, etc.) soon.

Events

You can add events to your shows which cause the MPF to post events at the show step where they're included. For example:

```
- tocks: 1
  events:
    hello_world
    goodbye_guys
```

In this case, MPF will post two events, one called "hello_world" and one called "goodbye_guys" at this step in the show.

Putting it all together

Remember that you can combine matrix lights, LEDs, flasher, GIs, coils, and events into single shows (and even single steps within a show):

```
- tocks: 2
  leds:
    l_led0: ff0000
  lights:
    l_pick_a_prize: ff
    l_extra_ball: ff
    l_right_start_feature: ff
  flashers:
    upper_right
    upper_left
  gis:
    main_left: ff
- tocks: 1
  leds:
    led0:00ff00
  lights:
    l_pick_a_prize: 00
  events:
```

```
hello_event
coils:
  c_left_slingshot: pulse
```

Other important light concepts

There are a few other important concepts to understand that we haven't covered yet.

Colors and Color Lists

Throughout this documentation about lights, we've referenced colors as six-digit hex strings, such as "000000", "ffffff", or "ffaa00." (In most cases these would be 2-byte red-green-blue color values, though technically speaking they are the values that apply to the first, second, and element depend on your hardware and how the light objects are configured in your machine configuration file. So if you configure your light in a way that doesn't have the first output to the red element, the second to the green, and the third to the blue, then you would have GRB or RBG or whatever else you configured your file as. Probably easiest to fix your yaml file in that case. :)

MPF has intelligence to handle scenarios where the color length doesn't match the number of elements that make up a light. If you have a single-color light, then you can pass it a color of "ff" which the light driver will translate to 255 which will then be used to activate the light at full intensity. If you pass a color like "ff0000" to an light that only has a single element, then MPF will translate that color to [255, 0, 0], but since the light only has a single element it will only take the value from the first item in the dictionary and ignore the rest. Similarly if you send a color of "ff" to a three-element RGB LED, then MPF will translate that to a color dictionary of [255, 0, 0], meaning you will get the first element at full intensity and the second and third elements will be off. (In other words a color value of "ff" will enable an RGB LED to be red (assuming the order of the elements in your yaml file is red-green-blue.)

Again, none of this really matters if you're using entirely RGB LEDs. But what about single-element, single-color lights?

Understanding Single Color lights

Single color lights only have one element, and as far as MPF is concerned, they don't have a "color" at all. Single color lights simply use the 2-byte color value as their intensity. So it can be confusing if you have a single-color blue light, because to turn it on at full brightness you would pass it a color of "ff" (which is internally translated to "ff0000" and ultimately [255, 0, 0]. So it can be kind of weird to think, "Wait, I'm sending this light the color "red" but it's coming out "blue?" Again, that's because single-color light don't really have color as far as MPF is concerned. They just have a single intensity value and the color is determined by whatever color the plastic insert in your playfield is.

The bottom line is the word "color" is used throughout this documentation. If you have RGB LEDs, then "color" really is the color that the LED will be. But if you have single-color LEDs or lamps, you just use two bytes whenever color is used (color="ff", color="a0", etc.). "Color" in the case of single-element lights is really more like "brightness."

Using variables to 'save' colors

Remember in python that you can use variables anywhere. You might want to define a bunch of colors once in your game, like

```
self.red = ff0000
self.yellow = ffaa00
self.green = 00ff00
```

Doing this means that you can "tune" your colors in one central location (at least in terms of light commands and light scripts, and then whenever you need to specify a color, you can use for variable, like this:

```
`self.machine.light_controller.enable("LED1", priority=6, color=self.green`
```

This might also be nice for defining a "white" color with RGB LEDs where the operator can specify whether they want a "cool" or "warm" white setting. (Note at this time you can't use color definitions in light shows. We've added that to the todo list.)

Understanding the Concept of "Blends"

Throughout this documentation there have also been several references to "blends." Almost all of the methods that enable lights or play scripts of shows have a parameter "blend" which can either be True or False. The blend parameter is universally used to describe what happens when an light is "off." When blend=True, that means that whenever a light is off that any lower priority light shows, scripts, or commands that have enabled that light will "show through." To understand this, let's look at an example where you have a script running at priority 2 that flashes an RGB LED between red and off. You also have a lower-priority command (running at priority 1) that has the LED set to blue.

If your higher priority script has blend=True, then when it is off the lower LED setting will show through, so you will have the effect of the LED alternating between red and blue. If your higher priority script has blend=False, then it will "block" the lower priority setting and your LED will alternate between red and off.

Blends also affect what happens when an LED is fading between a color and off. For example, in the same example scenario from above, if your priority 2 script is slowly "pulsing" the LED between red and off and you have blend=True, then because it's running on top of a blue setting you'll get the effect of the LED fading between red and blue. Again in that scenario with blend=False, then the LED will fade between red and off.

Note that blends only affect what happens when an LED is either off, or fading to/from off to a color. There is no support for any concept of "alpha" blends that blend full colors. (Maybe that's a future feature?)

Also note that blends are intended to deal with lights receiving conflicting instructions at different priorities. If you simply want to flash an RGB LED between two colors, it would be much easier just to use a script that alternates between the two colors you want. No need to mess with blends in that case.

Configuring Max Brightness Settings

The MPF light controller supports the concept of a "maximum brightness" setting for each light in a machine. Max brightness is a floating-point multiplier value (between 0 and 1) that's applied to every command sent to the an LED. The default multiplier is 1.0, which means if you tell an LED output to turn on with the intensity 255, then it will get 255. You can set the max brightness value to be less than one to turn down the brightness.

For example, if you configure an LED to have a maximum brightness of 0.85, then sending an intensity command of 255 will actually be sent as 216.

You configure `max_brightness` as a dictionary of floating-point values which is maintained by each light device object in your game. For example, if you have an light defined as "shootAgain," you can configure its `max_brightness` via:

```
`self.game.leds.shootAgain.max_brightness = [0.85, 0.85, 0.85]`
```

The `max_brightness` value must always be a dictionary, even for single element LEDs (in which case you'd use ``max_brightness=[1.0]``).

Note that the `max_brightness` settings can be a bit weird for LEDs and incandescent lamps connected to traditional lamp matrixes since the matrix timing means we can only set 12 levels of brightness instead of 255 for direct-controlled LEDs.

As for what you can do with this setting, that's up to you as the programmer. You can configure different elements of the same LED to have different settings to compensate for elements that are out-of-whack. You can make system-wide brightness compensation settings accessible via the operator's menu. Maybe you have an ambient light sensor and adjust the LEDs based on how light or dark the room is. Really the sky's the limit. The important thing is that you change the `max_brightness` setting of any LED object at any time. (The change is instant.) And you can still continue to pass color commands to your LEDs without worrying about the brightness setting. (In other words you don't have to do any complex math just to turn down your LEDs.)

By the way, experiment with different default brightness settings. On our test machines we can't tell a difference between 0.85 and 1.0. So maybe you get longer life with lower settings? (Though we don't know. That might not be true at all.)

Global Fades & Defaults

The light items in MPF also support the concept of a "default_fade" value which is an integer which specifies the fade time (in ms) that should be used for each light when it is enabled or disabled instantly. You can access this setting in the same way you access the `brightness_compensation` setting. For example:

```
`self.game.leds.LED1.default_fade=50`
```

Using a `default_fade` is nice because it allows the LEDs to "feel" more like incandescent bulbs that fade on and off instead of harshly turning on and off. This is a matter of personal taste. Some people like the LED "look" and others hate it. In our game we wrote code which reads these default fade settings in via the yml file, and we've also written an operator menu setting that allows the operator to specify their own default fades to suit their individual tastes.

By default the `default_fade` is 0.

Default fades only apply to light instructions that instantly turn a light on or off. If you use a light command to fade an light on over 20ms, then it will fade on over 20ms regardless of what your `default_fade` is set to. We can imagine incorporating `default_fade` changes in your gameplay. (Perhaps you want a cool strobe effect where you quickly flash a bunch of lights with a fade of 0 even though your default is 20?)

Sync Locking

One of the cool features of this light controller is that it sets a "current time" variable once per update loop. This means if you have a large update—maybe you're stopping a bunch of shows and looping through a bunch of scripts—as long as they all happen in the same game loop then they will all have the same "start" time. Also when light shows and scripts are processed, the "next" action time is based on when the last action time should have happened. This means you can have multiple shows and scripts that all use the same timing, they will stay in sync, and if your game loop gets bogged down then the lights will remain in sync.

Playing & Stopping Shows

Once you've created your show files and entered them into your configuration file so MPF will load them, the next step is to actually play your shows.

Background on how MPF handles show playback

Before we get into the specifics of playing shows, let's look at some important concepts about how MPF handles playing shows.

You can play multiple shows at the same time

First, understand that it is possible to play multiple shows at the same time. This makes it easy to break down complex shows into a bunch of smaller, simpler shows which just play at the same time. For example, take a look at this Judge Dredd attract mode light show that has a lot going on:

https://www.youtube.com/watch?v=_C4ivRWPeMU

At first you might think this is crazily complex, but it's actually 7 simple shows which all happen to be playing at once. The back-and-forth "sweep" of the chain lights is one show, the "sweep" of the J-U-D-G-E drop target lights is another show, the rainbow cycling of the perp lights is a third show, the DMD display is its own show, etc. So just by making all these simple little shows and playing them all at once, we can make something that looks complex but that's really easy. (Plus if you break your shows into lots of little shows, you can use those smaller shows in other places throughout your game.)

When you play multiple shows at once, you can start and stop individual ones as you wish. So when the machine is in attract mode, you might have one show going in a loop for the display, and then you have a bunch of shows for some playfield lights, then after awhile you switch up the playfield to a different light show, but you can keep the display show running untouched.

Shows run at different "priorities" which is how they handle conflict

When you play a show, one of the options you can specify is the "priority" that show will play at. (If you don't specify a priority, the show will play at Priority 0.) This priority controls what happens if two different shows want to control the same thing at the same time. So if one show is running at Priority 0 and is flashing an LED red, and another show running at Priority 2 wants that LED to be blue, well then that LED will be blue. After the higher priority show doesn't need that device anymore, it will be restored to whatever any running lower priority show wants it to be.

For display shows, the running priority of a show affects what priority the slides are that it puts on the display. The display will only show the slides from the show if they're the same or higher priority than whatever slide it's showing at that moment.

You can't run a show faster than your machine loop rate

Note that since the show controller is completely controlled by your game code, you can't run a show faster than your [machine tick HZ rate](#). This doesn't mean that your `tocks_per_second` can't be faster than your game loop. It just means that if it is then show actions at each step will not actually be executed until the next game loop.

For example, if you have a machine loop rate of 30HZ, and you have shows playing at 1000 tocks_per_sec, that's fine as long as each step in your show is at least 33 (or so) tocks. So if you have steps that are playing 100ms apart, this will work fine since your 30HZ machine loop can easily handle a show which has 100ms between steps.

If this all sounds confusing, then just play it safe by not playing shows faster than your machine loop rate. :)

Playing and Stopping Shows: 3 different options

There are several ways to play (and stop) shows (once they've been loaded) in MPF, including:

- The ``ShowPlayer:`` section of the machine configuration file.
- Posting show start and stop action events.
- Manually in code.

Playing shows via the config file

You can make entries in your machine configuration files which cause shows to stop and start based on different [MPF events](#). For example, you can start playing the attract mode light shows and display show when the attract mode starting event is posted, and then when a the game starting event is posted, you can stop those shows and start a display show with includes the player's ball number and score.

Details for how to start and stop shows via the config file are available in the [`ShowPlayer:` section](#) of the configuration file reference. Note that all of the various show options mentioned there are covered in the [Show Settings page](#).

Posting show start and stop action events

You can also play and stop shows via the action events ``action_play_show`` and ``action_stop_show``. ([What are action events?](#)) You can pass any of the standard show settings (covered [here](#)) when starting or stopping a show via an action event. This is nice if you want to play or stop shows via [Logic Blocks](#).

Playing shows via code

You can also play and stop shows via code—whether it's code you add into a Scriptlet or if you're writing something custom.

Once you've loaded a show, you play it with the show's `play()` method, like this:

```
self.machine.shows['rainbow_show'].play(repeat=True, tocks_per_sec=4,
priority=3, blend=True)
```

You can stop a running show via its `stop()` method, like this:

```
self.machine.shows['rainbow_show'].stop()
```

Again, the specific keyword parameters you can use when playing and stopping shows is covered in the [Show Settings page](#), which we'll look at now.

Light Scripts

A show "script" lets you attach a simple script to a single light which does something more than just turning it on. A script is usually pretty simple, like "turn red for 500ms, then off for 500ms, then repeat." But you can also make more complex scripts with multiple steps, like cycling through all the colors of the rainbow. More features of scripts:

- Set as many steps as you want, including how long each step is and whether the light fades to that step or switches instantly.
- Assign variables to scripts for easy reuse. So you can have a script called `self.flash_red` which you can play at any time. You can specify which light that script is played for when you play it. So you can end up with a library of scripts you can reuse through your game.
- Use the `blend` setting to specify whether the light blends with lower priority stuff when it is off in the script.
- Specify how many times a script repeats, or whether it repeats forever.
- Set the priority of the script. This is the same priority system of the light shows which means it handles how multiple things setting this LED at the same time work.

From a technical standpoint, a light script is a Python list of dictionaries. Each list item is one step in the script, and each dictionary contains key/value pairs which specify the color and (if you want the script to fade to the next color) the fade time (in ms). The key with scripts is they are *not* tied to a specific light when you create them. (That happens when you run them.)

Creating and Playing Scripts

Creating a script is easy. Here's a script called "flash_red" that will flash an LED between red and off:

```
self.flash_red = []
self.flash_red.append({"color": "ff0000", "time": 100})
self.flash_red.append({"color": "000000", "time": 100})
```

To run the script, use the show controller's `run_script()` method, like this:

```
`self.machine.show_controller.run_script("LED1", self.flash_red, "4", blend=True)`
```

Most likely you would define your scripts once when the game loads and then call them as needed.

You can also make more complex scripts. For example, here's a script which smoothly cycles an RGB LED through all colors of the rainbow:

```
self.rainbow = []
self.rainbow.append({'color': 'ff0000', 'time': 400, 'fade': True})
self.rainbow.append({'color': 'ff7700', 'time': 400, 'fade': True})
self.rainbow.append({'color': 'ffcc00', 'time': 400, 'fade': True})
self.rainbow.append({'color': '00ff00', 'time': 400, 'fade': True})
self.rainbow.append({'color': '0000ff', 'time': 400, 'fade': True})
self.rainbow.append({'color': 'ff00ff', 'time': 400, 'fade': True})
```

Playing a script causes the show controller to dynamically create a light show which is played just like any other light show. If you want to save a reference to the light show that's created when a script is played, you can call it like this:

```
`self.blah = self.machine.show_controller.run_script("LED2", self.flash_red, "4")`
```

Then you can use that reference to stop the show, change the playback speed, change the priority, etc.

Stopping a Script

You can use show controller's `stop_script()` method to stop a script. In a practical sense there are several ways you can use this.

- Specify `lightname`` only to stop (and remove) all active light shows created from scripts for that light name, regardless of priority.
- Specify `priority`` only to stop (and remove) all active light shows based on scripts running at that priority for all lights.
- Specify `lightname`` and `priority`` to stop (and remove) all active light scripts for that light name at the specific priority you passed.
- Specify a `show`` object to stop and remove that specific show.

If you call `stop_script()` without passing it anything, it will remove all the light shows started from all scripts. This is useful for things like end of ball or tilt where you just want to kill everything.

Some examples:

To stop every running script, use the following: (Technically this stops all the shows that were started from scripts versus started from show yaml files)

```
`self.machine.show_controller.stop_script()`
```

To stop every script attached to the LED called "LED1":

```
`self.machine.show_controller.stop_script(lightname="LED1")`
```

To stop every script that's running at priority 10:

```
`self.machine.show_controller.stop_script(priority=10)`
```

Creating Playlists

The show controller has a Playlist class you can use to create playlists of shows that will play in order. Each step of the playlist can have one or more shows (that play at the same time), and you can specify how the playlist will move from one step to the next. To create a playlist, create an instance of the Playlist class, add shows to it, and configure the settings for each step. Here's an example:

First, make sure you've imported the Playlist class, like this:

```
`from mpf.system.show_controller import Playlist`
```

Then write some code like this:

```
self.my_playlist = Playlist(self.machine)
self.my_playlist.add_show(step_num=1, show=self.machine.shows['show1'],
tocks_per_sec=10)
self.my_playlist.add_show(step_num=2, show=self.machine.shows['show2'],
tocks_per_sec=5)
self.my_playlist.add_show(step_num=2, show=self.machine.shows['show1'],
tocks_per_sec=10)
self.my_playlist.add_show(step_num=3, show=self.machine.shows['show3'],
tocks_per_sec=32, blend=False)
self.my_playlist.step_settings(step=1, time=5)
self.my_playlist.step_settings(step=2, time=5)
self.my_playlist.step_settings(step=3, time=5)
```

The first line creates an instance of the Playlist class named "self.my_playlist".

The next 4 lines use the playlist's "add_show()" method to add shows to each step. Note that the show called "self.show1" has been added to both Step 1 and Step 2, meaning it will play for both steps.

The final three lines configure the settings for each step. In this case it's just that each step will play for 5 seconds before moving on. This is also where you could specify that you'd like

to move on based on a "trigger show" which has played a certain number of times before moving on. See the documentation in `show_controller.py` for details.

Once your playlist is all setup, you can start via: ``self.my_playlist.start(priority=100, repeat=True)`` The priority specified what priority all the shows in that playlist will play at, and ``repeat=True`` means that this playlist will loop. (Forever in this case. You can also specify a number of times to repeat before exiting.)

Stopping the playlist is easy too:

```
`self.my_playlist.stop()`
```

Displays & DMDs

Pretty much every pinball machine has some type of display, whether it's a set of 1980s-style 7-segment numeric displays, an early 90s-style alphanumeric display, a dot matrix displays (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 animations. 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, and you can configure alternate translations for text in multi-language environments. You can also apply decorations and transitions to your displays and their elements. And, like just everything else in MPF, you can do most of your display configuration via the machine configuration files.

Here are a few photos of the MPF Media Controller's display system in action. These were all created with the configuration files and without manual programming. The first photo shows a traditional DMD with a single text element. The second is on-screen window display made up of an image element (the background), a virtual DMD element, a drawing element (the thin white box around the DMD), and a text element (the "Judge Dredd" words in the lower right corner). The third image "color" DMD on an LCD monitor that you could install in your backbox, and the fourth image is a full-color RGB LED matrix (so it's like a color DMD, but a matrix of LEDs and not an LCD).





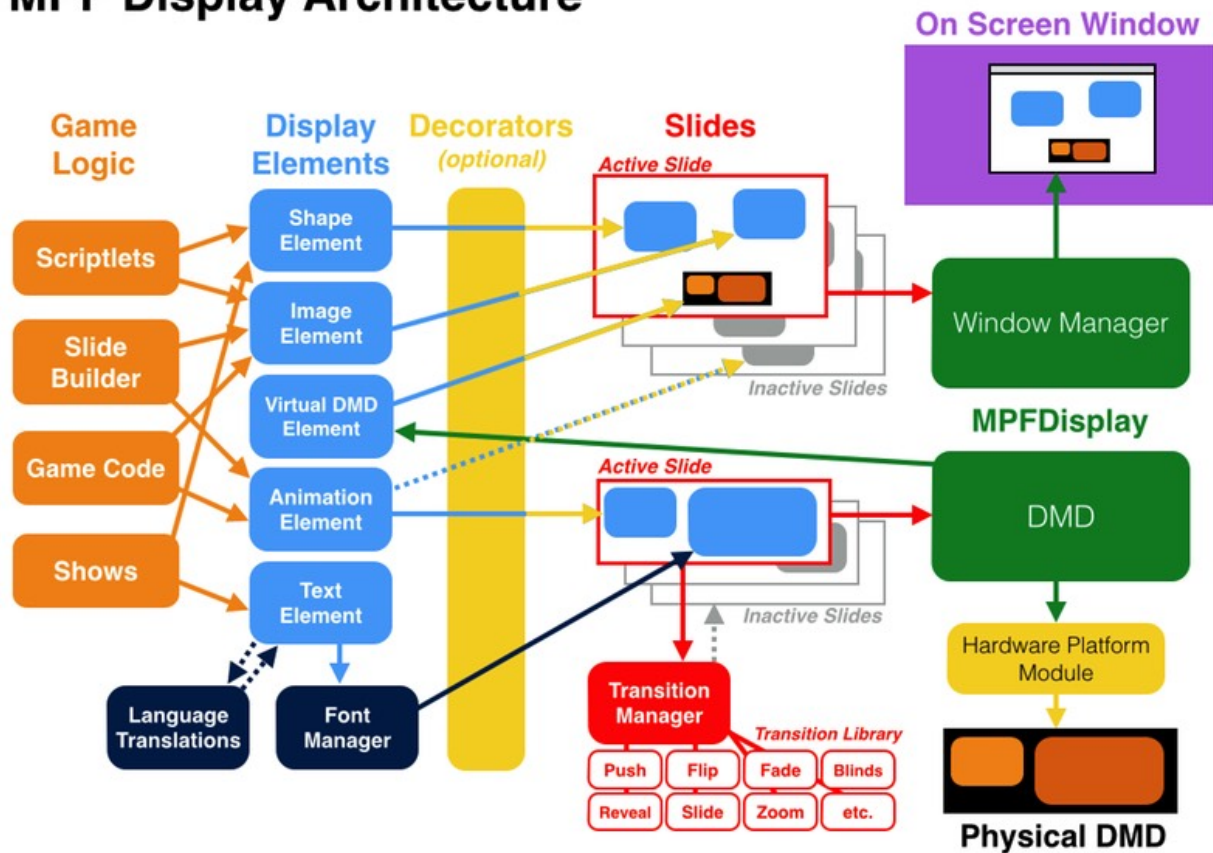
Before we go into the details of all the various display components, let's start with an overview of how the MPF display architecture works. (If you don't care about the details and just want to start using your display, you can jump directly into our [step-by-step tutorial](#) which covers how to get your display running.)

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 [SDL](#) 1.2 (which it interfaces via [Pygame](#)). Pygame & SDL are prerequisites for the MPF Media Controller, and it won't run without them.

We put together 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.

MPF Display Architecture



The major components of The MPF Media Controller's display system are:

- Your **game logic** which is responsible for generating the content that will be displayed. This could come from [settings in your machine configuration file](#), entries in show files, or game code you write manually.
- Every object you put on a display is called a **Display Element**. MPF supports several different types of display elements, each with their own settings and properties.
 - **Text elements** let you display text. You can pick the font, the color, etc. The actual text strings in Text elements can be run through the Language module before they're sent to the display so you can do on-the-fly text replacements. This is used for multi-language translation and for installing alternate (i.e. "family friendly") text strings.
 - **Image elements** let you show images.
 - **Animation elements** let you play animations (i.e. videos). You can start and stop them, specify whether they should repeat, specify the playback rate, etc.
 - **Shape elements** let you draw simple shapes onto the display. Boxes, lines, circles, etc.

- Every display element lets you [specify its position on the display](#), either via pixel-level accuracy, or with positional keywords like "top", "center", "left", etc. You can also specify the layer (z-order) of elements to control which ones are drawn on top of each other if they overlap, and you can control alpha transparencies which affect how they blend with elements below them.
- Next, you can apply [Decorators](#) to elements which can cause them to sparkle, blink, pulse, etc. It's really easy to write your own custom decorators too.
- You arrange all of your elements on a [Slide](#). A slide is the same size of the display, so arranging various display elements on a slide is how you arrange them on the display. Every display has an "active" slide which is the slide that's currently being shown. It can also have one or more inactive slides that are waiting in the background to be shown later.
- A [Transition Manager](#) lets you use cool effects to transition from one slide to another. MPF ships with several different types of transitions (push, fade, reveal, flip, etc.), and it's really easy to write your own transitions if you'd like to make custom ones.
- An [MPF Display](#) is a physical display in your game. For example, the DMD is an MPF display, as is the on-screen window. MPF actually supports multiple simultaneous displays (each with their own unique sets of slides), so you can have an on screen window and a physical DMD at the same time, and they don't have to show the same content, (though they can if you want). You can even use the contents of one display as an "display element" that's part of another display. (For example you can have the DMD display show up in a portion of your on screen window while still showing other display elements around it.) We current have MPF Displays for the on screen window and the DMD, and we'll add a segmented display soon.

All these concepts come from PowerPoint. :)

The creators of MPF (Brian Madden and Gabe Knuth) both have day jobs as IT industry analysts, and we both give dozens of presentations per year. In other words, we 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." Those elements can be text, images, animations/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 decoration effects (blink, sparkle, spangly, 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.

Displays

A "display" in MPF is a piece of physical hardware that shows slide. For example, the DMD is a display, the on screen window is a display, and (when we get to it), the segmented score displays in the backbox are displays.

MPF contains separate modules (called "display modules") for each type of display (since the code that drives each type of a display is very different). Even though they're all different, they all plug-in to MPF in the same way and they all work basically the same. (Yes, even segmented displays.)

Each different type of display has different properties, including size (length and width in pixels), how many colors it supports, its refresh rate (in frames per second), etc. MPF can support multiple different displays at the same time, each with their own unique settings. (You might want to update the DMD at 60fps but the on screen window at only 30fps.)

Each display is responsible for managing its own [slides](#) and for creating new slides when needed.

DMD

MPF supports dot matrix displays (DMD) devices. It can support physical devices plugged into the 14-pin DMD ribbon cable on a modern pinball controller, as well as on-screen "virtual" DMDs:



For physical DMDs (left photo above), MPF can support up to 16 shades of pixel brightness for traditional mono-color DMDs. (This can be both new LED-style and older gas discharge displays, and even old displays that were only only 2 or 4 shades can be driven at 16 shades by modern pinball hardware.) MPF can support refresh rates at the limit of the hardware, typically about 60 Hz.

MPF can also support newer full color RGB LED "matrix" displays that are arrays of RGB LEDs. (These are similar to the Color DMD product, except they're actual arrays of RGB LEDs and not an LCD screen.)

MPF can also drive on screen virtual DMDs (right photo above). You can configure the color of the dots, the shape, the spacing between pixels, etc. You can also specify the size of how big the DMD is rendered on screen, including rendering it in "life size" at high resolution (meaning you could put an LCD monitor in your backbox instead of a hardware DMD to save money. (\$70 for a 14" LCD versus \$400 for a DMD)

MPF can also support driving a physical DMD and on screen virtual DMD at the same time. (With the DMD content from the physical DMD automatically showing up in the virtual DMD.)

What about Color DMDs?

MPF can support both "real" and "LCD" color DMDs.

LCD color DMDs aren't physical DMDs per say, rather, they're LCD monitors in the back box which run software to make the displays look like they're low-resolution color DMDs. (So we use modern high-res displays to look like they're simple multi-color hardware DMDs.)

"Real" color DMDs are matrix arrays of RGB LEDs (often referred to as "Smart Matrix" displays) which you drive via separate hardware such as a Teensy with SmartMatrix shield.

In either case, you create your content for a low-res DMD which you then display on an LCD window or on the physical matrix hardware:



All DMD options let you adjust the pixel style and spacing, color palettes, and other characteristics of the display to get it dialed in exactly how you like it.

Configuring your DMD

You configure your DMD in the [dmd: section](#) of your machine configuration file. If you would also like to have a virtual DMD in your on screen window, you configure that as a [virtual_dmd display element](#).

Window Manager

The Mission Pinball Framework includes a Window Manager that is responsible for controlling the on screen window that pops up when MPF is running. Here's an example that's been configured with a background image, some title text, and an on-screen virtual DMD.



This on screen window is completely optional. If you're running a production pinball machine from a small single board computer, you probably just care about the DMD and wouldn't even bother plugging in a monitor or having an MPF window. On the other hand, if you're at home working on your machine, it's nice to have the window pop up which can show you what's on the DMD or other information about your machine. (The on screen window can show whatever you want, in addition to troubleshooting and game status information.)

MPF can support multiple displays at the same time. So it's absolutely possible to run both an on screen window display and a physical DMD at the same time.

Using the on screen window as your pinball machine's primary display

You can also put an LCD monitor in your pinball machine and use the LCD window as your primary display. There are several options for this, including:

- Run a full size "virtual" DMD which looks just like a traditional DMD except you can get an LCD monitor for \$70 instead of having to spend \$400 for a real DMD panel.

- Run a "color DMD" which is like a software DMD except the dots are different colors. (Or, put another way, it's like using a color monitor for your display but you make it look like it's running at a really low resolution and you have a filter to make the individual pixels look like dots.)
- Run a fully size high resolution display, like Jersey Jack does with the Wizard of Oz and The Hobbit.

In all three of these scenarios, if you're using your LCD display as your machine's primary display, then you'll probably run it full screen mode and spend some time getting the placement of all the display elements tuned perfectly. (You'll probably also use a "real" computer and not a low-power single board computer, because real LCD displays have 1-2 million pixels with 24-bit color, and pinball DMDs have 4,096 pixels with 4-bit color.)



Segmented Displays

MPF does not yet support segmented displays but we plan to soon. This document is just included here so you know we're working on it! :)

We'll support both alphanumeric and classic 7-segment numeric-only displays.

Slides

A "slide" in MPF's display system is essentially just a container for one or more [display elements](#). Slides don't actually show anything directly—they just hold display elements. For example, you don't put text on a slide per se, rather, you create a text display element which you then add to a slide.

The size of a slide (in terms of the number of pixels in its width and height) is controlled by the MPF Display that it's on. So a DMD display with a resolution of 128x32 that can show 16

shades of pixels will have slides that are 128x32 with 16 shades. An on screen window that's 800x600 with 24-bit color will have slides that are 800x600 at 24-bits.

In fact the act of creating a new a slide is actually function of the MPF display where you say "make me a new slide" and then it generates an empty slide that you can start putting your elements on. Keep in mind that MPF can support multiple displays at the same time, and in those cases each display has its own set of slides.

Each [MPF Display](#) can have multiple slides, though only one is "active" (i.e. "visible") at a time. You can add and delete elements from a slide anytime you want. If you add an element to the active slide then it will just appear on the display. If you add an element to an inactive slide then it will be there the next time you make that slide active.

You can also change and modify elements anytime you want, even when they are on an active slide. For example, you might have a [Text element](#) on a slide showing the player's score, and when the player gets more points then MPF just updates the text content within that existing element. Same slide, same element, just new text in that element.

Slide Events

When a slide is removed, if that slide has a name (set via the slide: setting in the slide_player), then an event will be posted in the form of *removing_slide_<slide_name>*. This can be used to trigger additional slides to be created.

The technical implementation for this is the MC sends a trigger to MPF, and MPF posts it like any event. If *slide_player:* entries are configured for it, then MPF will send the event to the MC.

Display Elements

Display elements are where the real action takes place in MPF's display system. They're what you (as a game programmer) interact with most often. They're the "stuff" that shows up on a display. (Well, technically, they're the stuff that is added to a slide, and then that slide in turn is shown on a display.)

Display elements live in files in the `/mpf/media_controller/elements/` folder. Currently MPF includes the following types of display elements:

- Text
- Shape
- Image
- Animation
- Movie

- Virtual DMD
- Character Picker
- Entered Chars

These display elements plug-in to MPF's display system in a standard way, so it's easy for us (or for you) to write your own display elements if you need to be able to display something that we don't include out-of-the-box. (We also have plans for more types of display elements, including ones to render segmented displays to the on screen window and ones that show the current status of lights, switches, and coils.)

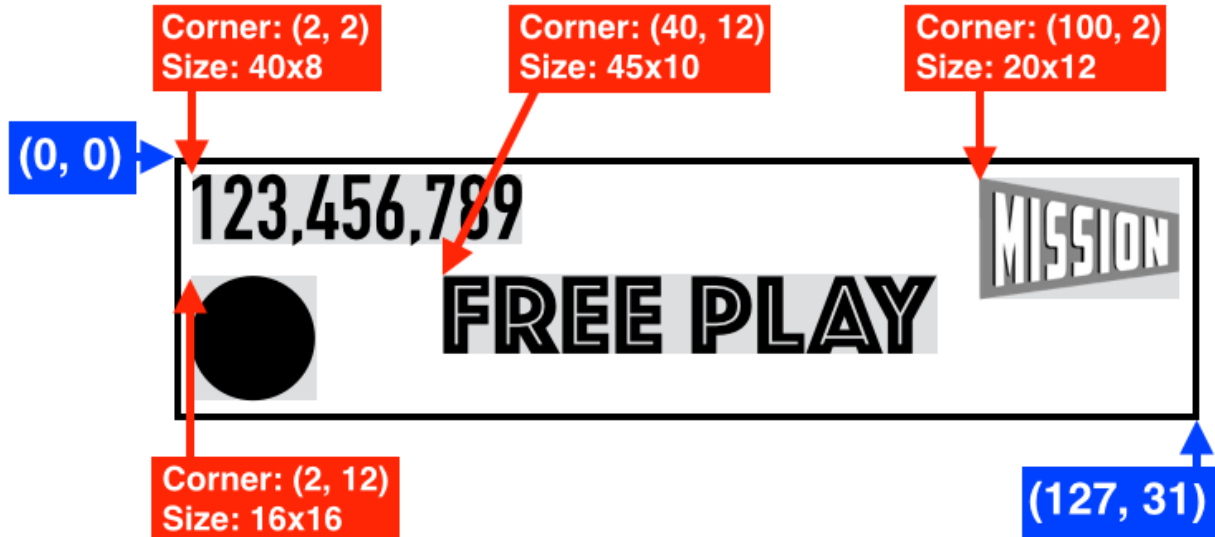
Positioning & Placement

All [display elements](#) in MPF are positioned on a slide in the same way, using the same settings, with the same results—regardless of what *type* of display element it is. Here's what you need to know:

- **All display elements are rectangular in shape.** Even if the image or shape is a circle or irregular, the bounding box that makes up the display element is rectangular. (Remember that individual pixels in a display element can be transparent, so it's possible for a shape to look like a circle even though it's really a square.)
- **All display elements have a width and height** which defines the size of the rectangle that makes them up.
- **All display elements have an (x, y) position** which represents the x, y position on the display of their upper-left corner.

Let's look at an example slide for a 128x32 display with four display elements on it. The light gray boxes represent the bounding rectangles of each display element.

- The 123,456,789 text element is positioned at (2, 2), meaning its upper left corner is 2 pixels over from the left edge and 2 pixels down from the top edge. This element's size is 40x8, which means the coordinates of its lower right corner are (42, 10). (Not that that really matters, since all positioning in MPF is done off the upper left corner.)
- The shape element with the circle on it is positioned at (2, 12), meaning its upper left corner is 2 pixels over from the right and 12 pixels down from the top. This is an example of where the positioning can seem confusing, because the pixel at (2, 12) is actually transparent, and to the viewer it wouldn't even appear like that circle has a pixel there. Since the size of the circle's bounding rectangle is 16x16, the lower right corner is at (18, 28).
- Etc.



Elements can be positioned so they're off the slide, even using negative numbers

It's possible for individual display elements to be positioned in such a way that they are partially (or completely) off the side. That's perfectly fine. In the diagram below:

- We moved the "y" coordinate of the 123,456,789 text element from 2 to -2, so now the top part of that element is off the slide. If you view this slide then it will still show the parts of the element that are in bounds.
- The "Free Play" text element is positioned at (25, 35), meaning it's completely off the slide. This is fully valid, and this element is still considered to be an element that's part of this slide. It's just that it doesn't happen to be positioned where anyone can see it. (Why would you want to do that? Several reasons, like if you wanted to create a really big slide and scroll it. More on that later.)
- The circle shape element is positioned at (128, 32). At first you might think, "Hey, this display is 128x32, so if I want to put an element in the lower right corner, then I just position it at the max." Obviously this is wrong, as the diagram below shows, since the positions are based on the upper-left corners of elements. And to add insult to injury, if you position an element at (128, 32) on a display that's 128x32, you won't even see a single pixel! (Since the display coordinates start at (0, 0) instead of (1, 0), the lowest rightmost pixel on a 128x32 display is at position (127, 31). So your element positioned at (128, 32) is the first pixel off the edge in both directions. :)



Applying this Knowledge to your Game

Ok, so now you've got the theory behind how elements are placed, how do *you* actually go about placing display elements on a slide. Really there are four settings you need to know about: `h_pos`, `v_pos`, `x`, and `y`.

`h_pos`:

This is "horizontal position", and it describes how this element will be placed on the slide horizontally. Valid options are `left`, `center`, and `right`.

The names are probably pretty self-explanatory. If you set `h_pos` to `left`, then the element will be positioned on the left edge of the slide. If you set it to `center`, then it will be centered, and if you set it to `right` then it will be positioned on the right.

If you read the theory part above, you know that positioning display elements in the center or the right is actually kind of complex since all the positional coordinates are based on the upper left corner of an element. So even when we center an element, we still have to figure out what the left position will be. Fortunately MPF does this for you, so if you just say "center" then it will figure it out.

(The formula for centering, in case you're curious, is $(slide_width - element_width) / 2$, and the formula for "right" positioning is $slide_width - element_width$).

v_pos:

This is "vertical position", and it's just like `h_pos` except it's for `top`, `center`, or `bottom`.

x:

This is the position for the "x" (horizontal) coordinate of your display element. `x: 0` is the left edge of your slide. Increasing `x` values move the display element to the right, and decreasing `x` values move it to the left.

The exact behavior of how your `x` value effects the horizontal positioning of your element depends on a few things:

- If you specify an `x` value and you do *not* specify a `h_pos` value, then the `x` value is the actual "x" pixel position on your slide. (Remember this can be negative or it can be higher than your slide is wide.)
- If you specify an `x` value and you also specify an `h_pos` value, then the `x` value represents the number of pixels of offset from that `h_pos`.
- If you specify an `x` value that is a decimal in-between 0 and 1, then it will be treated as a percentage value. You can also use this with or without an `h_pos` setting.

Let's look at some examples since this can seem tricky until you understand it.

```
h_pos: center
x: 2
```

The element will be positioned 2 pixels to the right of center.

```
h_pos: right
x: 2
```

The element will be positioned 2 pixels to the right of the right, which means it will be extending off the right edge of the slide by two pixels.

```
h_pos: right
x: -2
```

The element will be positioned 2 pixels to the left of the right edge, meaning there will be a 2-pixel gap between this element and the right edge of the slide.

```
h_pos: right
x: -.1
```

The element will be positioned 10% in from the right edge. Exactly how far 10% is depends on the dimensions of your element. (These decimal percentage values are really only useful for resizable on screen windows when you want things to stay relative to each other. If you have a fixed-resolution display in your machine, you can probably use absolute pixels for everything.)

```
x: 2
```

The element will be positioned 2 pixels to the left of the left edge. Note that if you do not include an `h_pos` value, it has the same effect as including `h_pos: left` since the left edge is the "zero" position for `x`.

If you don't specify an `x` or an `h_pos` value, then MPF will use the default setting (as specified in the `displaydefaults:` section of your config file. The out-of-the-box setting is `center`.

y:

The `y` value is just like `x`, except it relates to the vertical position, and it bases its anchor on the `v_pos` value. We probably don't have to go through an example here. Just remember that `y: 0` is the top of the slide, and the negative `y` values move the element higher up, and positive `y` values move the element down.

"Z-Ordering" (Layering) values

MPF also lets you specify what happens when you position two (or more) display elements so that part (or all) of them overlap. In this case you can specify layers which control which element is drawn on top of the other elements.

You can also control transparency settings for display elements, including whether the entire element should be partially transparent, and/or whether specific pixels should be transparent. (See the documentation for each display element for details.)

layer:

This is a numeric value of the layer of this element. Higher values equal higher layers, so an element with a layer of 1 will be drawn on top of an element with a layer of zero. There's no limit to the numbers you can use here. You can use 0, 1, 2 or 100, 1000, 10000, or -2, -1, 0—it really doesn't matter. All that matters is the relative values when the slide it being put together. The default layer (if you don't specify one) is zero. It's totally fine for you not to specify a layer for 99% of your elements. You really only need it in the rare cases where things are on top of each other.

opacity:

This is a decimal value (between 0 and 1) which affects the overall transparency of this display element. A value of zero means it's totally clear (i.e. invisible), and a value of 1 means it's totally opaque.

The key here is that this opacity setting only affects the overall opacity of the pixels which are not already transparent. Some elements might have individual pixels which are already transparent, and in those cases even if you have an opacity value of 1 here then the transparent pixels will still be transparent.

The default value is 1.

Where do you specify these positioning values?

So now that you know all the details about how to position and control display elements, you're probably wondering how do you actually do this? Where do you enter these values?

Well, in a nutshell, you do this in lots of places. Wherever you create and/or configure a display element, you also have the option of specifying `h_pos`, `v_pos`, `x`, `y`, and `layer` values. (Or any combination of none, some, or all of them.)

For example:

- In the [slideplayer: section](#) of your config file, you create entries for different display elements you want to show on the display based on MPF events.
- When you [create shows](#), you can add display elements into the `display:` section of a step, and those elements include these positional options.
- If you're manually adding display elements to a slide (either via a [scriptlet](#) or any other actual code you're writing), you can specify these positional values as arguments when you add a display element to a slide via the slide's `add_element()` method.

Here's an example of a Text element (from a show file) which has several elements with positioning values:



These positional settings combine with the display element's "other" settings

The last thing to keep in mind is that each type of display element has its own settings you need to specify. For example, the [Text element](#) has settings for the text string, font, size, antialias, etc., the [Animation element](#) has settings for the name of the animation, frames per second, whether it repeats, etc.

So in all cases, you can mix and match these positional `h_pos`, `v_pos`, `x`, `y`, and `layer` settings with the other settings that each element needs.

Text

The [Text display element](#) is used to render text to the display. Text can be displayed directly as entered, or you can access player variables and machine variables to dynamically control what text shows up.

If you're using a display which supports fonts (pretty much anything except for the segmented display), you'll also need to configure [MPF fonts](#). Luckily MPF can use standard TrueType fonts. We have a tutorial which explains how to import and configure TrueType fonts [here](#).

Text display elements use the same [positioning & placement settings](#) as other display elements. There are also text-specific settings, including:

text: (required)

This is the string of the text you'd like to display. To dynamically display text based on player variables or machine variables, see the section on dynamic text further down on this page.

font:

The name of the font (as you defined it in your configuration file) this text will be rendered in. If you don't specify a name, it will use whatever settings you have for the "default" font in your config file. Even though this setting is called "font", the [fonts you define in your config file](#) can also have settings which include the actual TrueType font, the size, and antialias settings.

number_grouping:

True/False setting. If True, it groups the number by thousands. (For example, "12345" will be rendered as "12,345.") You can specify the separator character in the config file.

antialias:

True/False. If True, the text will be antialiased, False and it won't be. See our guide for using TrueType fonts mentioned above for more information. Antialiasing is a property of the MPF font object, so if you specify it here you will specifically override the setting for the font you specified.

size:

You can override the font size setting as defined in your MPF font by entering a size value here. In most cases you would not want to do this. If you want to create text that's bigger or smaller, it's best to create a new entry in the [Fonts: section](#) of your config file for that size. That way you can fine tune the size and cropping settings.

shade:

If this text element is being used with a DMD, `shade` is the brightness value (0-15) of the text. If you don't specify a value here, it will use 15 (full brightness).

bg_shade:

If this text element is being used with a DMD, `bg_shade` is the brightness value (0-15) of the background. If you don't specify an option it will use 0 (black). Note that entering a value here will disable alpha blending.

color:

The hex string of the color when used with 24-bit surfaces. This has to be six characters, and you do *not* include the pound sign (#). So it's just entered like this: `color: ff5500` or `color='ff5500'` (depending on whether you're specifying this via a YAML file or via code). Default is white (`ffffff`).

bg_color:

The hex string of the background color when used with 24-bit surfaces. If you specify this, it will disable the alpha blending. Default is black (`000000`).

opaque:

A True/False setting which controls whether or not the background of this text element is opaque. If True, then if you position this text element on top of another, then the background color will block whatever's behind it. If False, then the background is transparent so only the characters in the text will show up. Default is False. If you set a `bg_shade` or `bg_color`, that will force `opaque` to be True.

Dynamic Text

Typically the text you enter in the `text:` section of the `text` display element is just shown on the display exactly as you enter it. However you also have the option of showing the values of player variables and machine variables in your text.

When you use the techniques described below to place dynamic text in slides, the value of the text is updated in real time whenever the underlying player or machine variable changes, even if the slide has already been created.

Accessing the current player's variables

You can access the value of a player variable for the current player by wrapping the variable name like this: `%player|<variable>%`. For example:

```
text: "%player|score%"
```

You can mix-and-match replaced text with hard-coded text. For example:

```
text: "YOUR SCORE: %player|score%"
```

Note that you don't technically need quotes around these entries, but it's probably a good practice to include quotes since quotes will force the YAML processor to view your entries as a string. Otherwise certain characters in your string could confuse or break things. For example, the first example above needs quotes because a YAML value can't start with a percent sign, and the second example would break because the colon after SCORE would confuse the YAML parser. Putting both of these examples in quotes fixes those problems.

Accessing variables from any player

You can access the value of a player variable from any player by wrapping the variable name like this: `%player<number>|<variable>%`. For example:

```
text: %player2|score%
```

The example above will always show the value of player 2's "score" variable, regardless of what the current player is.

What's cool is that if the player you're referring to doesn't exist, then MPF will not show anything. This means you can (for example), build a 4-up score display which shows all 4 players' scores, but if there's only a 3-player game going on then the display won't show anything for player 4.

Accessing machine variables

Finally, you can access machine variables with the `machine|` prefix, like this:

```
text: %machine|credits%
```

Putting it all together

You can combine all these techniques together to build up complex slides. For example, here's a typical 4-up score display slide:



The `slide_player:` configuration for this uses several text display elements:

```

player_1_ball_started:
- type: text
  text: "%player1|score%"
  number_grouping: true
  min_digits: 2
  v_pos: top
  h_pos: right
  x: -60
  y: 2
- type: text
  text: "%player2|score%"
  font: medium
  v_pos: top
  h_pos: right
  number_grouping: true
  min_digits: 2
  x: -2
  y: 2
- type: text
  text: "%player3|score%"
  font: medium
  v_pos: bottom
  h_pos: right
  y: -10
  x: -60
  number_grouping: true
  min_digits: 2
- type: text
  text: "%player4|score%"
  font: medium
  v_pos: bottom
  h_pos: right
  y: -10
  x: -2
  number_grouping: true
  min_digits: 2
- type: text
  text: BALL %player1|ball%      %machine|credits_string%
  v_pos: bottom
  font: small
  y: -1

```

Image

The *Image* [display element](#) is used to show images on a display.

Details on how to load images are in the configuration file reference for the [Images: section](#).

Image display elements use the same [positioning & placement settings](#) as other display elements. There are also image-specific settings, including:

image: (required)

The string name of the image you want to display.

width:

Used to change the width of the image. Not yet implemented.

height:

Used to change the height of the image. Not yet implemented.

Animation

The *Animation* [display element](#) is used to display and play an animation.

Details on how to load animations are in the configuration file reference for the [Animations: section](#).

Animation display elements use the same [positioning & placement settings](#) as other display elements. There are also animation-specific settings, including:

animation: (required)

The string name of the animation you want to play.

start_frame:

What frame number the animation should start on. The first frame is Frame 0. Default is 0.

fps:

How many frames per second the animation will play at. Default is 10, but that default is completely arbitrary and you really should set this to whatever your animation is meant to be played at.

repeat:

True/False as to whether this animation repeats when it reaches the end. Default is False.

drop_frames:

True/False which controls whether this animation should drop frames if it gets behind. If True, the animation will always stay "current", even if it has to skip some frames to stay on time. If False, the animation will play every frame, even if it means that it slows down. Default is True.

play_now:

True/False as to whether this animation should start playing immediately. If False, the animation will load and display the start_frame and wait to be played manually. Default is True.

width:

Used to change the width of the animation. Not yet implemented.

height:

Used to change the height of the animation. Not yet implemented.

Movie

The *Movie* [display element](#) is used for playing MPEG-1 movies to the display. This is typically used for background loops or for cut scenes.

The animation and movie display elements are similar. Animations can have transparencies and would typically be used for characters or foreground objects, or for DMD animations where the file size isn't too big.

Note that today *MPF can only play movies encoded in MPEG-1 format*. This is due to a limitation of Pygame (the Python multimedia library that MPF uses). At this point it's easy enough to use ffmpeg to convert whatever movies you want to play to the MPEG-1 format that MPF needs.

Also note that the *Movies module is not available on the Mac*. This is also due to Pygame, as the Mac version of Pygame does not support movies.

We will be moving MPF off of Pygame in late 2015, but until then, you can only use MPEG-1 movies and can't use movies on the Mac.

Details on how to load movies are in the configuration file reference in the [movies:](#) section.

Movie display elements use the same [positioning & placement settings](#) as other display elements. There are also movie-specific settings, including:

movie: (required)

The string name of the movie you want to play.

repeat:

True or False. (Or yes/no). If True, the movie will loop.

play_now:

Whether the movie should start playing right away, or whether it should start paused and wait to be told when to start playing.

start_frame:

The frame number you want the movie to start on. Default is 0. (The beginning.)

Shape

The *Shape* [display element](#) is used for drawing simple shapes onto the display.

Shape display elements use the same [positioning & placement settings](#) as other display elements. There are also shape-specific settings, including:

shape: (required)

A string name of the shape. Options include:

- box - Draws a box of a certain size. Additional settings include width and height.
- line - Draws a line from one point to another. Additional settings include width and height. The line is drawn from point (0,0) to the width and height specified.

thickness:

The thickness of the line in pixels. Default is 1. If you set this to zero, the shape will be filled in.

shade:

For DMD displays, the shade (intensity) of the shape, from 0-15. Default is 15.

color:

For 24-bit displays, the hex color string of the shape.

Virtual DMD

The *Virtual DMD* [display element](#) is used to render a DMD display into another display. In other words, it grabs the content of the DMD, renders it to look nice, and makes it available for other displays. This is how you get a DMD element in your on screen window.

Virtual DMD display elements use the same [positioning & placement settings](#) as other display elements. There are also virtual DMD-specific settings, including:

pixel_color:

Hex color value of a fully bright pixel. Default is `ff5500` which is an amber-orange color that looks like traditional DMDs. You can pick any color you want. Here are some examples:

MPF Virtual DMD Color Examples



TOP LEFT SIZE 5
CENTERED SIZE 10
BOTTOM CENTERED SIZE 7

pixel_color: ff0000



TOP LEFT SIZE 5
CENTERED SIZE 10
BOTTOM CENTERED SIZE 7

pixel_color: ff5500



TOP LEFT SIZE 5
CENTERED SIZE 10
BOTTOM CENTERED SIZE 7

pixel_color: ffffff



TOP LEFT SIZE 5
CENTERED SIZE 10
BOTTOM CENTERED SIZE 7

pixel_color: 00ff00

dark_color:

Hex color value of the pixels when they're off. Default is `221100`.

pixel_spacing:

The space in pixels between each pixel that's rendered on the virtual DMD. Here are some examples from a virtual DMD that's 512x128:

Pixel Spacing = 2



Pixel Spacing = 1



Pixel Spacing = 0



width:

Width (in pixels) of the virtual DMD.

height:

Height (in pixels) of the virtual DMD.

Displaying a Color DMD

Note that if your DMD is configured to be color, then this Virtual DMD element will display color pixels:



No changes are needed in this section to enable this. Since the virtual DMD just displays whatever's on the DMD, then if your DMD is color, the virtual DMD will be color too.

Character Picker

The *character picker* [display element](#) is used to render a list of characters to the display that the player can use to pick from when entering their name or initials. It's typically used in the high score entry mode, though you could use it in any slide if you want to do other things (custom message in the operator menu, allow players to set their name for a player profile, etc.)

You can specify what characters are included in the list, the spacing and positioning of the characters, the font they used, colors for the selected and unselected characters, and images to be used for "special" characters like the backspace and end characters.

The character picker's size and position on the screen is controlled just like any other display element, so you can refer to the [positioning & placement settings](#) settings there.



Character picker-specific settings include:

font:

The name of the font that will be used to render the characters. This is [configured like any other font](#) in MPF.

name:

The name setting for the character picker display element is no different than the name of any display element. However, the name is more important for the character picker because this is the name you use to link a related [entered chars](#) display element to display the characters that have been entered so far.

selected_char_color:

The color of the character that is selected (highlighted) in the list. For mono-color displays, this is an integer value corresponding to the brightness. For color displays, this is a six-character hex color code. Typically this would be dark.

selected_char_bg:

The background color of the character that is selected (highlighted) in the list. For mono-color displays, this is an integer value corresponding to the brightness. For color displays, this is a six-character hex color code. Typically this would be a light color to create a bright box around the selected character (which itself would be dark).

char_x_offset:

This is a positive or negative value that lets you fine-tune the x position (horizontal) of the character in the box. Most font render the individual characters to go right up to the edge of their bounding box, so this value lets you shift the character horizontally so it's not smashed up against the edge.

char_y_offset:

This is the same as the *char_x_offset*, but it affects the y (vertical) positioning.

char_width:

This is the width, in pixels, that each character will take up in the list. This is needed because most fonts are variable-width, meaning a character "i" takes up much less horizontal space

than a "w". When an entire list is rendered to the display, it looks weird if all the characters have different spacing, so this setting lets you specify how many pixels wide each character will be regardless of how wide the actual font character is rendered.

char_list:

This is a list of all the characters (in order) that will be in this character picker. Basically it's where you specify a list of letters the player can pick from. For example: "ABCDEFGHIJKLMNOPQRSTUVWXYZ_ ". (If you put it in quotes, you can also include a space as a valid character in the list.)

back_char:

This is the name of the MPF image asset that will be used to render the "back" arrow which, when chosen, causes the entered characters to delete the last one and go back a space. This image is what the back character looks like when it's not selected.

end_char:

This is the name of the MPF image asset that will be used to render the "end" character which is what the player selects when they're done entering their name.

back_char_selected:

This is like the *back_char*, except it's the image that's used when the *back_char* is selected.

end_char_selected:

This is like the *end_char*, except it's the image that's used when the *end_char* is selected.

image_padding:

This is how many pixels of spacing you want on the left and right sides of the back and end character images.

shift_left_tag:

The tag of the switch in the machine that causes the character picker to shift the selected character one position to the left.

shift_right_tag:

The tag of the switch in the machine that causes the character picker to shift the selected character one position to the right.

select_tag:

The tag of the switch in the machine that causes the character picker to select the currently-selected (highlighted) character.

max_chars:

The maximum number of characters the player can select. If you just want initials, this would be 3. If you want to allow full names, this could be longer, like 10.

timeout:

This is the timeout (in MPF time string format) that specifies how long the character picker will sit there waiting for the player to enter their name before it times out.

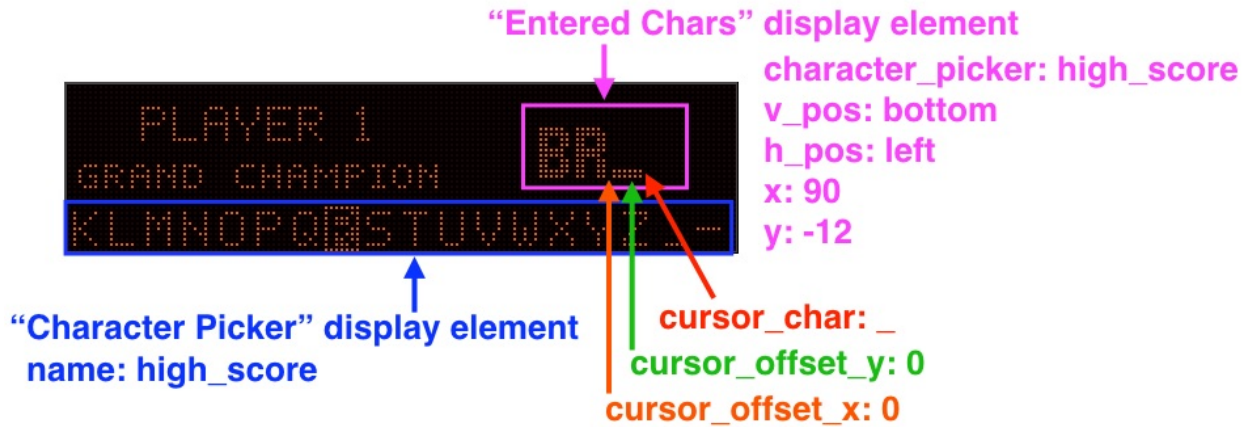
return_param:

The character picker works by collecting the characters the player has entered and sending them back to MPF as an input. This setting is where you specify what that return parameter will be called.

Entered Chars

Entered Chars (short for "Entered Characters") is a [display element](#) that works hand-in-hand with the [character picker](#) display element. It's where the characters the player is selecting when entering text appear on the display.

The entered char's size and position on the screen is controlled just like any other display element, so you can refer to the [positioning & placement settings](#) settings there.



Entered chars-specific settings include:

character_picker:

The name of the character picker display element that the player uses to generate the characters for this entered chars display element. It can be whatever you want, just make sure you have the same value set for the *name*: section of the *character_picker* element.

cursor_char:

The character that will be used as the cursor character. The underscore is the default.

cursor_offset_x:

Let's you specify the x (horizontal) offset of the cursor character, in pixels. This can be positive or negative and is used to fine-tune the placement of the cursor character.

cursor_offset_y:

Like *cursor_offset_x*, except for the y (vertical) offset.

cursor_decorator:

This is the [decorator](#) that will be applied to the cursor character. It's a separate setting from the overall decorator that's applied to the entered chars element so you can make the cursor blink without all of the entered characters blinking.

Transitions

Transitions are used to switch from one active slide to another in a fun way.

If you don't have a transition when you switch slides, then the new slide will just instantly appear in place of the old slide.

Each transition has lots of different settings to control exactly how it works, though they all have two things in common:

- The duration of the transition (1 second, 5 seconds, 250 ms, etc.)
- The new slide that will be transitioning in.

We already mentioned that each display in MPF only has one active slide at a time. I guess you could say that's kind of not true, because during a transition you actually have two active slides while the transition is taking place—the outgoing old slide and the incoming new slide. (Though technically speaking what happens is the transition creates its own slide which itself is composed of the visible parts of the incoming and outgoing slide.)

When a transition is taking place, both the incoming and outgoing slides are active, so if you have animations or other moving elements then they will continue to play and move even while the transition is taking place.

Also if you want to go nuts, you can actually run multiple transitions at the same time. So you could have one slide fading in while it itself had another slide revealing itself. Kind of cool.

We currently have two transitions written:

- [Move In](#). The new slide moves in on top of the current slide. You can specify whether it moves in from the top, left, right, or bottom.
- [Move Out](#). The current slide moves out of the way, revealing the new slide below it. Like the Move In transition, you can specify which direction it moves out towards.

We will be creating lots more transitions in the future. If you want a list of ideas, just open PowerPoint and take a look at all the slide transitions in there. :) We'll probably end up creating all of those for MPF. Here's the list we have so far:

- Push: The new slide pushes the current one out of the way, so the current one slides out while the new one is sliding in.
- Fade: Fade from one slide to the next.
- Fade through color: Fade from the old slide to a color, then from the color to the new slide. If you do this with a bright color and really fast, then it's kind of cool, like a "flash" effect that ends up with a new slide under it.
- Zoom: The current slide zooms in to reveal the new slide.

- Shrink: The current slide shrinks down to nothing, revealing the new slide.
- Fall: The current slide "falls" down, revealing the new slide.
- Pixelate: The new slide comes in pixel-by-pixel (but really fast) to reveal the new slide on top of the current slide.
- Slice: The current slide is sliced into pieces to reveal the new slide.
- Blinds: Like a reveal, except it happens in lots of rows at once.
- Corners: The new slide comes in from all four corners to meet in the middle.

MPF's transition architecture is easily extendable, so it's very straightforward for you to write your own transitions which we can then add to MPF and share with everyone.

Move In

The "Move In" transition is used to have the new slide smoothly move in on top of the existing slide. You can configure the direction the new slide comes in from, as well as how fast it moves.

Here's a demo of it in action:

<https://www.youtube.com/watch?v=0vpG0NFee-0>

Settings:

- Transition name as it's entered into YAML files: `move_in`
- Direction. `top`, `right`, `left`, or `bottom`. Default is `'top'`.
- Duration: An [MPF time string](#). Default is 1 second.

You can add this transition to your ``SlideBuilder:`` section in your machine configuration file and to the ``display:`` section of show files.

Move Out

The "Move Out" transition is used to have the current slide smoothly move out, revealing the new slide beneath it. You can configure the direction it moves and how long it takes to fully move out.

Here's a demo of it in action:

<https://www.youtube.com/watch?v=wojAl93fSW0>

Settings:

- Transition name as it's entered into YAML files: `move_out`

- Direction. top, right, left, or bottom. Default is 'top'.
- Duration: An [MPF time string](#). Default is 1 second.

You can add this transition to your `SlideBuilder:` section in your machine configuration file and to the `display:` section of show files.

Fonts

The Mission Pinball Framework uses regular TrueType fonts to render text for the on screen window display as well as for the DMD.

As a game designer, you collect the `.ttf` font files you want and put them into your game folder. Then if you ever copy your game files to another computer, the fonts go with them.

Part of your role as a game designer is to pick the fonts that you want to use and then to configure them so that they look good in a pinball machine. (This is especially important for DMDs since they have limited color and resolution.)

When you're ready to start thinking about fonts, there are several resources for you:

- A [step-by-step guide](#) detailing how to add TrueType fonts into your machine.
- Instructions for the "[Font Tester](#)" graphical tool we built which you can use to see which fonts (and with which settings) will look best on your DMD.
- The configuration file reference documentation for the [fonts: section](#) of your config file, which is where you actually enter your font configuration information.

Decorators

Decorators are used to, well, "decorate" display objects in MPF. They're essentially like display effects that are applied on top of [display elements](#).

For example, you might have a Text display element which shows the text "PRESS START" on the display. If you want that text to blink, then you can apply the "blink" decorator to it, and boom! You now have blinking text.

Decorators can be applied to any display element (text, image, animation, shape), etc. You can chose to decorate all the elements on a slide or just certain ones. You can apply multiple decorators to the same element, and decorators continue to function even when the slide is transitioning in or out.

Different types of decorators have different settings. The blink decorator, for example, lets you specify the on time and off time (in seconds), as well as how many times the element will blink. (Once, twice, forever, etc.)

So far we've only created one type of decorator (blink), but we have plans to create more. (Fade, slide, rotate, expand, sparkle, shimmer, color shift, invert, edge glow, etc.) It's very easy to create and plug-in your own decorators too. (If you do then please share them with us so we can include them in the MPF package!)

Blink

The "Blink" decorator is used to cause a display element to blink. You can specify the on time and off time (in seconds), as well as how many times you want the element to blink.

Here's a demo of it in action. Notice that the decorator continues to function even as the slide is transitioning out.

<https://www.youtube.com/watch?v=tFNq-wJNGEs>

Settings

- Decorator name as it's entered into YAML files: `blink`
- `on_secs`: The number of seconds the element is on. Decimals are ok.
- `off_secs`: The number of seconds the element is off. Decimals are ok.
- `repeats`: The number of times the blinking will repeat. Enter `-1` for forever.

Example Usage

Here's an example of how to apply this decorator to text which is in a show file. Notice that the decorator only applies to the "PRESS START" text, and not the "FREE PLAY" text.

```
- tocks: 3
  display:
    - type: Text
      text: "(PRESS START)"
      decorators:
        type: blink
        repeats: -1
        on_secs: .4
        off_secs: .4
    - type: Text
      text: "(FREE PLAY)"
      color: 00ff00
      v_pos: bottom
      font: small
      transition:
```

```
type: move_in
duration: 1s
direction: right
```

Sounds & Audio

Audio and sound support is handled via the a plug-in called the Sound Controller (in `/mpf/plugins/sound.py`). All audio files are played back via Pygame—a required add-on module for Python that MPF also uses for its display system and the on screen window.

The basic concept with audio in MPF is that you collect all your audio files (.wav or .ogg) and put them in the `/sounds` folder in your machine folder. 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 offset, 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.

As we said, the current sound support is pretty basic, but we have future plans for a lot of stuff, including:

You also have the option to set up multiple "tracks" for sound. Maybe you have one track for background music, one for voice callouts, and one for sound effects. Then when you play a sound you can specify which track it will play on, and MPF will use the properties of the track to make sure it plays properly. (The sound effects track might allow multiple sounds to be played at once, while the voice track would want to make sure only one sound is playing at a time.)

You configure your sound system in the [sound_system:](#) section of your machine configuration file. You add settings for individual sound files in the [sounds:](#) section, and you can configure sounds to automatically play when certain MPF events are posted in the [sound_player:](#) section.

Machine HZ & Loop Rates

In the world of custom pinball, there's a lot of conversation around "loop rates" and "machine speed" and things like that. MPF gives you fine-grained control over how "fast" your machine runs, both in terms of how fast MPF polls the pinball controller hardware and how often MPF "wakes up" to do things. There are a lot of misconceptions about these topics, so this page is meant to clear those up and explain exactly how these concepts are implemented in MPF.

The MPF main loop

There's a setting in the machine-wide config called `timing: hz:` which controls how many times per second MPF "wakes up" to run its main loop. The default value for this setting is 30, meaning that MPF runs its main loop every 33ms. (Technically it's 33.333333ms, but we'll use 33ms for simplicity.)

Several things happen in this loop:

- Any periodic system timers that should tick now are processed.
- Any sleeping tasks that are scheduled to wake up are run.
- Any delays that are ready to be processed are run
- A machine variable called `tick_num` is incremented by 1.
- An event called `timer_tick` is posted, which itself causes lots of things, including:
 - Light and display shows that need to advance are advanced
 - The display is updated
 - BCP messages are read from the incoming BCP queue
 - The switch controller processes delayed switch actions
 - Plus a few other things...

Once these actions are done, MPF goes back to sleep until it's time for the main loop to run again. The amount of time MPF sleeps varies depending on how long the previous loop actions took. For example, if the MPF loop is set to run every 33ms and the loop took 6ms to complete, then MPF will sleep for 27ms so the next loop starts on time.

MPF records its overall start and stop time, as well as tracking the number of main loops that took place. When you stop MPF, it does some quick math and shows what your target and actual loop rates were on the console and in the log file, like this:

```
INFO : Machine : Target MPF loop rate: 30 Hz
INFO : Machine : Actual MPF loop rate: 30.13 Hz
```

It's a good practice to check to make sure these values are in the same ballpark. (On very short runs of MPF it's possible to see loop rates above the target due to the way the timing is started.) If you have a target loop rate of 30 and an actual loop rate of 15, you know you either need to lower your loop rate or buy faster hardware. :)

The hardware polling loop

Most "action" in a pinball machine happens because a switch changes state. In pinball, we want the machine to respond "instantly" (or as near-instantly as possible) to switch actions, so therefore *MPF does not process switch changes in its main loop*. Rather there's a second

timer that runs while the main loop is sleeping to poll the pinball controller hardware to receive any switch changes.

That hardware polling loop, by default, happens every 1 millisecond. (You can control this via the `timing: hw_thread_sleep_ms`: machine wide config setting which is set to "1" by default.)

So while the main loop is sleeping, MPF wakes up (every 1ms in this case) to poll the hardware to see if there are any switch changes, and if so it services that switch (by posting an event that that switch has changed and/or processing and switch handlers that have been added for that switch).

If multiple switch events came in via a single hardware poll, MPF will service them all, one-by-one, in the order they were received.

Then when MPF is done servicing those switches, it will go back to sleep (for another 1ms), poll for new switches, go to sleep again, etc. When the time for the main loop sleep is up, MPF will do a full wake-up and run the main loop again.

What happens if a switch change happens while MPF is in its main loop?

MPF does not poll the hardware for switch updates while the main loop is running, so if a switch changes state while the main MPF loop is doing it's thing, MPF won't process it until the main loop is done.

This is by design.

The main reason for this is that while MPF is in the process of running its main loop, it would be confusing if that loop was interrupted to process another switch. So instead MPF just runs its main loop as fast as it can, and then when its done it polls the hardware again to see if anything switches changes while the main loop was running.

The main loop runs quickly—typically under 10ms—so in practical terms this only means that the "delay" of processing switches that changed while the main loop is running is 10ms or less.

We've gotten into arguments with people about whether this is acceptable, but the reality is most humans see things as "simultaneous" if they happen within 50ms or so of each other, so if switch processing is delayed by 10ms or so, no human will notice. In fact old Williams WPC hardware running at 2Mhz was much slower than this in terms of how it read and processed switches, so if it wasn't a problem for the past 30 years then it's not a problem now. :)

Why not just run the hardware polling as fast as possible?

Some people have questioned why MPF "sleeps" for 1ms between polling the hardware for switch changes. Wouldn't it be better to poll the hardware as fast as possible?

Technically speaking, MPF can do that. (Just set the `hw_thread_sleep_ms`: to 0. This will produce some impressive numbers for your hardware loop rates that you can brag to your friends about, but in reality it serves no practical purpose and in fact makes things worse overall!

Why is this worse? First, remember that the default polling interval is every 1 millisecond, so if you set the polling to 0ms then what do you gain really? Best case is you shave a few tenths of a millisecond off the response time. So what? Humans are slow.

Second, and more importantly, if you run your hardware polling loop as fast as possible, that means your MPF process is going to consume 100% of your CPU (which makes sense since you're telling it to run as fast as possible!) But that's bad for a few reasons. First it means that your CPU is heating up and just running full out 100% all the time. That is not good in terms of longevity of your hardware. Second is it means that your OS will generally be unresponsive since you essentially have a "bad" process that's trying to take up as much CPU as it can.

On many computers you'll see MPF only take less than 10% of the CPU just by letting it "yield" to other processes once a millisecond. Heck, you can even change that hardware thread sleep time to 2 or 3 or 5ms with no perceivable difference in performance and drop your CPU usage even more.

Viewing your actual hardware loop rate

When you stop MPF, the console will also show the average of how many times it was able to poll the hardware per second. Again this is based on MPF knowing how long it was running and then dividing that by the number of times it polled the hardware. When you're using the virtual hardware platform, you'll see really high numbers, like this:

```
INFO : Machine : Hardware loop rate: 748.9 Hz
```

Physical hardware will be lower since there's delay in the USB bus, but even with physical hardware you'll still see hardware polling over 100Hz which means there was only an average of 10ms delay from the time a switch changed until MPF knew about it.

These two loops do not affect "instant" actions like flippers and pop bumpers

The other thing to keep in mind is that so-called "instant" response actions—like the flipper buttons controlling the flippers or pop bumper or slingshot switches activating those

devices—are handled directly by the pinball controller with no interaction required from MPF or the host computer.

All modern pinball controller hardware support these types of instant-response rules. These rules are constantly being updated, overwritten, added, and removed. For example, when a ball starts, MPF will update the rules on the hardware to enable these instant-response actions. When a ball ends, MPF removes those rules. When the next ball starts, MPF enables those rules again, etc.

Instant action rules are based on switch actions. So when a switch tied to a pop bumper is activated, the hardware controller pulses the pop bumper coil. Then the next time that MPF polls the hardware, it will receive notification that the pop bumper switch is activated, and based on that MPF can process a score, update the display, and play a sound. So in reality there might be a few millisecond delay between the time the hardware controller fires the pop bumper coil and when MPF processes it, but again this all happens faster than humans can perceive. (Heck, the speed of sound is only about 1 foot per millisecond, so there's even a 4ms delay from the sound coming out of the speakers in the backbox until it reaches the player's ears.)

We did some experiments where we added a delay between the flipper button press and the activation of a flipper, and for most players we could take that delay all the way up to 30ms before people even noticed. (Not that we recommend that, rather it was just to prove a point that humans are slow!)

The last thing to know about these instant response rules in MPF is that while most people think of them just in terms of flippers, pop bumpers, and slingshots, MPF actually writes hardware action rules for anything that needs to happen instantly, including things like kickbacks and diverters. So even if you have a crazy slow MPF main loop rate of 10hz (meaning there is 100ms between loops), a fast-moving ball in front of a diverter will still cause that diverter to fire in time because that's being serviced by the pinball controller hardware rather than the MPF main loop.

Keyboard Interface

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. (You can also simulate switch events via a phone or tablet with our OSC interface.)

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 in case 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.

If you want to use the keyboard interface, details of the keyboard mapping configuration can be found in the [keyboard: section of the configuration file reference](#).

Multi-language Support

MPF supports the ability to change the language of your pinball machine. This can be used for actual languages, like changing it from English to German, or it can be used to install "alternate" language packs (like changing the language to "family friendly" which might just replace a few text strings and audio callouts). The ability to support multiple languages is built into the core of MPF and it's pretty easy to use.

Features of MPF's multi-language support

- If you don't care about multiple languages, then the language system stays out of your way. You don't have to do anything weird or think about multi-language at all if you don't care.
- If you later want to add multi-language support, it's easy, even if you didn't think about it when you were originally creating your game.
- Creating the actual "translations" is also easy, so you can send them out to non-technical translators and then plug their results right into your game.
- The language system can be used in novel ways for things other than actual languages. For example, rather than translating from one language to another, you

can use it to replace a few key elements to create a "family friendly" version of a game.

- You can use as much or as little of the language system as you want. So this means if you only want to use it for one or two things here and there, that's ok easy.
- Translation settings are changeable on the fly, so you can change the language of a machine from player-to-player (or even during a ball) without having to "reload" anything.
- The multi-language system can be able to control all aspects of the game which could contain language-specific things, including text, images, and animations, as well as sound and audio.
- You can configure multiple languages in a hierarchy, so when a string that needs translating comes up, it can first look for it in Language 1, then Language 2, etc. until it finds it.

Working with text strings in multiple languages

You can configure text strings for alternate languages in the ``LanguageStrings:`` section of your configuration file. Details on how that works are in the [configuration file reference](#).

To use the text string language replacement system, all you have to do is wrap the text you want replaced in parenthesis. You can do this anywhere—in text elements that occur in show steps, in SlidePlayer elements in your config file, etc.

Basically anywhere you have a ``text`` setting, instead of entering the text like this:

```
text: GAME OVER
```

you instead enter it like this:

```
text: (GAME OVER)
```

Doing that means that whenever that text is displayed, the language module will look in its list of text strings for the current language and see if it can find one for "GAME OVER". If it finds a match, it will return the replacement text. If it doesn't find a match, it will strip off the parenthesis and return the original text.

A sample ``LanguageStrings:`` section of your config file could look like this:

```
LanguageStrings:
  French:
    GAME OVER: JEU TERMINÉ
  German:
    GAME OVER: SPIEL IST AUS
  Kid-friendly:
    GAME OVER: WE LOVE YOU SO MUCH!
```

(Note that the French translation has an accent mark, so be sure to save your config file in UTF-8. Also be sure your font has a character for that.)

Setting the current language

You configure the current language and the search order for alternate languages in the [`Languages:` section](#) of your configuration file. Right now that's something you have to do manually, but we'll soon create MPF events so you can control all this from your shows and config files.

Plugins

The Mission Pinball Framework is extensible, meaning you can easily add your own code or modules without having to "hack" the core code. When you download the framework, it includes several plugins for things you might find useful right out of the box. You're free to use these as is, to customize or extend them, or to replace them altogether with your own.

Built-in plugin modules include:

- **OSC.** An interface to control a pinball machine from an iOS or Android device.
- **Auditor.** Records game, player, and switch audit information to a log file.
- **Info Lights.** Lights up backbox lights based on different game and player events. (Typically found in EM machines.)

We have several plugins in the works which we will include soon, including:

- **Ball Save.** Gives the player another ball if they lose it too quickly.
- **Ball Search.** Looks for missing and stuck pinballs.
- **Service Menu.** An mode for operators and owners to change machine settings.
- **Credit.** Handles coins, dollars, and tokens.
- **Switch Recorder.** Lets you record and "play back" switch events from one game for repeat testing.
- **Test Mode.** Lets you test coils, lights, and switches.
- **High Score.** Manages and tracks high scores.
- **Valid Playfield.** Used to track whether a ball launch was successful.

Auditor

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:
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
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](#).

Info Lights

The Info Lights plug-in is used to communicate game status information via lights, including the number of players, the current ball, match numbers, tilt status, and whether a game is in progress. This is typically used in EM games since they have lights for all these things in the backbox.

It's technically possible to do this all manually via logic blocks and light scripts, but this plug-in is dead simple. You just map your lights to game roles and then forget about them. Done.

If you want to use this plugin, add it to your list of plug-ins in your configuration file, like this:

```
Plugins:
  info_lights.InfoLights
```

Details on how to configure this plug-in are available in the [`InfoLights:` section](#) of the config file reference.

OSC Module

The Mission Pinball Framework includes an OSC plugin which allows you to interface with a pinball machine via an OSC client running on a phone or tablet. (If you've never heard of OSC, it stands for "Open Sound Controller" and is a protocol that's similar to MIDI.) OSC software clients for phones and tablets are typically used by DJs in clubs to control musical devices, but as it happens the OSC protocol is perfect for interfacing with a pinball machine too! (Frankly one of the main reasons we chose OSC was because there are already a ton of different OSC clients for iOS and Android, so if by building an OSC interface to our framework we get don't have to write our own client app.)

You can see a [demo of the MPF OSC interface here](#).

Controlling a pinball machine via an OSC client is great for testing and debugging purposes when you're not sitting next to your physical pinball machine. You can literally "play" your game via your phone. (Though, we have to admit, it's also fun to fire up your OSC client on your phone when a friend of yours is playing your game and to mess with them by hitting the flipper buttons via your phone without them knowing about it. :)

Personally we have been using an OSC client app for iOS called "TouchOSC," but there are literally dozens of options and you can use whatever you want.

Currently our framework's OSC interface lets you control switches, lights, coils, and events from your OSC client. You can also use the OSC client to view the current status of lights and switches.

Our OSC plugin allows multiple OSC clients to connect at the same time, so you can connect an iPhone and iPad at the same time and view/control lights on one and switches on another. :)

Prerequisites

- [PyOSC](#) (You should also be able to install this via `pip install pyosc` from the command prompt.)

Using this plugin

1. Configure options for this plugin in the [osc: section](#) of your Machine Configuration Files.

Configuring OSC Clients

We'll add full documentation for configuring OSC clients soon, but in the meantime here's the down-and-dirty guide.

You need to send an empty OSC message to the address `/sync` to establish a connection with the OSC host running as part of MPF. If you've set your machine type to 'wpc', you can alternately send an empty message to `/wpcsync`. Using `wpcsync` tells MPF to look for and to send WPC coil, switch, and lamp numbers instead of their names.

Switches

On your client, you can use `/sw/switchname` to control or display switch states. MPF will send and OSC data value of 1.0 when a switch is active, and 0 when it's inactive. On the flip-side, it will expect to receive a value of 1.0 to activate a switch or 0.0 to deactivate a switch.

Matrix Lights

On your client, you can use `/light/lightname` to control or display switch states. MPF will send and OSC data value of 1.0 when a switch is active, and 0 when it's inactive. On the flip-side, it will expect to receive a value of 1.0 to activate a light or 0.0 to deactivate it.

Coils

Send an OSC message to `/coil/coilname` to pulse that coil. Currently this only supports pulse, and it's one-way. (i.e. the OSC client will not display the current status of coils like it does with switches and lights.)

Events

You can send an OSC message to `/ev/eventname` to cause MPF to post whatever event you want.

RGB LEDs

This feature is not yet implemented.

Switch Player (Test Automation)

The switch player plugin is used to automatically "play" a sequence of switch events into MPF. This is useful for testing since it means you can write switch sequences which always play back in the same way, and it's easier to get deep into the game without having to set up an OSC client or set up complex key mappings.

Then you can add specific switch step entries in the in [switchplayer: section](#) of your config file.

Creating your own Plugins

If you want to add to, modify, or extend the core functionality of the Mission Pinball Framework, you can create your own plugins. Plug-ins are conceptually similar to [scriptlets](#), except plugins are designed to be generic things that could apply to any pinball machine, whereas scriptlets are meant to add custom code to individual games.

To create a plugin, all you do is create a python file that contains the class for your plugin, then you add it to the `/plugins` folder. You can enable your plugin for a specific machine by adding it to the [Plugins section of that machines configuration file](#).

The only thing to know about creating a plugin is that when it's loaded, the machine controller will pass a reference to itself as the only argument for the `__init__()` method. So for example, you'd want to start your plugin like this:

```
class MyPlugin(object):
    def __init__(self, machine):
```



```
self.machine = machine # save a reference here so you can access
everything else in the game
```

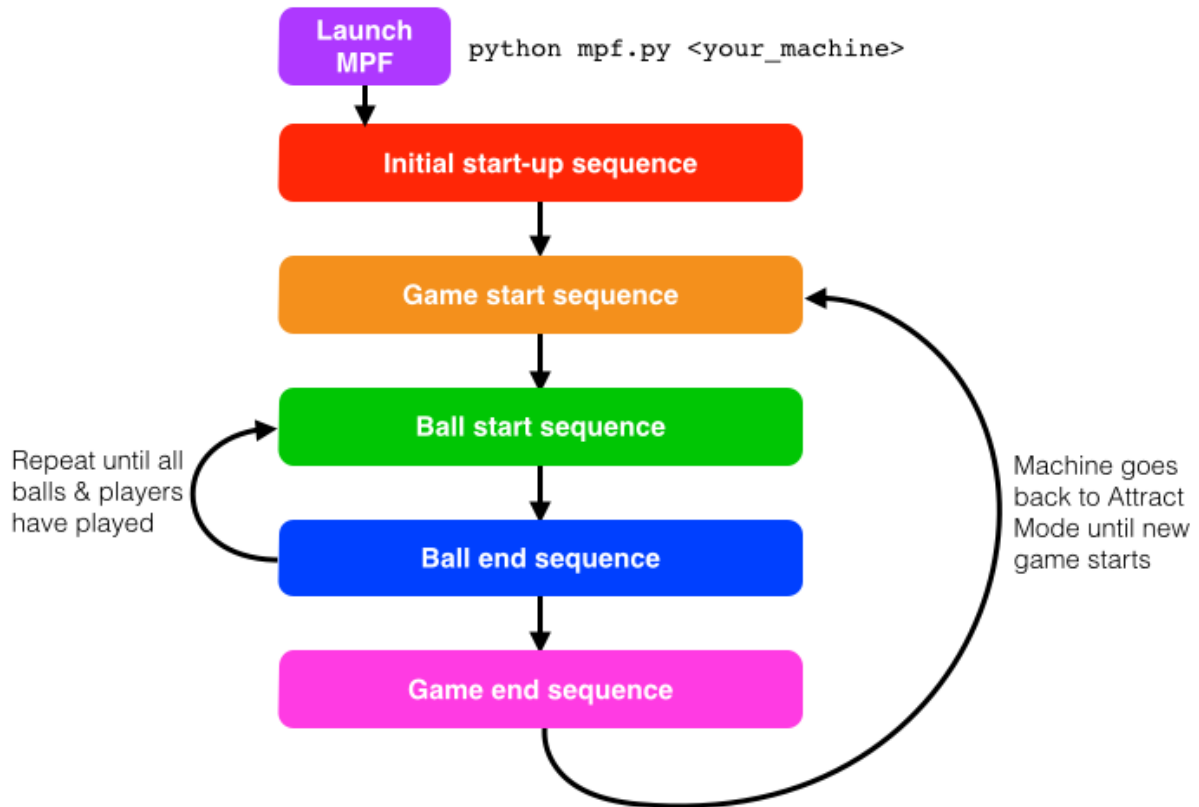
What you do with your plugin is up to you. Since plugins and scriptlets are so similar, you can check out our [example scriptlet file](#) for some ideas. (The biggest difference is that since scriptlets will probably be used by beginners, they subclass `Scriptlet` which sets up the environment for them. Plugins subclass `object` since if you're writing a plugin, you probably know what you're doing.

You can also check out all the existing plugins which come with MPF in the `/plugins` folder for an idea of how they work.

System Flows

The software that runs a pinball machine is really complex. Even though MPF hides a lot of that complexity from you (the game programmer), 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.



Initial Start Up & Boot 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 `python mpf.py` 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, 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. 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*.

Ball Start Sequence

Now let's look at what happens when a new ball starts. 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 is 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_start*.
 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 tagged with `ball_add_live`, 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 and posts the *ball_add_live* event.
 6. The ball device with the `ball_add_live` tag responds by ejecting its ball.
 7. When that ball eject is confirmed (based on the settings for that device), the ball controller posts the *ball_live_added* event.
 8. 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.

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.

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 and 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! :)

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.

7. Game Programming

When it comes time to program your game, here's the basic high-level steps you can follow:

1. Follow the [step-by-step tutorial](#). When you finish that, you'll have all your hardware defined and a basic functioning game with a base game mode and basic display.
2. Browse through the ["How To" guides](#) which will give you step-by-step instructions to build additional things we didn't cover in the initial tutorial.
3. Start building your [game logic & rules](#).
4. If you need to add custom Python code to extend what you can do with config files alone, read our intro to custom code first.

Game Logic & Rules

In MPF, there's no single "feature" that's used to create game logic and rules, rather, it's a combination of many different parts of MPF, including:

- MPF events
- Player variables
- Game modes
- Logic blocks
- Timers
- "Real" programming

This section of the documentation explains the theory and general approach to how you program game logic and rules. The nitty gritty details for each section are covered in the full documentation for each item. You can also follow our step-by-step tutorial for specific instructions and examples on how to set all of this up.

MPF Events

MPF makes *heavy* use of the concept of "events." An event is like a message that one part of MPF can post that other parts of MPF can respond to. (More info [here](#).) Events have quasi-friendly names like "ball_started" or "ball_live_added" or "target_x_lit_hit." In MPF, there are lots and lots and lots of events. Seriously. Dozens per second. Maybe hundreds.

Events are your key to creating your game logic completely in config files without "real" programming. You'll notice as you browse through the configuration file reference that

events are often the *result* of something happening, and/or that they can be the *trigger* which causes something to happen.

Some examples of events that get posted as the result of things: (This is a tiny fraction of the list. See [here](#) for more examples of events.)

- A ball enters a device. Event is posted.
- A shot is made. Event is posted.
- Ball drains. Event is posted.
- A switch is hit with a tag. Event is posted.
- Shows can contain events which are posted when they get to a certain step.
- etc.

Some examples of how you can use events to make things happen:

- [Modes](#) have a list of one or more events that, when posted, cause them to start and stop.
- Logic Blocks and Timers (both explained below) use events to start and stop and post events as they make progress.
- [SlidePlayer](#) config entries show display content on the DMD or LCD based on events.
- [SoundPlayer](#) config entries play sounds based on events.
- [ShowPlayer](#) config entries play light/sound/display shows based on events.
- [Scoring](#) is configured to assign points based on events.
- etc.

So generally speaking, you'll be using events as the basis for the bulk of your game logic.

Player Variables

A Player Variable is what we call a "per player" setting in MPF. Basically these are used for everything you want to track per player and remember from ball-to-ball. MPF automatically uses a few different ones out of the box, including:

- ball (what ball this player is on)
- number (what player number this player is. e.g. *Player 1*, *Player 2*, etc.)
- score (the player's score)
- extra_balls (how many extra balls this player has stacked up)

MPF's player system is very flexible. There is no central "list" of player variables, rather, anything can be stored as a player variable and it's automatically remembered for that player (and only that player).

Player variables are part of the game logic and rules conversation because every time a player variable is changed, an event is posted. (Seriously, events are really important!!)

The event has the name `player_<variable_name>`, and it's posted along with three key/value pairs: *value* (the new value), *prev_value* (the previous value), and *change* (the numeric change in the value, or if the values are not numeric, a simple True/False as to whether the value actually changed).

For example, if the player's score is 1,000,000 points and something in MPF gives them 50,000 points, an event will be posted called `player_score` with key/value pairs *value: 1050000*, *prev_value: 1000000*, *change: 50000*.

If you combine the fact that *every player variable change causes an event to be posted* with the fact that you can set lots of things in your config files to be triggered by events, now you can start to see how this is used for game logic.

Some examples:

- You can configure a score display via a SlidePlayer entry which shows and updates itself based on the `player_score` event.
- You can configure an extra ball show to play based on the `player_extra_balls` event being posted with a change of 1.

Events posted by changing player variables will become even more important as we dig into game modes and logic blocks

Game Modes

In MPF, game modes are like little self-contained instances of an MPF environment that only apply when a particular mode is active. (Full details on game modes are [here](#).) In other words, you can add scoring, shots, timers, light shows, display effects, logic blocks, and many other things to a game mode, and when that mode is active, everything you specified in that mode is active. When that mode ends, everything you configured for that mode ends too.

The trick with game modes in MPF is that you can use them for a lot more than what you might think of as a "traditional" mode. And you can have lots of them running at the same time. The key is to break your game logic down into lots of little pieces, and then create a mode for each piece.

Also each mode can run at a different priority which affects the priority of display, lighting, and sound effects that come from that mode. Higher priority modes can also block lower priority modes from getting access to certain things. (Shots, for example.)

For example, we typically create a mode called "base" which we set to priority 100 which represents the base game mode. In it we'll configure all of our shots as well as our default scoring values for everything. We'll also configure default lighting, show, and sound effects.

Let's imagine we want to create a "super jets" game element where the pop bumpers are worth 100 points each, but after the player has 50 pop bumper hits then the "super jets" mode is enabled and the pop bumpers are worth 10,000 points each. To do that, we would configure the sound effect, lighting effect, and pop bumper score as part of our base mode.

We would also create a Counter logic block which was triggered by a tag added to each pop bumper. We would configure that logic block to start at zero, count up by 1 when a pop bumper was hit, not reset between balls, and finish when it got to 75.

Then we'd also create a game mode called "super_jets" which we would set to run at priority 200 and configure to start based on the event posted by our counter logic block finishing. We might configure a voice callout (via a SoundPlayer: entry) and a display effect (via a SlidePlayer: entry) which would play when the mode started. We would also configure a scoring entry for the pop bumpers to award 10,000 points per hit, and we would enable blocking so that scoring event was not passed down to the base mode. (Otherwise a pop bumper hit when our super_jets mode was active would be worth 10,100 points.) We could choose to keep this mode running when the ball ends or to reset it. If we want to make the second round to super jets require 75 hits instead of 50, we could configure a second logic block in the base mode which would start counting when the first logic block ended.

This of course is just one simple example, but it shows one way you can use modes to control game logic. (All without programming!) You can imagine similar modes for the skill shot, combos, extra ball lit and extra ball collected, progress towards multiball, multiball itself, and pretty much anything else you need to do.

Logic Blocks

Logic Blocks are the logical "glue" you use to create game logic and tie together shots, scoring, display, lighting, and sound effects. They're based on MPF events. They watch for an event (or events) to happen, and when they do, they post a "complete" event. Logic blocks can be configured to start over when they're complete, reset between balls, and enable and disable based on other events.

There are three types of logic blocks: *Sequence*, *Counter*, and *Accrual*.

- Sequence logic blocks watch for a sequence of events to happen in order. e.g. Event A, then B, then C, then D, then it posts the "complete" event.
- Accrual logics blocks watch for a bunch of different events to happen, but in any order. So once events A, B, C, and D happen, then the "complete" event is posted.
- Counter logic blocks just count (up or down) the number of times an event happens. So after event A happens 15 times, post the "complete" event.

You can read the documentation on logic blocks for the full details of how to use them, but in terms of using them for game logic, hopefully it's pretty clear how simple and powerful they are.

Some examples:

- To start a wizard mode, you could configure that mode's start event to be an accrual logic block which is based on the events posted by your 15 different game modes completing.
- You can configure a target to light based on accrual logic block which is configured for 3 different ramp shots made in the same ball.
- You can configure the a light extra ball mode to start based on the event posted by the bonus multiplier moving to 8x.
- You can configure progress towards starting a multiball mode with a counter logic block based on ball_locked events being posted.

Timers

Timers are kind of like logic blocks except they run automatically based on time rather than waiting for events to move them towards completion. You can configure timers (also via the config files) which start when a certain event is posted, then they post a "complete" event when they end. Timers can run up or down, and you can specify what the count is being ticks. You can also specify how fast they tick. The default is 1 second per tick, but most pinball machines use something we call "pinball time" where each "second" in a countdown mode is actually more like 1.5 or 2 seconds of real world time. You can also configure events which add, subtract, pause, restart, or reset timers.

So how do timers relate to game logic? Lots of ways:

- If you want to create a timed mode, you can add a timer entry to that mode's config. Then also configure the mode to stop when the timer posts its "complete" event.
- You can use timers in combination with counter logic blocks and a game mode to track "combo" shots:
 - Create a game mode called "combos" which has its start events based on a shot to a ramp or loop.
 - Within that mode, create a counter logic block that increments a combo count for each shot that's made.
 - Create a timer in the combo mode that counts down (3 seconds or whatever you want) and set the mode's stop event to be that timer completing. When the timer ends, the mode will stop and unload, along with its counter logic block and any scoring, lighting, or display effects you had.

- Since logic blocks are stored per-user, the next time the combo mode starts (based on the next ramp shot), the combo progress will be resumed. (Of course you can reset this per ball each time or whatever you want, if you want.)

"Real" programming

Even though we try to make it so you can do as much as possible with your machine config files, it's possible that you'll eventually come to a point where you have to do actual programming. (Though feel free to post to our [MPF users forum](#) first to ask. Hopefully we can find a way to do what you want to do or write the code for you.)

This section isn't meant to be a programming guide ([we do have that though](#)), rather, it's to help you figure out what how you might want to approach your custom programming.

First, the good news is that MPF was designed to be extended with custom programming. You don't have to "hack" or "break" anything to add some custom code.

Second, we've created a few different options for adding custom code, and they're both really easy to use. You don't have to be a Python master or understand anything about subclassing or method resolution order or PEP 20. You can just write some very simple code to do what you need to do and move on.

The main thing when it comes to adding Python code to your game is to decide whether your code is mode-specific or machine-wide. If it's mode-specific, you'll add it to your mode's `code` subfolder, and if it's machine-wide, you'll create a [scriptlet](#).

Examples of machine-wide scriptlets:

- In *Judge Dredd*, the crane which unloads balls from the Deadworld orbit is a scriptlet to replace the Deadworld ball device's `eject()` method.
- Same for the Cryo-Claw in *Demolition Man*.
- In *Star Trek: The Next Generation*, a scriptlet is used to pre-stage the required number of balls in the VUKs after a game ends. (1 ball in the left VUK, 1 in the left cannon VUK, 1 in the right cannon VUK.)

Game Modes

Using modes is a key concept to your game programming.

(more to come...)

Managing Assets

An "asset" is an external file that's used in your MPF game. Examples of assets include sound files, images, animations, etc. MPF includes a generic asset manager which is responsible for finding, loading, processing, and unloading assets. Multiple instances of this asset manager are setup (automatically, in the background) depending on what type of assets and modules you use in MPF.

Currently MPF sets up asset managers to manage the following types of assets:

- Images
- Animations
- Movies
- Sounds
- Shows

The asset managers are tightly integrated with the game modes, meaning you can specify assets on a per-mode basis.

For each asset (or type of assets), you can configure options that control when that asset is loaded. For example, some assets that are used throughout the game might be pre-loaded when MPF boots. Other mode-specific assets might be loaded when a mode starts and then unloaded when a mode ends. You can also configure assets so that they're loaded "on demand" (i.e. they're only loaded when they're needed and then unloaded as soon as they're done.)

Each asset manager runs in a background thread, meaning that your game still runs while the asset managers are loading assets in the background. (For example, you might choose to pre-load a mode's intro animations, sounds, and light shows so they're instantly available when a mode starts, and then while the intro animation is playing for that mode, the asset managers could load the remaining assets that mode needs in the background while the intro animation is playing.)

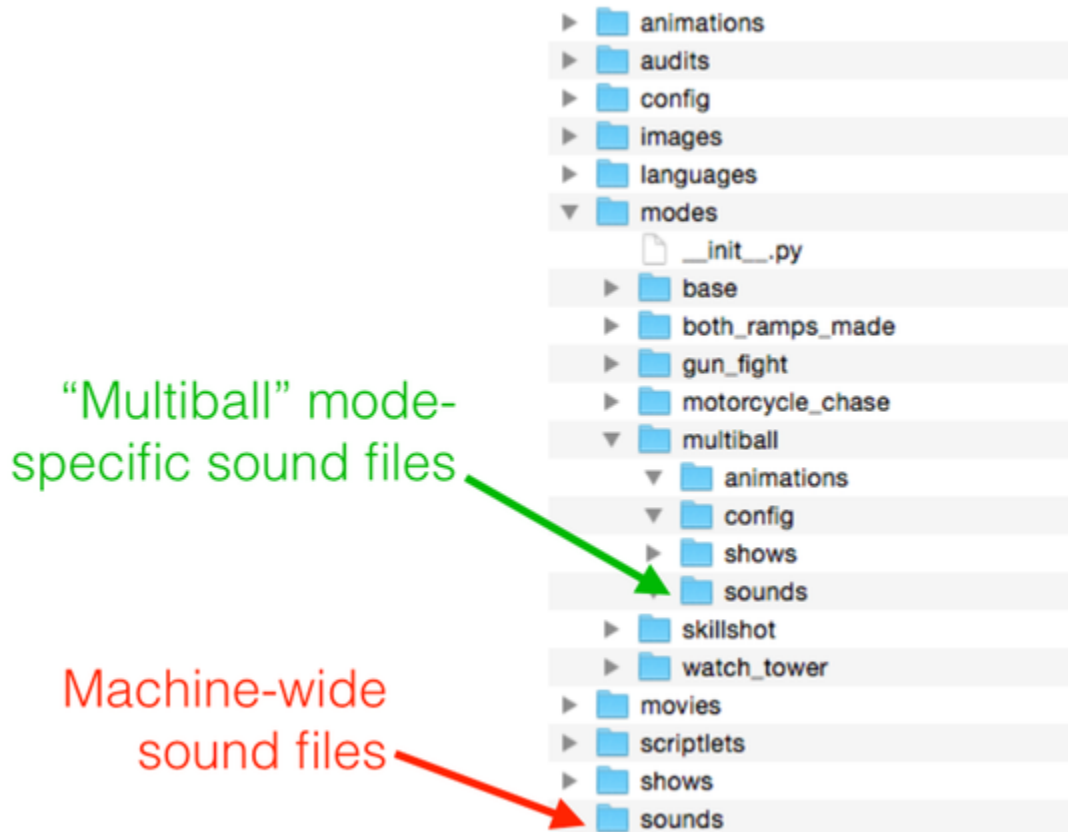
MPF gives you complete control over how assets are loaded. If your machine has plenty of memory and you don't want to monkey around with all this dynamic loading and unloading of assets, that's fine. You can configure your defaults so that every asset in the game is pre-loaded as the machine boots. Or if you're on a memory-constrained system (like a BeagleBone Black), you can get fancy with dynamic loading and unloading.

Asset file locations

Each type of asset is stored in a subfolder which matches the name of the asset type. (The folder names can be overridden if you like.) For example, sound files are stored in a "sounds" folder, animations are in an "animations" folder, etc.

These asset folders can exist in two locations, either in a mode's specific subfolder (for mode-specific assets), or in the root of your machine folder (for game-wide assets). For example, if your game is called "Demonstration Man," you would store machine-wide sounds in the `machine_files/demo_man/sounds/` folder, and then you would store sounds for the multiball mode in the `machine_files/demo_man/modes/multiball/sounds/` folder.

Here's an example from within a machine's folder:



Configuring default asset settings

Each different type of asset in MPF has settings specific to that type of assets. For example, animations have settings like how many frames per second they play, whether they should loop, whether they should be converted to 16-shade mono to show on a DMD, etc. Sound assets have settings like what track they should play on (e.g. voice versus sfx versus music).

Other settings, like whether an asset should be pre-loaded when MPF boots, whether it should load when a mode starts, or whether it should load on demand apply to all types of assets.

You can create an `AssetDefaults:` section in your config file to control which default settings apply to specific types of assets. Further, you can apply default settings on a per-folder basis within each asset type's folder. For example, you could configure sound assets in a "voice" folder so they play on the voice track, and sound files in an "sfx" folder so they play on the sound effects track. This means you can control the settings of assets simply by putting the assets in the proper subfolder!

You can also further customize the settings for a specific asset by creating an entry for that asset in your config file. This is completely optional, and in general you probably won't have to do that for every single asset. But maybe you have a few sounds here and there that are at a different volume than the rest, so in that case you can create configuration entries in your config file to set those specific sounds to play at a lower volume. The rest of their settings will be controlled based on what folder they're in and what `AssetDefaults:` settings you have applied to those folders.

In other words, *you do not have to create configuration entries for every single asset file in your game*, because there will probably be hundreds of them and that would suck. You can specify the settings for your assets on a folder-by-folder basis, and then just override the defaults here and there as you need.

Asset loading versus registration

When MPF boots, it will scan through your asset folders (both in the root of your game and in all your modes folders) to find and register all the asset files so they're available for use. Then it will look at the `AssetDefaults:` settings for that folder and any asset-specific settings to figure out whether it should actually load the asset into memory on boot, whether it should wait until a mode starts, or whether it should just load the asset on demand as its needed.

MPF differentiates between the process of "registering" and asset versus "loading" and asset. The registration process is how MPF knows that an asset exists. It says "there is a file called 'slingshot.ogg' in the /sounds folder, and it should be played on the track called "sfx." Registration happens when MPF boots for every asset in your game. The registration process doesn't take up much memory, so it's ok to have hundreds or thousands of assets registered at all times.

The actual "loading" process is when MPF reads the asset file from disk into memory. The exact way this works (and how much memory it consumes) varies depending on the type of asset. For example, an audio file that is 44.1KHz, 16-bit, stereo, will require $44100 \times 16 \times 2 = 1,411,200$ bits = 176 kb of memory per second of audio. If you have an hour of sound in your game, it could require over 600MB of memory to hold all those sound files in a state where they're ready to play instantly. So that's why MPF has the ability to dynamically load and unload assets, and why the loading process is separate from the registration process.

Asset loading times versus asset availability

Because all assets in MPF are registered when MPF boots, any asset is available to be called at any time. However the registered asset must have its file loaded into memory before it can be used. As we mentioned already, you can control when assets load their files into memory. But what happens if an asset is called when its related file has not been loaded into memory? In that case the asset manager for that type of asset will fetch the asset file from disk (via a background thread) and load it into memory right away. (If there's a queue of assets loading, it will jump to the front of the queue.) Then once it's loaded, the asset manager will notify whatever called for that asset to let it know the asset is ready to be used.

So how long does this loading process take? It depends on a several things, including:

- How big is the asset file?
- How much processing has to happen to get the asset file ready to use? (Decompressing, decoding, etc.)
- How fast is the disk? (Magnetic hard drive versus SSD, etc.)
- How fast is the computer? (BeagleBone Black versus 2GHz x86, etc.)

You also have to take into consideration that most types of assets aren't critical at the single-digit millisecond level. For example, when a ball hits a pop bumper and you want to play a sound, you probably want that sound asset preloaded so it can play instantly. But if the player gets an extra ball and you need 100ms to load the extra ball animation, voice callout, and light show, no one will ever notice.

You'll have to do some experimentation to figure out the right balance of preloading, loading per mode, and on-demand loading of your assets. You may find that you have plenty of memory to just preload everything, or you may find that your assets load fast enough that you can just set them to all be on-demand. It will really depend on your exact situation.

8. Advanced Programming

One of the goals of the Mission Pinball Framework is that we want to get as much machine configuration as possible into the [machine configuration files](#). It's our intention that maybe 80-to-90% of a machine's "code" can be built out of the box just by configuring those files. And so far we're pretty close. In addition to controlling switches, lights, and coils, you can also use the configuration files to set up shots, scoring, audio, DMD animations, events, and even game modes. And you can further extend MPF without programming with plugins for things like ball search, ball save, valid playfield, keyboard interface, LCD displays, coins and credits, and lots of other things—again all without having to write a single line of Python code.

That said, we recognize that to "finish" a game, you'll have to write some custom Python code. Our goal is that these code snippets should be easy to write and test, and (most importantly), they should not require you to "hack" the framework code itself. (Not that we have a problem with hacking—we love it! The problem though is if you hack the MPF code then it's really difficult to reintegrate your hacks with the official builds which are released every few weeks.)

The decision as to when to use config files versus when to use custom code is not a black-or-white decision. Really it comes down to what you want to do, what examples are out there, and how comfortable you are with Python. Some people who love coding in Python just use MPF config files to get the bare bones game up and running, while other people choose to do as much as they can in config files and to keep the custom coding to a bare minimum.

Regardless of where you fall on that spectrum, we tried to build MPF to make it as easy as possible to mix in actual Python code. We also created a few Python classes that hide a lot of the complexity of MPF, meaning you can get your custom code integrated with just some basic Python knowledge.

There are two easy ways to add custom Python code to your game project:

- **Scriptlets**, which are "machine-wide" chunks of code that are always active.
- **Mode-specific code**, which allow you to write custom Python code which is only active when a particular game mode is active.

As for when to use scriptlets versus mode-specific code, we tend to use scriptlets for any custom code that will be used through MPF (either in multiple game modes or when a game is not running). For example, we used scriptlets to hold the Python code to control the Claw in *Demolition Man*, the cannons in *Star Trek: The Next Generation*, and the Dead World lock and crane unloader in *Judge Dredd*.

Documentation of the MPF API

MPF's API (which is how you access various aspects of MPF via Python code) is documented via [Sphinx](#)-based documentation generated from the docstrings in the code. It's available [online](#) or via [PDF](#). (Note the API reference is different than the documentation you're reading now.)

You can also access the generated HTML pages via the [gh-pages branch](#) of the MPF repo on GitHub.

Scriptlets

A scriptlet is a standalone piece of Python code you write which provides specific custom functionality for a specific pinball machine. You put your scriptlet files in the `/machine_files/<your_machine>/scriptlets` folder, and then you add a reference to them in your machine configuration files. Then when you run your machine code, the scriptlets are loaded.

What's cool about the scriptlets is that they can interact with the MPF in a number of different (and deep) ways, none of which require actual hacking of the MPF code. For example, you can use scriptlets to:

- Post events which other MPF components (or other scriptlets will act on).
- Act on events that the MPF posts. You can even use scriptlets to "hook" boolean and queue events, which means you can insert your scriptlet into the game or ball start or stop process, the player add process, whatever...
- Register Tasks so your scriptlet can get control every game loop, or they can yield (sleep) as needed and wake up when they have more work to do.

The intention of scriptlets is that they're for machine-specific stuff. If you want to write something that's more generic that you would use across multiple games (like if you want to write your own Match routine or coin handling module), then you would write those as plugins instead. (Though really scriptlets and plugins are very similar.)

Creating a scriptlet

The easiest way to understand how to create a scriptlet is probably to go through an example of creating one. We'll eventually post lots of them here as examples, but for now let's start with one.

This scriptlet is for our *Demo Man* machine sample game. It turns on the GI (general illumination) lights during attract mode, and then makes sure that all the playfield lights are off before a game starts

```
# Attract mode Scriptlet for Demo Man

from mpf.system.scriptlet import Scriptlet

class Attract(Scriptlet):

    def on_load(self):
        self.machine.events.add_handler('machineflow_attract_start',
self.start)
        self.machine.events.add_handler('machineflow_attract_stop',
self.stop)

    def start(self):
        for gi in self.machine.gi:
            gi.on()

    def stop(self):
        for light in self.machine.lights:
            light.off()
```

When you write a scriptlet, you put all your code in a class which you subclass from `Scriptlet`. The class name can be whatever you want, and it doesn't matter if it you use a class name that's used anywhere else in MPF. You can put multiple classes in a single scriptlet file or put one class per file—it really doesn't matter. Since this scriptlet is for the attract mode, we've named the class `Attract`.

Next you create a method called `on_load()`. This method will be run automatically (one time only) when the scriptlet is loaded, so you use it to set up your scriptlet.

In our example, we use the `on_load()` method to register event handlers which will cause our other methods to run when MPF posts certain events.

You'll see that we register the scriptlet's methods `self.start` and `self.stop` for the event notification of the attract mode stopping or starting. The actual method names are arbitrary. We just picked `start()` and `stop()` since they make sense. But we could have called them whatever we want.

In our example, our `start()` method will be called when MPF posts the `machineflow_attract_start` event which will loop through all the GI strings and turn them on. And when the Attract mode ends (either because a player started a game or because the operator went into the service menu), MPF will post the `machineflow_attract_stop` event which will call our `stop()` method which will turn off any lights that are currently on.

Saving your scriptlet

We saved our scriptlet with a file name `attract.py` and put it in the scriptlet folder with our other *Demo Man* machine files.

Adding your scriptlet to your game

The final step is to "install" your scriptlet into your game. You do this in the machine configuration files, in a section called `scriptlets:.` Just add the file name, then a dot, then the class name, like this:

```
scriptlets:
    attract.Attract
```

You can add as many as you want, one per line.

So that's pretty much it! When you run your game code, MPF will automatically import your scriptlets and then run the `on_load()` method from each one. Everything else they do is up to you!

Another Scriptlet example

We include another scriptlet example in the `/machine_files/new_machine_template/scriptlets` folder called `'new_scriptlet_example.py.'` This example scriptlet doesn't really do anything useful, (though it is fully functional), rather, it shows off some different things you can do in a scriptlet in addition to playing some light shows:

```
"""Example scriptlet which shows different things you can do."""

from mpf.system.scriptlet import Scriptlet # This import is required

class YourScriptletName(Scriptlet): # Change "YourScriptletName" to
    whatever you want!
    """To 'activate' your scriptlet:
    1. Copy it to your machine_files/<your_machine_name>/scriptlets/
    folder
    2. Add an entry to the 'Scriptlets:' section of your machine config
    files
    3. That entry should be 'your_scriptlet_file_name.YourScriptletName'
    """

    def on_load(self):
        """Called automatically when this scriptlet is loaded."""

        # add code here to do whatever you want your scriptlet to do when
        the
        # machine boots up.

        # This example scriptlet has lots of different examples which show
        (some)
        # of the things you can do. Feel free to delete everything from
        here on
        # down when you create your own scriptlet.

        # you can access the machine object via self.machine, like this:
        print self.machine
```



```

print self.machine.physical_hw
# etc.

# you can access this scriptlet's name (based on the class name
above):
print self.name # will print "YourScriptletName" in this case

# you can write to the log via self.log:
# The logger will be prefaced with Scriptlet.YourScriptletName
self.log.info("This is my scriptlet")
self.log.debug("This is a debug-level log entry")

# you can access machine configuration options via
self.machine.config:
print self.machine.config['Game']['Balls per game']

# feel free to add your own entries to the machine configuration
files,
# like: self.machine.config['YourScriptlet']['Your Setting']

# you can post events which other modules can pick up:
self.machine.events.post('whatever_event_you_want')

# you can register handlers to act on system events
self.machine.events.add_handler('ball_add_live_success',
                                self.my_handler)

# you can create periodic timers that are called every so often
from mpf.system.timing import Timer
self.machine.timing.add(Timer(callback=self.my_timer, frequency=10))
# (Or save a reference to the timer if you want to remove() it
later.)

# you can register a handler for the machine tick which will be
called
# every machine tick!
self.machine.events.add_handler('timer_tick', self.tick)

# you can create a task that can yield as needed

def my_handler(self):
    # This is just an arbitrarily-named method which is the handler for
    # `ball_add_live_event` from the on_load(). Feel free to create as
    # many methods as you want in your scriptlet!
    print "A new ball was added"

def my_timer(self):
    print "another 10 seconds just passed"

def tick(self):
    # this will run every single machine tick!!
    pass

```

You can run this scriptlet as is by copying it into your `/machine_files/<your machine>/scriptlets` folder and then adding ``new_scriptlet_example.YourScriptletName`` to the Scriptlets section of your machine configuration files.

New scriptlet template file

We also include a blank scriptlet template file in the `/machine_files/new_machine_template/scriptlets` folder called `'new_scriptlet_template.py.'` This is the file we use as the starting point to create our own scriptlets:

```

"""Template file you can customize to build your own machine-specific
scriptlets.
"""

from mpf.system.scriptlet import Scriptlet

class YourScriptletName(Scriptlet): # Change "YourScriptletName" to
whatever you want!
    """To 'activate' this scriptlet:
    1. Copy it to your machine_files/<your_machine_name>/scriptlets/
folder
    2. Add an entry to the 'Scriptlets:' section of your machine config
files
    3. That entry should be 'your_scriptlet_file_name.YourScriptletName'
    """

    def on_load(self):
        """Called automatically when this scriptlet is loaded."""
        pass

```

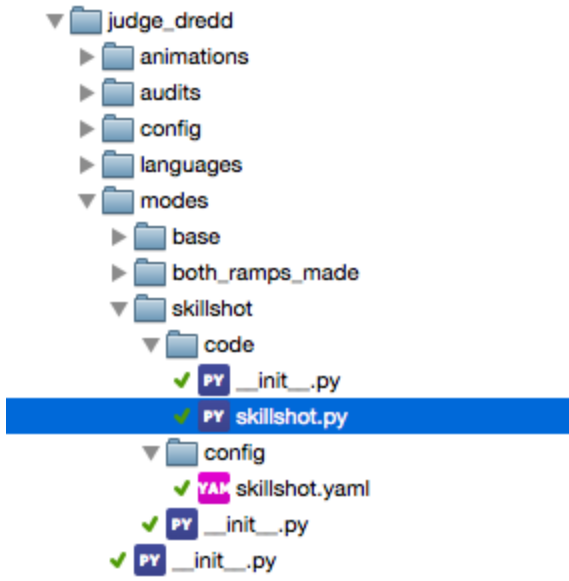
Mode code

In MPF, you can add custom code to specific game modes. This code is conceptually similar to scriptlets, except that it's mode-specific. In other words, the code in a scriptlet runs and is active always, whereas mode-specific code is only active when that particular game mode is active.

To add mode-specific code:

1. Create a `code` subfolder in your that mode's `mode` folder.
2. Create a file in the code folder to hold your game code called `<mode_name>.py`.
3. Create an empty file called `__init__.py` in your code folder. (That's two underscores, then "init", then two more underscores, then ".py".)
4. In your mode config file (in your `<mode_folder>/config/<mode_name>.yaml` file, add a setting in the `mode:` section for your mode. For example `code:`
`skillshot.SkillShot.`

For example, here's what the folder structure would look like to create some custom code for the skillshot mode:



And here's what the `skillshot.yaml` mode config file would look like:

```
mode:
  start_events: ball_starting
  stop_events: timer_mode_timer_complete
  code: skillshot.SkillShot
  priority: 300
```

To actually start writing code, you need to subclass MPF's `Mode` class. Here's an example of the bare minimum in need in your `skillshot.py` file using the `skillshot` mode example from above:

```
from mpf.system.modes import Mode

class SkillShot(Mode):
    pass
```

Note that the name of your subclass here matches the name of the class you specified in the `code:` section in the mode's config file.

From there, you can add whatever methods you want. The `Mode` super class contains three methods which are designed to be overwritten in your own mode-specific code:

- `mode_init()` - runs on MPF boot when the mode is read in and set up.
- `mode_start()` - runs when the mode starts
- `mode_stop()` - runs when the mode stops

There's also another helper method in the parent class called [add_mode_event_handler\(\)](#). This method is used exactly like the regular event modules

[add_handler\(\)](#) with one addition: Any event handlers you register here will automatically be removed when the mode stops. Using this method (instead of the regular one) in your mode code is completely optional, but it makes things a bit cleaner since you don't have to track and remove mode-specific event handlers on your own.

MPF Coding Best Practices

These will be fleshed out. For now this is just a list so we remember what to write about.

- For events, try to move on quickly. If anything is long-running then use a callback and come back later.
- Don't mix direct calls and events.
- Remember that there's an event queue and events are handled sequentially in the order they're posted, so events are *not* guaranteed to be handled when they're posted. If you need something to fire after an event, you need to either make it fire by an event or call it via a callback.
- Don't make the valid playfield count too high, since it's disruptive for a ball to be plunged and to score without a valid playfield.

9. Tools

The Mission Pinball Framework includes several tools (in the `/tools` folder) which help you when you're creating your machine. Current tools include:

[subpages]

Future Tools

We'll be creating more tools in the near future to help with other things, including:

- Configuration file validator.
- Conversion tools to convert images to DMD files.
- DMD file players.
- Graphical visualizers for machine states, light shows, etc.
- GUI configuration tools
- Plus whatever else you need. (Let us know in the forums!)

Config File Migrator

MPF includes a config file migration tool that can help you migrate your MPF configuration files to the latest config version. (Read [here](#) for details on what the config file version is.)

To use the tool, switch to the `tools` folder in your MPF package and run it from the command line, specifying the location of your machine folder as a command line argument, like this:

```
>python config_migrator.py c:\brian\machine_files\sttng
```

You should see results that look something like this:

```
Migrating MPF config files from v1 to v2

Migration Results
=====
Backup location for existing files: c:\brian\machine_files\sttng\
previous_config_files\2015-07-27-16-45-40
Files migrated successfully: 9
Files skipped: 20
Files that require manual intervention: 2

Open up each of these files to see the details of the sections you
```

```

need to manually update:
c:\brian\machine_files\sttng\config\game_logic.yaml
c:\brian\machine_files\sttng\config\hardware.yaml

```

The config migrator will automatically scan through the folder (including subfolders) looking for `.yaml` files. If it finds them, it will open them up and make sure that they are the previous version, and, if so, it will make changes to them. In other words, if the current MPF config file version is 2, then the migrator tool will only attempt to migrate files that have `#config_version=1` as their first line. The tool will not touch light or display show files, and it will not touch config files whose version already matches the latest version of MPF.

The migration tool automatically creates a backup of each file it makes changes to in a folder called `previous_config_files` in the root of the path you specified. In there it creates another folder based on the date and time the tool was run, and that folder contains the original versions of the `.yaml` files the migrator tool made changes to.

Command line options

`-f` force the re-migration of already migrated files. (This option means that the migration tool will also target files that have the latest `config_version`, rather than the default of only targeting files that are the previous version.)

What does the config migrator actually do?

The config migrator tool can do three things.

First, it does simple "find and replace" changes to section names that have changed. For example, many section names changes between config version 1 and version 2—`Autofire Coils:` was renamed to `autofire_coils:`, `BallDevices` was renamed to `ball_devices:`, etc. These changes are pretty easy and safe for the tool to do automatically. (Note that even though section names in the MPF config files are not case sensitive, any renamed sections will be changed to lowercase.)

Next, the migrator tool can identify any sections of the config file that have been deprecated (removed). For example, early versions of MPF included a section called `plugins:`, but in current versions, plugins are loaded automatically and do not need to be included in the config file. So if the migrator tool finds a `plugins:` section in a config file, it adds a comment into the file letting you know that you can safely remove that section.

Finally, the config migrator searches for sections that have major changes that will require manual intervention. In these cases since the changes are major, the tool cannot automatically make the changes for you. When files containing these sections are found, the tool will add some text to the top of the `.yaml` file explaining what section has changed, and it gives you a URL where you can get more information about the new way that section

works. In these cases, the warning text that the migration tool adds to the `.yaml` file will essentially "break" that file (since adding that text means the first line in the file is no longer `#config_version=x`. This was done on purpose to ensure that you can't accidentally run MPF with outdated sections.

As you can see in the example output from running the migration tool above, it prints a summary of the results on the screen. Files migrated successfully: 9 means that 9 files are completely migrated and ready to go. They have the latest `#config_version` entry and all the find-and-replace for renamed sections is done. Files skipped: 20 means that the migration tool found 20 files that were either light show or display show files, or they were files that were either too old or too new to migrate automatically. (In the future we'll add a support to migrate through multiple versions at a time.) Finally, the Files that require manual intervention: 2 result means that there were 2 files with sections that need manual changes. Then you'll see the list of those files. In this case, they're `game_logic.yaml` and `hardware.yaml`. Let's take a look at `hardware.yaml` and see what the migration tool did:

```
# -----
# MIGRATION WARNING:
# This file contains a "targets" section which underwent major changes in
# config_version=2.
# You will have to read the docs and re-do this section.
# New documentation for this section is here:
# https://missionpinball.com/docs/configuration-file-reference/targets/

# When you're done, delete this message so #config_version=2 is the first
# line in this file.
# -----

#config_version=2

machine:
  hz: 60
  ...
```

Notice that the text at the top of the file explains that it's the "targets" section of this file which needs to be manually updated, and it also provides a link to the new targets documentation. Next we can use CTRL+F to search the file for "targets" which will reveal this section:

```
...
Targets:
  t_topLaneLeft:
    switch: s_topLaneLeft
    light: l_topLaneLeft
  t_topLaneCenter:
    switch: s_topLaneCenter
    light: l_topLaneCenter
  t_topLaneRight:
    switch: s_topLaneRight
    light: l_topLaneRight
  t_leftBankTop:
```

```

    switch: s_leftBankTop
    light: l_leftBankTop
t_leftBankMiddle:
    switch: s_leftBankMiddle
    light: l_leftBankMiddle
t_leftBankBottom:
    switch: s_leftBankBottom
    light: l_leftBankBottom
t_rightBankTop:
    switch: s_rightBankTop
    light: l_rightBankTop
t_rightBankMiddle:
    switch: s_rightBankMiddle
    light: l_rightBankMiddle
t_rightBankBottom:
    switch: s_rightBankBottom
    light: l_rightBankBottom
...

```

From there you can check the link and update that section as needed.

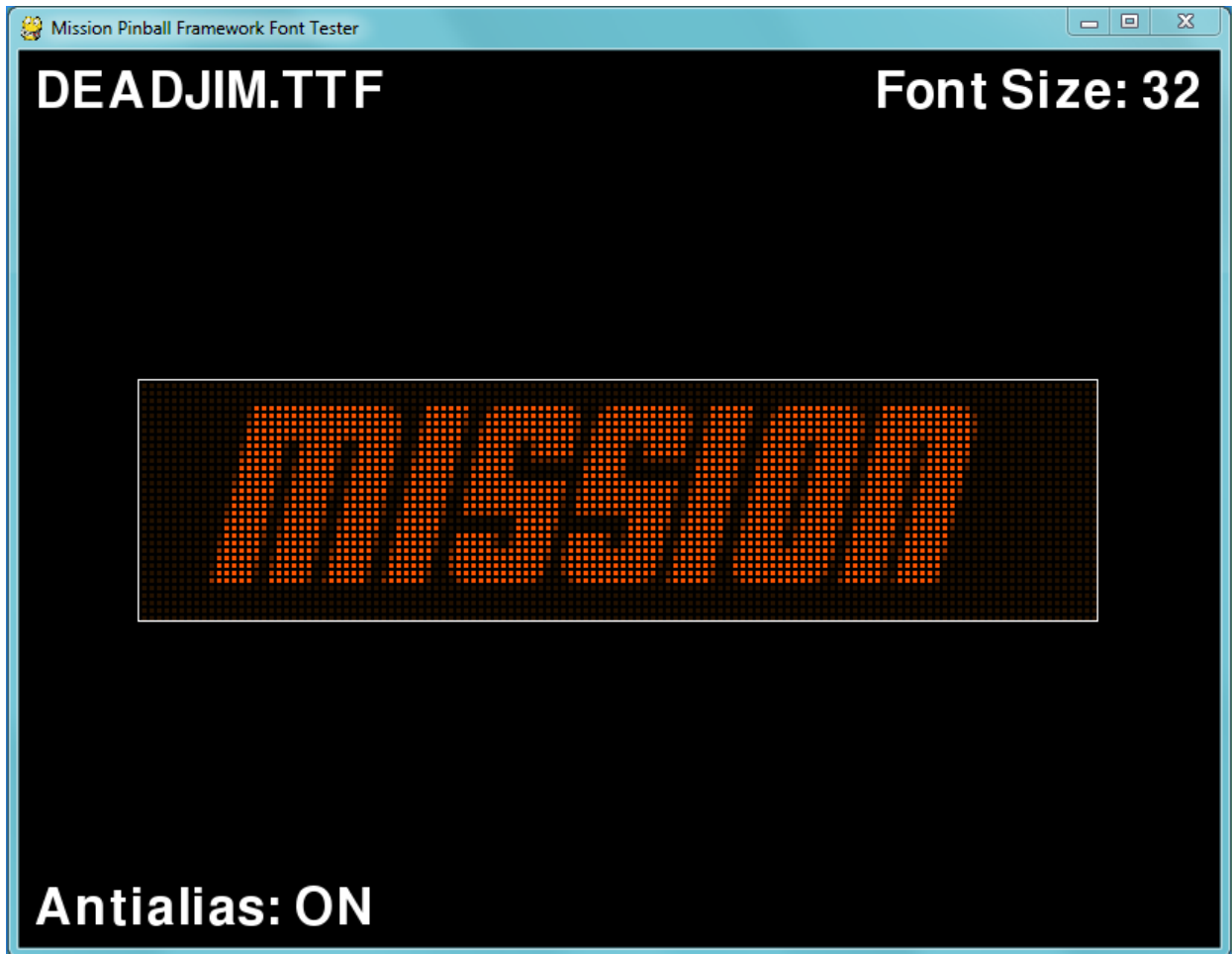
When you're done, be sure to delete the block of warning text at the top of the config file so that the `#config_version=x` is the top line.

Font Tester

The MPF Font Tester (``/tools/font_tester.py``) is a graphical tool that lets you view how TrueType fonts will be rendered on your DMD. You run the tool by specifying the path to a folder which contains TrueType fonts. Anything you type shows up in the on screen DMD preview box. You can use the left and right arrow keys to flip between fonts, and you can use the up and down arrow keys to increase or decrease the font size. There are also shortcuts to enable and disable antialiasing, to vertically shift the font, to highlight the font's bounding box, and to save snapshots of the screen for later use.

There's a [how to guide covering adding TrueType fonts to your game](#) which makes extensive use of this Font Tester tool.

Here's what the Font Tester looks like in action:



We also recorded a ten-minute YouTube video which demonstrates how the font tester is used:

<https://www.youtube.com/watch?v=xKuR80tUJhA>

Running the Font Tester

Run the font tester tool from the command line by specifying the path to a folder which has TrueType font files (.ttf) you want to test. You can optionally specify a file name too which is the first font it will load. Otherwise it just starts at the beginning of the alphabet. For example:

```
python font_tester.py c:\Windows\Fonts
```

or

```
python font_tester.py ../mpf.fonts/pixelmix.ttf
```

Instructions

Once the window pops up, it will show a virtual DMD in the center with your initial text and the initial font.

- To change the text that's rendered, just start typing. You can do uppercase and lowercase letters, and numbers.
- Use the `Left` & `Right` arrow keys to cycle through the different fonts in the folder.
- Use the `Up` & `Down` arrow keys to increase & decrease the size of the font.
- Use `SHIFT + Up` and `SHIFT + Down` to vertically adjust the text placement in the DMD window.
- `CTRL+A` (or `CMD+A` on a Mac) toggles Antialias mode
- `CTRL+B` (or `CMD+B`) toggles showing the green Bounding Box.
- `CTRL+S` (or `CMD+S`) takes a snapshot of the screen and saves it to the 'font_snapshots' folder.

Limitations

- Only works with TrueType fonts with .ttf extensions.
- Doesn't work with shift symbols. (i.e. SHIFT+1 shows "1" and not "!")

Configuration Options

There are lots of configuration options available for the Font Tester which you can easily access by editing `font_tester.py`. All these settings entries are in the beginning of the file and easy to find.

- `window_size = (800, 600)` - The size (in pixels) of the main window.
- `dmd_size = (128, 32)` - The pixel size of the native DMD.
- `dmd_screen_size = (640, 160)` - The pixel size of the on screen DMD.
- `pixel_color = (255, 85, 0)` - R, G, B colors of the font pixels.
- `dark_color = (34, 17, 0)` - R, G, B colors of the 'off' pixels.
- `pixel_spacing = 2` - Pixel spacing between dots.
- `loop_ms = 100` - How many ms it waits per cycle.
- `font_bg_color = (0, 200, 0)` - R, G, B color of the CTRL+B box.

- `max_chars = 10` - How many characters are displayed.
- `snapshot_folder = "font_snapshots"` - Path of the CTRL+S screenshots.
- `prefer_uncompressed_snapshots = False` - Do you want uncompressed BMPs?.
- `snapshot_flash_brightness = 255` - Color of the snapshot flash.
- `snapshot_flash_steps = 5` - Steps for the flash to be done.
- `text_string = "HELLO"` - Initial text.
- `font_size = 10` - Initial font size.
- `antialias = False` - Initial antialias setting.
- `bounding_box = False` - Initial bounding box settings.

OSC Sender

`osc_sender.py` is a command line tool which sends OSC commands to an OSC server, such as the Mission Pinball Framework with the OSC plug-in. (It will also work with pyprogame with the [OSC mode Brian created](#) for it.) It's located in the MPF download package at `/tools/osc_sender.py`.

This tool was originally created in response to a request on the [PinballControllers.com forum](#) for someone who wanted to send switch events to his pinball machine via a web page, but really the sky's the limit with this thing since you could script it or tie it into whatever you want.

You can read a list of the various OSC addresses MPF uses in the documentation section about [configuring OSC clients](#).

To use it, pass two parameters. First is the OSC address (the thing with slashes, *not* the host IP address), and then the data, like this:

```
python osc_sender.py /sw/start 1
```

By default this utility will also send a second OSC message with "0" as the data to the same address after the first one is sent. Otherwise you'd have to use it twice to activate the switch and then deactivate the switch. If you want to not do this (like if you want to set a switch and then keep it set), then add a `-t` command line option to enable "toggle" mode.

Also this will pull the local IP address via Python, but that might not be the IP address you want to use (like if you're using it on a network or if there is more than one IP and it's grabbing the wrong one). In that case use the `-s` command line option to specify an IP address (like `-s 10.0.1.14`).

This utility will use the default port of 8000 as the target port on the OSC server it's looking for. Use the command line option `-p` to specify a different port.

This OSC Sender is generic and not pinball specific at all. It just sends your OSC address and data to whatever OSC server you want, then exits. Run it as often as you want.

10. Configuration File Reference

As we mentioned in the introduction, MPF uses the YAML configuration files to configure itself for just about everything your machine does. This section of the documentation is the configuration file reference which has all the detailed options about how all these files work.

There are two types of config files: *machine config files* and *mode config files*.

- **Machine config files** are stored in the `<your_machine_folder>/config` folder. Anything specified there applies to your entire machine and are active for the entire duration of MPF running.
- **Mode config files** are stored in the `<your_machine_folder>/modes/<mode_name>/config` folder. They are activated when the mode starts and deactivated when the mode ends.

Not all settings are usable in the mode config files. Refer to the details in each entry for details.

Before we jump in, take a look at a made-up example of the types of things you might find in your config files.

```
hardware:
  platform: p_roc
  driverboards: wpc

switches:
  flipperLwR_EOS:
    number: SF1
  flipperLwR:
    number: SF6

coils:
  flipperLwRMain:
    number: FLRM
  flipperLwRHold:
    number: FLRH

flippers:
  LowerRight:
    main_coil: flipperLwRMain
    hold_coil: flipperLwRHold
    activation_switch: flipperLwR
    eos_switch: flipperLwR_EOS
    label: Right Main Flipper
```

Remember you can have as many configuration files as you want (for both the machine and mode specific config files. MPF will read them all in and merge them into a single master list.

So whether you have all these sections in a single huge file called *my_game.yaml* or whether you break them into *coils.yaml*, *flippers.yaml*, and *switches.yaml* is up to you.

Important config file concepts

Before you start editing your configuration files, there are several important concepts that you should be familiar with. These concepts apply to all config files, including machine-wide and mode-specific configs:

[subpages]

Case-sensitivity in config files

One change with MPF 0.17 / config_version 1 is that settings in config files are no longer case sensitive. This was done to prevent confusion as people would typically miss case settings.

For example, in MPF prior to version 0.17, the following would work:

```
SlidePlayer:
```

While the following would not:

```
Slideplayer:
```

What happens internally starting with MPF 0.17 is that all the settings (i.e. the "keys" of the key/value pairs in config files) are converted to lowercase internally.

We believe that this should not be a problem and in fact should be transparent to most game programmers. We also have attempted to make sure that all functions that reference objects that are set up in the config files also convert their references to lower case. For example, if you have a coil defined like this:

```
Coils:
  flipperLeftMain:
    number: ...
```

And then later you refer to it via your code as

```
self.machine.coils['flipperLeftMain']
```

That will still work because the device collection object that holds the list of coils will convert the incoming request to lowercase.

All that said, there's one "gotcha" with this. The case insensitivity means that you cannot differentiate between devices bases solely on case differences.

For example, in versions of MPF prior to 0.17, it was perfectly valid to define lane lights based on uppercase letters. For example, the three lights that make up "W", "I", and "N" lanes could have been defined as `lane_Win`, `lane_wIn`, and `lane_wiN` previously. Starting with MPF 0.17, you'll have to differentiate them in some other way. (For example, `lane_win_w`, `lane_win_i`, and `lane_win_n`.)

We believe this change was for the better, but we're always open to options. If you believe we should change this behavior, please start a discussion in our [MPF Development forum](#).

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:

```
flipperLeft:
  number: SD18
  tags: flipper, player # this is a list
```

```
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
```

```
light_special:
  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 all the items on one line separated by spaces: (Obviously you can't do this if your individual items have spaces in their names. In that case, just use commas.)

```
config: item1 item2 item3
```

Third, you can enter each item on its own line, like this: (Be sure that you indent your list items, and that they are all indented the same amount.)

```
config:
  item1
  item2
  item3
```

Fourth, 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
  - item2
  - item3
```

So you have four 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
```

So in the example above, the configuration item has a list with three items. The first list item contains item1 and item 2, the second list item contains item3, item4, and item5, and the third list item contains item6.

You can also enter each item on it's own line and then use dashes to signify where a new list item starts, like this:

```
config:
  - item1
    item2
  - item3
    item4
    item5
  - item6
```

Note that the indentation of all your items is the same, but that the dash is "outdented".

How to configure "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*, targets have *enable_events*, *disable_events*, and *reset_events*, target 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 in [here](#).

All this is done via the config files with no custom Python code needed! :)

How to format time values in config files

Your 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, 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 any letters, then MPF will read in the time duration as milliseconds.

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 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)
```

#config_version

Starting with MPF 0.17, MPF expects that the first line of a YAML configuration file specifies what version of the MPF config it uses. For example:

```
#config_version=3
```

This line needs to be the very first line in an MPF config file.



This section *must* be included in your machine-wide config files.



This section

must be included in your mode-specific config files.



This section does not need to be included in "show" .yaml files.

Each version of MPF knows what version of the config files it will look for. If it tries to load a config file with the wrong version, it will post an error and exit.

The important thing to know about this is that not every new version of MPF will require a new version of the config files. So that's why MPF 0.17 and 0.18 both use config file version 1, MPF 0.19 uses version 2, etc. We did this so that game programmers don't have to go through all their config files and update their versions if MPF doesn't actually require any version changes.

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. Details on the tool and how to use it are [here](#).

Which versions of MPF require which config_versions?

- MPF 0.30: `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)

#config_version=4

(This page is a work in progress)

MPF 0.30 introduces `#config_version=4`. Here are the changes:

New Sections:

`coil_player`

`gi_player`

flasher_player

trigger_player

led_player

widget_player

Renamed Sections:

In all config_players, "tocks_per_sec" is removed, and a "speed" option is added (since show steps are based on time instead of tocks)

Removed Sections:

timing: hz

Changes:

Power settings for coils in the coil_player have been change to values from 0.0 to 1.0 instead of 0-100.

light_player split to led_player and light_player

All players

All "playable" configs are now available both in shows and slides

#config_version=3 changes

MPF 0.20 introduces *#config_version=3*. Here are the changes:

Renamed Sections:

Old name	New name
targets:	shots:
target_groups:	shot_groups:
_remove_profile_events	_remove_active_profile_events

autofire: switch_activity	autofire:reverse_switch
led: log_color_changes	led: debug
Events that started with <i>shot_</i>	Events now start with <i>target_</i>
attract_start	mode_attract_started
min balls	min_balls
balls per game	balls_per_game
max players per game	max_players
shot_profiles: steps	shot_profiles: states
step_names_to_rotate	state_names_to_rotate
state_names_to_not_rotate	step_names_to_not_rotate
Events that ended with <i>_remove_profile</i>	Events now end with <i>_remove_active_profile</i>
max balls	(removed)

Targets have been renamed to "shots"

Pretty much everywhere that you had the word "target" in your old config, that will now be "shot" in v3. This includes:

- The `targets:` section
- The `target_groups:` section
- The `target_profiles:` section
- Any events that began with *target_**

You'll need to find-and-replace and change these from *target* to *shot*.

Note that you *cannot* simply do a global "find and replace" for every instance of `target` to `shot` because there are a lot of places where the word `target` is used. For example, you can't change *drop_targets* to *drop_shots*, or a ball device's *eject_targets* to *eject_shots*.

Convert existing "shots" to new-style shots

Previous versions of MPF config files did have functionality for shots. That functionality has been rolled into the new *shots* sections. (Which is awesome, because that means that what used to be your old shots can now be combined into groups, control lights and LEDs with different profiles, track states when they're hit, etc. And it means that what used to be called targets now can have sequences of switches to hit them instead of only one switch.)

Because of this change, you'll have to manually go through your existing config files and convert & combine any of your old `shots:` sections with the new shots. Fortunately that's pretty simple.

For an old shot with one switch, there are no changes:

```
shots:
  right_ramp:
    switch: right_ramp_exit
```

If you had a section called "targets:" in your old file that's renamed to shots, make sure you combine what was the old shots section into the new shots section so your file only has a single section called shots.

For an old sequence shot:

```
left_ramp:
  type: sequence
  switches: left_ramp_enter, left_ramp_exit
  time: 3s
```

Remove the `type: entry` and change `switches: to switch_sequence::`

```
left_ramp:
  switch_sequence: left_ramp_enter, left_ramp_exit
  time: 3s
```

Note that you can apply all the benefits of what used to be called "targets" to your sequence shots now, including grouping them, rotation, light scripts with steps, hit events, etc.

The Configuration File Migration Tool has been updated

You can use the [config file migration tool](#) to scan your machine folder for all your YAML files and update them from v2 to v3. It can do pretty much everything except for dealing with your existing `shots:` sections, which it will mark for you to change. (So it will change everything *target*-related to shots and that should all work, but your existing *shots* entries will have to be manually converted to the new shots format. In general that shouldn't be too difficult.)

Drop targets are now separate from shots

In prior versions of the config, `drop_targets` could have profiles attached to them, and `drop_target_banks` were like `target_groups` with profiles.

In the new config, `drop_targets` are now "just" drop targets. They're a switch, a coil to reset them, and (optionally) a coil to knock them down.

`drop_target_banks` are just a group of `drop_targets` as well as (optional) bank-wide reset or knockdown coils.

So any of the profile-related settings (like *profile*, *advance_events*, *enable_events*, *disable_events*, etc.) are now removed from the `drop_targets:` and

`drop_target_banks`: sections of your config files. (Note that *reset* and *knockdown* events still apply.)

Of course you might want to still have the shot-like functionality with your drop targets, and that's still possible. The way you do that in the new config is you would just setup a shot based on the drop target's switch (and optionally its light or LED). So you can still manage shot profiles, states, advance them, get hit events, etc., but that all comes from shots you have assigned to the drop target switches, rather than to the drop target itself.

"Attract" and "Game" are now regular modes

Prior to `config_version=3`, there were two types of modes in MPF. Game modes were what you would think of as regular modes. They were only active while a game was in progress, and you configured them in your `modes` folder and created them for base, skillshot, multiball, etc. MPF also had what we called "machine modes" which was what the overall state that the machine was in, such as *attract* or *game*.

Now we have gotten rid of machine modes and changed the regular game modes so that they can also run when a game is not in progress. As part of this, we have converted the attract and game machine modes into regular modes.

We also made a change to MPF to add a `modes` folder to the `mpf` folder. So this means that there are now two modes folders, one at `mpf/modes` and one in `your_machine/modes`. When MPF runs, it will combine all the modes from both locations to apply to your game.

At this point, we only have game and attract modes in the `mpf/modes` folder, but in the future we envision creating "default" modes for tilt, volume_control, coin & credits, service menu, coin_door_open, etc.

You can customize the built-in modes by creating a folder for that mode with the same name in your own machine's mode folder, then you can add a mode config yaml file there which will be merged in with the mode's default yaml file.

We'll write more information about this in the future, but for now it's important to know that you will see attract and game modes running with this new version of MPF. (Attract runs at priority 1, and game runs at priority 2. Your own game modes should start at priority 100.)

Mode-specific code changes

If you have custom mode code in your game, you will have to make a few changes.

The name of the module that holds the parent `Mode` class has been renamed from "mode" to "modes, meaning you need to change this:`

```
from mpf.system.modes import Mode
```

To this:

```
from mpf.system.mode import Mode
```

Also any keyword arguments that were attached to an event that's used to start or stop a mode will now be passed to that mode's `mode_start()` and `mode_stop()` events, meaning you need (potentially) change your mode code so those methods can accept keywords. If you add `**kwargs` to those method definitions, that will act as a catch-all for any keywords that are passed that you might not be expecting. You can change it like this:

Old:

```
def mode_start(self):
```

New:

```
def mode_start(self, **kwargs)
```

Old:

```
def mode_start(self):
```

New

```
def mode_start(self, **kwargs)
```

#config_version=2 changes

In MPF 0.19, we have incremented the version number of the YAML configuration files that MPF uses. (We use the version number to make sure that the YAML files you've created are compatible with the latest version of MPF. More details [here](#).)

In order to use MPF 0.19 and newer, you will have to update your YAML files to version 2. We know this is a pain, but with the complete rewrite of the "target" code (standups, rollovers, drop targets, etc.) in 0.19, none of the existing target configuration will work. While we were at it, we decided to implement several other changes we had been wanting to do for awhile. (Mostly find-and-replace name changes.)

To make this update as easy as possible, we created a [config file migration tool](#) that scans your existing YAML files and auto-updates them as much as possible and then highlights the areas you'll have to change manually. There's one catch: if you haven't already added `'#config_version=1'` to your config files, do it now or they will be skipped (this includes your mode config files).

Target / Rollover / Standup / Drop Target config sections

(Note: this page is not done yet... so that's why it seems like random notes.)

targets, targetgroups, rollovers, rollovergroups, droptargets, droptarget groups: throw out everything you once knew

switch events are new

double_zeros is out, min_digits: 2 is how you'd do that now

tps in light scripts -> tocks_per_sec

movies -> videos

holdpatter

machine_flow_attract_start/machine_flow_attract_stop are now just called attract_start and attract_stop.

Other "find and replace"-type changes:

Over the past year of creating MPF, we've had a lot of variation of several config file sections. So we're redoing everything now to be consistent, and as such, the following sections have changed in config file version 2. You can do a simple find+replace across your YAML files for these changes.

Old	New
assetdefaults	asset_defaults
autofire coils	autofire_coils
balldevices	ball_devices
ballsearch	ball_search
droptargets	drop_targets
droptargetbanks	drop_target_banks
languagestrings	language_strings
lightplayer	light_player
lightscripts	light_scripts
logicblocks	logic_blocks
ledsettings	led_settings
machineflow	machine_flow
matrixlights	matrix_lights
mediacontroller	media_controller
openpixelcontrol	open_pixel_control
score reels	score_reels

score reel groups	score_reel_groups
showplayer	show_player
slideplayer	slide_player
soundplayer	sound_player
soundsystem	sound_system
switchplayer	switch_player
targetgroups	target_groups
virtual platform start active switches	virtual_platform_start_active_switches

accruals

Accruals are a type of Logic Block where you can trigger a new event based on a series of one or more other events. They are a key part of implementing game logic in MPF.



This section can be used in your machine-wide config files.



This section can be used in your mode-specific config files.

Here are the Logic Blocks that we use for our Big Shot pinball machine:

```

logic_blocks:
  accruals:
    light_special:
      events:
        - sw_eightball,
        - drop_targets_solids_lit_complete,
drop_targets_stripes_lit_complete
      events_when_complete: lighting_special,
action_target_specialright_light
      enable_events: ball_started, collect_special
      disable_events: ball_ended, lighting_special
      reset_events: ball_ended, lighting_special

    collect_special:
      events:
        - target_specialLeft_lit_hit, target_specialRight_lit_hit
      events_when_complete:
        collect_special
      enable_events: lighting_special
      disable_events: ball_ended, collect_special
      reset_events: ball_ended, collect_special

    lighteightball:
      events:
        - sw_eightball
      events_when_complete:
        action_light_ball8_on
        action_light_eightBall1500_on
      enable_events: ball_started
      disable_events: ball_ended
      reset_events: ball_ended

```

```

unlighteightball:
  events:
    - collect_special
  events_when_complete:
    action_light_ball8_off
    action_light_eightball1500_off
  enable_events: ball_started
  disable_events: ball_ended
  reset_events: ball_ended

openDiverter:
  events:
    - balldevice_eightballhole_ball_enter
  events_when_complete:
    open_diverter
  enable_events: ball_started
  disable_events: ball_ended
  reset_events: ball_ended

```

Some settings for Accrual Logic Blocks apply to all logic blocks, so you can [read about them there](#), including:

- The "name" of the counter (e.g. "tilt" or "super_jets" in the examples above)
- enable_events
- disable_events
- reset_events
- events_when_complete
- restart_on_complete
- disable_on_complete
- reset_each_ball

These other settings are specific to the Accrual logic blocks:

events:

This is where you configure the actual events that make up the "steps" of your Accrual Logic Block. "Accrual" is another work for "sum" or "total", and Accrual Logic Blocks fire their completion event after each (and every) step has been completed. The real power of Accrual Logic Blocks is that you can enter more than one events 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 theres 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).

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.)

The steps of an Accrual Logic Block can be completed in any order. (In fact, that's the whole point. If you want a Logic Block where the steps have to be completed in order, that's what the Sequence Logic Blocks are for.)

Read [our note about how to enter lists of lists](#) into the config files to make sure you get the configuration right.

player_variable:

This lets you specify the name of the player variable that will hold the progress for this logic block. If you don't specify a name, the player variable used will be called `<accrual_name>_status`.

Real world examples of Accrual Logic Blocks

Let's go through the Big Shot example from the beginning of this section to look at how Accrual Logic Blocks are implemented in the real world.

```

light_special:
  events:
    - sw_eightball,
    - drop_targets_Solids_lit_complete,
drop_targets_Stripes_lit_complete
  events_when_complete: lighting_special,
action_target_specialRight_light
  enable_events: ball_started, collect_special
  disable_events: ball_ended, lighting_special
  reset_events: ball_ended, lighting_special

```

As you can probably guess from the name, the `light_special` Logic Block is responsible for lighting the special target in Big Shot. The special is lit by (1) completing either bank of drop targets, and (2) completing either the center 8 Ball rollover lane or the getting the ball in the 8 ball kickout hole.

So this means there are two steps (one for the 8 Ball and one for the drop targets), and since those steps can complete in any order, we use the Accrual Logic Block instead of a Sequence or Counter type.

For the 8 ball step, we use the event `sw_eightball`. `sw_xxxx` events are automatically posted whenever a switch with the `xxxx` tag is hit, or whenever a ball enters a ball device with the

xxxx tag. So in our case, we added `eightball` to the list of tags for the 8 Ball rollover switch and the 8 Ball kickout hole ball device, meaning that either one of those being hit will cause MPF to post the `sw_eightball` event and the first step of our Accrual to be marked as complete.

For the drop target step, we need either bank (called "Solids" and "Stripes" in Big Shot) to complete, so we added the events that are automatically posted when a drop target bank is complete (`drop_targets_<name>_complete`). So when either the `drop_targets_solids_lit_complete` or `drop_targets_stripes_lit_complete` events is posted, then our second step is complete.

Once those two steps are complete, our Accrual will post two events: `lighting_special` and `action_target_specialright_light`. The `lighting_special` event is what we use to actually enable the Accrual Logic Block that watches for the special (more on that in a bit), and the `action_target_specialright_light` is the action event which lights the target called `specialright`.

For events which enable this Accrual, we have two: `ball_started` and `collect_special`. The first is pretty self-explanatory: We want this Accrual to start looking for hits when the ball starts. The second, `collect_special`, is the event that is posted after the special is actually collected (again more on that in a bit). We include this because in Big Shot, after the player collects a special the drop targets are reset and they have the opportunity to hit the 8 Ball and complete another full bank of drop targets to repeat this process.

For events which disable this Accrual, we also have two: `ball_ended` and `lighting_special`. Again `ball_ended` is pretty straightforward: we stop tracking progress when the ball ends. `lighting_special` is the event that this Accrual Logic Block itself posts when completed, so this causes it to disable itself once it's complete.

For reset events, we also reset this Accrual when the ball ends (via `ball_ended`), and we reset it via `lighting_special` as well. Again, `lighting_special` is the event that this Accrual posts when it completes, so if we didn't reset it when it was complete then it would be weird because it would automatically be complete when it was enabled again after the player collected the special.

Now lets look at how we configured our `collect_special` Accrual which picks up where this `lighting_special` leaves off:

```
collect_special:
  events:
    - target_specialLeft_lit_hit, target_specialRight_lit_hit
  events_when_complete:
    collect_special
  enable_events: lighting_special
  disable_events: ball_ended, collect_special
  reset_events: ball_ended, collect_special
```

First, notice that *collect_special* is enabled when the event *lighting_special* is posted. *lighting_special* is the event (which we made up) that our previous *light_special* Accrual posts, so in other words when *light_special* is complete, *collect_special* becomes active.

For the events that make of the steps of *collect_special*, we actually only have one step (but with two events, meaning either one of those events completes that step, and since there's only one step then either one of those two events actually completes this entire Accrual.

Our events for completion are *target_specialleft_lit_hit* and *target_specialright_lit_hit*. Those are built-in events which are posted automatically when a target device is hit while lit. (The target device being either *specialleft* or *specialright* in this case.) You might think, "Wait, so anytime one of those targets is hit while lit then this Accrual will complete?" The answer is yes! But consider what it takes to make that happen. The special targets aren't lit ordinarily. The *specialright* is lit based on the completion of the *light_special* Accrual we talked about before, and the *collect_special* Accrual that we're talking about here isn't activated until the *lighting_special* event is posted from *light_special*.

You'll notice that we have two options for the event step here (*target_specialleft_lit_hit* and *target_specialright_lit_hit*), but the *light_special* accrual only lights one of them. So what gives?

In our Big Shot config file, we also have a target group configured that groups together the two specials, and we have a target rotator set up which is tied to our slingshot switches. So all our previous Accrual has to do is just light one of the special targets, and then the rotator will cycle between the two as slingshots are hit. The targets themselves will keep track of their own status (lit or unlit) and post the proper events when they're hit (for example, *target_specialleft_lit_hit* or *target_specialleft_unlit_hit*), so that's how this Accrual only completes when a the lit special is hit.

The rest of the settings for this *collect_special* accrual are pretty straightforward. We disable and reset it when the ball ends, and we also reset and disable it when *collect_special* is posted (which is the event that it posts when complete) so it's ready when it's lit again.

If you're wondering what actually happens when the *collect_special* event is posted, that's something we handle in a scriptlet in Big Shot. In this case we have a scriptlet which automatically loads whenever a game is started which registers a handler for the *collect_special* event and then fires the knocker coil and gives the player a credit.

So hopefully that shows the power and simplicity of the Accrual Logic Blocks. We know that at this point you might be thinking, "What? You're calling that simple???" But think about it: what if you had to write all the Python code to do all this logic manually? It would take hours and hours, and you'd be debugging it for weeks. But thanks to these two Logic Blocks, you can put all this functionality into your game in just a few minutes. (In our case we literally drew flow charts to map out the process which we used to create these Logic Blocks.)

At this point you should be able to look at the other three Accrual Logic Blocks from Big Shot (*lightightball*, *unlightightball*, and *opendiverter*) and understand what they're doing and why we used them.

animations

You use the `animations:` section of your configuration file to specify additional non-default settings for any animations you want to use in your game.



This section can be used in your machine-wide config files.



This section can be

used in your mode-specific config files.

Note: You do *not* have to have an entry for every single animation 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. (See the [assetdefaults:](#) section for details. Also be sure to read the section on [Managing Assets](#) for an overview of how assets work.)

Here's a example:

```
animations:
  rolling_ball:
    alpha_color: 15
```

Since Animations are just a type of Asset in MPF, there are a few config settings that are generic which apply to all types of assets, so read the [reference documentation on Assets](#) for details about how the `name`, `file:`, and `load:` settings work.

Then on top of the defaults, animations have one specific settings:

alpha_color:

This is a color that will be rendered as transparent, allowing any display elements at a lower layer to show through. Right now this is a DMD shade value 0-15. We'll soon add support for per-pixel alpha where you can build differing degrees of alpha blending into each pixel.

asset_defaults

The `asset_defaults:` section of your machine 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.



This section can be used in your machine-wide config files.



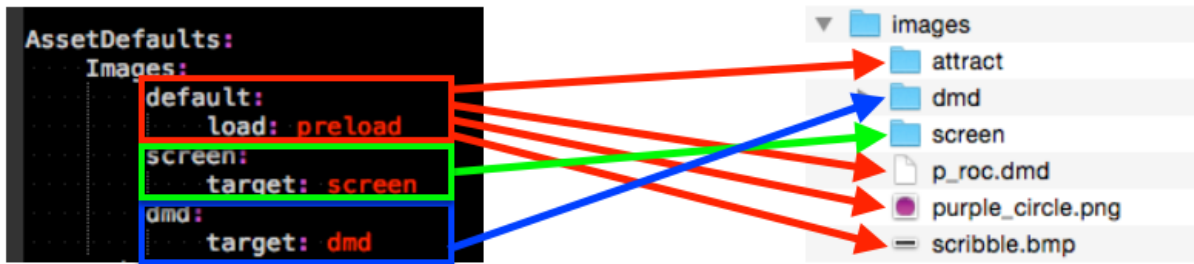
This section *cannot* be used in mode-specific config files.

An `asset_defaults:` section of your config file is required for each type of asset you use, so therefore there is an `asset_defaults:` section in the system-wide `mpfconfig.yaml` file. Let's take a look at it to understand how the `asset_defaults:` section works:

```
asset_defaults:
  images:
    default:
      load: preload
    screen:
      target: screen
    dmd:
      target: dmd
  animations:
    default:
      load: preload
      target: dmd
    screen:
      load: preload
      target: screen
  sounds:
    default:
      track: sfx
      load: preload
    voice:
      track: voice
      load: preload
    sfx:
      track: sfx
      load: preload
    music:
      track: music
      load: preload
  shows:
    default:
      load: preload
  movies:
    default:
      load: preload
```

First, notice that `asset_defaults:` has subsections for each different type of asset that MPF uses: Images, Animations, Sounds, Shows, and Movies.

Then under each specific asset type, there are a few more subsections. For example, the `Images:` section above contains the subsections `default:`, `screen:`, and `dmd:`. The `default:` section represents global default settings for this type of asset. But any additional subsections (for example `screen:` and `dmd:` in the `Images:` section) hold settings that will apply to a *subfolder with the name of the section*. The following diagram should make this clear:



A few notes on this:

- Any images in the "screen" folder will have the default settings of `target: screen` and `load: preload`, since the default settings would be applied first, then the folder-specific settings.
- Any images in the "attract" folder use the default settings since there is no folder-specific entry in the `asset_defaults:` section for `attract`.
- Any images in the root folder (`p_roc.dmd`, `purple_circle.dmd`, etc.) will also receive the `default:` settings since they are not in one of the folders with a name in the config file.

So... what types of settings can you specify in the `asset_defaults:` section? Any setting that the asset itself supports. (Just look up [Images](#), [Animations](#), [Sounds](#), Shows, or Videos in the config file reference to see a list of settings you can apply to an asset.)

assets

Even though you're reading a section called "Assets" in the configuration file reference, `assets:` is actually not a valid entry. Instead this documentation explains the generic common settings across individual types of assets, including:

- animations
- images
- movies
- shows
- sounds

Basically all of those things are assets, and so they all have certain settings in common, so we're explaining the common settings here so we don't have to waste space by copying-and-pasting this stuff into five different places.



This section can be used in your machine-wide config files.



This section can be

used in your mode-specific config files.

Even though these settings apply to any one of the five asset types, we'll use an image as an example:

```
images:
  insert_coin:
    load: preload
  hello_face:
    file: hello_face_300.jpg
    load: None
```

First, even though we've mentioned this before, remember that *you do not have to create an entry in your config file for every asset you want to use!*

<name>:

Each sub-entry in your asset 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. So in the images 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.)

file:

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.

load:

Specifies when this asset should be loaded. (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.)
- `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 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 get unloaded on demand too.)

Note that you can configure `load:` options in the `assetdefaults:` section of your machine 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 specify an `assetdefaults:` entry for that folder rather than specifying entries in your config for specific assets. The choice is up to you.)

auditor

The `auditor:` section of the machine configuration file lets you control what events the [auditor module](#) audits.



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 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

```

save_events

A list of events trigger the auditor to update the audit file on disk. We include *ball_ended* so we write out the audits after each ball, and also *game_ended* so we capture the final scores of each player.

audit

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

A list of which events you want to audit. These are the names of any events you want.

player

This is a list of player variables you want to track. 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.

num_player_top_records

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.

autofire_coils

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. (See the [Autofire coils device documentation](#) for the full explanation.)



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:

```
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.)

Also 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. Sub-options include:

coil:

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:

The name of the switch that fires that coil.

reverse_switch:

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*.

pulse_ms:

The number of milliseconds you want this coil to pulse for. If you don't specify a value here then it will use the default setting that you specified for that coil in the Coils section of the configuration file.

pulse_power:

This is the power of the initial pulse on a 0-8 scale. (0 is 0%, 4 is 50%, 8 is 100%, etc.)

hold_power:

This is the power of the hold on a 0-8 scale. (0 is 0%, 4 is 50%, 8 is 100%, etc.)

pwn_on_ms:

The number of millisecond for the "on" portion of a pwm-based hold action. Not all platforms support this, so see your platform-specific How To guide for details.

pwm_off_ms:

The number of millisecond for the "off" portion of a pwm-based hold action. Not all platforms support this, so see your platform-specific How To guide for details.

pulse_power32:

A 32-bit pwm mask for the initial pulse of the coil. Not all platforms support this, so see your platform-specific How To guide for details.

hold_power32:

A 32-bit pwm mask for the hold power of the coil. Not all platforms support this, so see your platform-specific How To guide for details.

pulse_pwm_mask:

An 8-bit pwm mask for the initial pulse of the coil. Not all platforms support this, so see your platform-specific How To guide for details.

hold_pwm_mask:

An 8-bit pwm mask for the hold power of the coil. Not all platforms support this, so see your platform-specific How To guide for details.

delay:

The number of milliseconds the coil will delay before firing or starting the coil action after the switch is activated. Note that not all platforms support this. (The P-ROC does not. The FAST Controller does.)

recycle_ms:

The minimum number of milliseconds you want between subsequent firings of this same rule. Put another way, if this autofire coil rule fires, then it will ignore additional switch actions until this `recycle_ms`: time is up. Note that not all hardware platforms support this in the same way. The P-ROC only has a delay option for 125ms. That is, there is either no delay, or a 125ms delay. Nothing in the middle. The FAST Controller lets you set the delay to whatever you want.

debounced:

True or False. Whether this rule should fire based on the switch being debounced or not. In most cases you should use False since you want these hardware rules to fire as fast as possible, but you can use True if you get a lot of random phantom firings.

drive_now:

True or False. If True, the hardware controller will check the current state of the switches and update the state of the coil when the rule is applied. If False, the rule will only apply the next time the switches change state.

A good example of this is for your flipper switches. When there's no game in progress, the autofire coil rules for the flipper are disabled, since you don't want the flippers to operate. Now imagine if a player holds in a flipper button and then starts a game. If they keep holding the button, you would want the flipper to activate as soon as they were enabled, which you would do by settings this to True. If this was False then the hardware controller would not apply the rule based on the current state of the switches when the rule was activated, meaning the player would have to release and then push the flipper button again for the first flip, which would be weird.

(By the way, the flipper example above was just to illustrate the point of how the *drive_now*: setting works. In reality you don't have to manually configure all your autofire coil rules for flippers—instead you just set up the flippers in the [flippers: section of the config files](#) and all the rules are set up for you automatically.

Device Control Events

Device control events are events you can use to control devices. They are configured in your machine-wide or mode config with settings that end in *_events*. For example, if a device has a setting for *enable_events*: and you add an event to that setting, then when that event is posted, the device will enable. You can add single events or lists of events to these settings, and you can also configure time-delays for how much time passes between the event being posted and the action to take place. Details are available in the [device control event documentation](#).

Autofire coils make use of the following device control events:

enable_events:

Enables this autofire coil by writing the hardware rule to the pinball controller hardware. Default is *ball_started*.

disable_events:

Disables this autofire coil by clearing the hardware rule from the pinball controller hardware.. Default is *ball_ending*.

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

ball_devices

The *ball_devices:* config file section contains settings for the various [Ball Devices](#) in a pinball machine.



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:

```
ball_devices:
    trough:
        tags: trough, home, drain
        label: Main Trough
        ball_switches: s_trough1, s_trough2, s_trough3, s_trough4,
s_trough5, s_trough6, s_troughJam
        entrance_count_delay: 0.5s
        eject_coil: c_trough
        confirm_eject_type: target
        eject_targets: right_shooter_lane
        jam_switch: s_trough_jam

    right_shooter_lane:
        tags: ball_add_live
        label: Right Plunger Lane
        ball_switches: s_shooter_right
        entrance_count_delay: 0.3s
        eject_coil: c_shooter_right
        confirm_eject_type: target

    left_shooter_lane:
        label: Left Plunger Lane
        ball_switches: s_shooter_left
        entrance_count_delay: 0.3s
        eject_coil: c_shooter_left
        confirm_eject_type: count
        exit_count_delay: 0.5s
```

ball_capacity:

Optional value for how many balls this device can hold. You only need to specify this if your device holds more balls than 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:

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 specific 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:

A list of timeouts (in [MPF time string format](#)) 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 use *20 seconds* as the default.

ball_switches:

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:

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:

This is the name of the event that will be used to confirm a successful ball eject if you have *confirm_eject_type: event*.

Default is *None*.

confirm_eject_switch:

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*.

Default is *None*.

confirm_eject_type:

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:

- *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.

- *target* - 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.
- *count* - The device will confirm the eject when it notices that a ball is "missing". You can set a value for *exit_count_delay* if you want to wait longer than your typical switch count delay to make sure the ball really made it out.
- *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.

Default is *count*.

eject_coil:

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_jam_pulse:

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 ejections 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_retry_pulse:

The new pulse time, in ms, that the eject coil will use if the eject has failed 3 times. This pulse time is used on the 4th pulse up until the device stops trying.

Default is *None* which means the ball device will not change the pulse time after 3 attempts.

eject_targets:

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 *left cannon*, 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.)

The default is *playfield*.

eject_timeouts:

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.

entrance_count_delay:

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_switch:

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*.

exit_count_delay:

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.

You can enter values here in seconds or milliseconds. See the full explanation of the time duration formats [here](#).

Default is *500ms*.

hold_coil_release_time:

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_coil:

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*.

hold_switches:

A switch (or list of switches) that indicates a ball is in position to be captured by a *hold_coil*.

Default is *None*.

jam_switch:

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.

Default is *None*.

max_eject_attempts:

Defines how many times this ball device will attempt to eject a ball before deciding that the eject permanently failed. A value of zero

Default is *0* which means there's no limit. (e.g. the device will just keep trying to eject the ball forever.)

mechanical_eject:

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*.

Device Control Events

Device control events are events you can use to control devices. They are configured in your machine-wide or mode config with settings that end in *_events*. For example, if a device has a setting for *enable_events*: and you add an event to that setting, then when that event is posted, the device will enable. You can add single events or lists of events to these settings, and you can also configure time-delays for how much time passes between the event being posted and the action to take place. Details are available in the [device control event documentation](#).

Ball devices make use of the following device control events:

eject_all_events:

Causes this ball device to eject all its balls. Default is *None*.

eject_events:

Causes this ball device to eject one ball. Default is *None*.

hold_events:

Causes this ball device to enable its hold coil. Default is *None*.

request_ball_events:

Causes this ball device to request a ball. Default is *None*.

stop_events:

Causes this ball device to stop all activity, including canceling any ejects or eject confirmations that are in progress. Does not cause this device to eject any balls. Default is *None*.

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

MPF also uses tags for ball devices for special purposes, including:

- **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.
- **ball_add_live** - Used to tag the device you want to use to launch new balls into play. Typically this is the plunger device.
- **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.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

ball_locks

The *ball_locks*: section of your config files let you setup logical ball locks which can be used to physically or virtually lock balls.



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:

```
ball_locks:
  bunker:
    balls_to_lock: 3
    lock_devices: bd_bunker
```

<name>:

The logical name of this ball lock device.

balls_to_lock: (required)

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:

A list of one (or more) ball devices that will collect balls which will count towards this lock.

source_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*.

request_new_balls_to_pf:

Boolean which controls whether this logical ball lock will automatically add another ball into play after it locks a ball. Default is *True*.

Device Control Events

Device control events are events you can use to control devices. They are configured in your machine-wide or mode config with settings that end in *_events*. For example, if a device has a setting for *enable_events*: and you add an event to that setting, then when that event is posted, the device will enable. You can add single events or lists of events to these settings, and you can also configure time-delays for how much time passes between the event being posted and the action to take place. Details are available in the [device control event documentation](#).

Ball locks make use of the following device control events:

enable_events:

Enables this device. Default is *ball_started*.

disable_events:

Disables this device. Default is *ball_ending*.

reset_events:

Resets this device by resetting the locked ball count to 0 and releasing all balls. Default is *machine_reset_phase_3, ball_starting, ball_ending*.

release_one_events:

Releases a single ball. Default is *None*.

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

ball_saves

The *ball_saves:* section of the config file lets you create [ball save devices](#).



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:

```
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: yes
    balls_to_save: 1
    debug: yes
```

<name>:

The name of this ball save device. (This is "default" in the config snippet above.) The name is used in the events that are posted from this ball save. (The complete list of events posted by ball saves is included in the [ball save device documentation](#).)

active_time:

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.

hurry_up_time:

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.

grace_period:

The “secret” time (in [MPF time string format](#)) the ball save is still active. This is added onto the *active_time*. Default is 0.

timer_start_events:

An optional event (or list of events) in [MPF control event format](#) which starts this ball saver's countdown timer. Default is *None*.

auto_launch:

True/False which controls whether the ball save should auto launch the saved ball or wait for the player to launch it. Default is *True*.

balls_to_save:

How many balls this ball saver should save before disabling itself. Set it to -1 for unlimited. Default is 1.

Device Control Events

Device control events are events you can use to control devices. They are configured in your machine-wide or mode config with settings that end in *_events*. For example, if a device has a setting for *enable_events*: and you add an event to that setting, then when that event is posted, the device will enable. You can add single events or lists of events to these settings, and you can also configure time-delays for how much time passes between the event being posted and the action to take place. Details are available in the [device control event documentation](#).

Ball saves make use of the following device control events:

enable_events:

Enables the ball save. If you do not have a *timer_start_events* entry then enabling by an event here will also start the timer. Default is *None*.

disable_events:

Disables the ball save. Default is *ball_ending*.

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

bcp

The `bcp:` section of your machine config file controls how the MPF core engine communicates with the standalone media controller.

? This section can be used in your machine-wide config files. ? This section *cannot* be used in mode-specific config files.

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
        connection_attempts: 5
        require_connection: no

event_map:
    ball_started:
        command: ball_start
        params:
            player: "%player"
            ball: "%ball"

    ball_ending:
        command: ball_end

    machineflow_Attract_start:
        command: attract_start

    machineflow_Attract_stop:
        command: attract_stop

    game_ended:
        command: game_end

    player_turn_start:
        command: player_turn_start
        params:
            player: "%number"

player_variables:
#   __all__
    ball
    extra_balls

```

connections:

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.

- `<connection name>`: The name you want to use to refer to this connection. This is called `local_display` in the example above.
- `host`: The host name of the media controller. The default is `localhost`.
- `port`: The TCP port the remote media controller is listening on. The default is `5050`.
- `connection_attempts`: How many times the MPF core engine will attempt to connect to the media controller before giving up. A value of `-1` means it will keep trying forever.
- `require_connection`: If this is true (or "yes"), then the MPF core engine won't run if it can't connect to the remote media controller. Also if it's true, MPF will quit if it loses the connection. The default is `False`.

event_map:

This section contains a list of MPF events that get mapped to BCP commands. You shouldn't have to change this. This is what MPF uses internally to map MPF events to the [BCP command specification](#).

- `<event_name>`: The name of the MPF event you're creating a mapping for.
- `command`: The name of the BCP command that will be sent when the MPF event is posted.
- `params`: A list of parameters that will be passed via BCP along with this BCP command.

player_variables:

The `player_variables:` section lets you specify which MPF player variables will be broadcast via BCP to the media controller. (MPF will send these any time there's a change.) You can either list out the individual names of the players variables you want to send, like this:

```
player_variables:
    ball
    extra_balls
```

Or you can use the entry `__all__` (that's two underscores, the letters "all", then two more underscores) to send every change of every player variable to the media controller. Here's an example:

```
player_variables:
    __all__
```

coils

The `coils:` section of the config files is used to map coil ([solenoid](#)) names to driver board outputs. We 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
    pulse_ms: 30
    tags:
flipper_right_hold:
    number: A0-B0-1
    tags:
knocker:
    number: A0-B1-0
    pulse_ms: 20
    tags:
pop_bumper_left:
    number: A0-B1-1
    pulse_ms: 18
    tags: ball_search
ball_gate:
    number: A0-B1-2
    hold_power: 3
    tags: ball_search

```

The options are as follows:

<name>:

Each subsection of *coils*: is the name of the coil as you'd like to refer to it in your game code. This can really be anything you want, but it's obviously best to pick something that makes sense.

number: (required)

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 whether you're using a P-ROC or FAST controller, and what type of driver board you're using. See the platform-specific how to guides for details:

- [FAST Pinball Controller driver numbering](#)
- [P-ROC driver numbering](#)
- [P3-ROC driver numbering](#)
- WPC driver numbering (keep reading below)

WPC driver board number format

If you're using a WPC driver board (like if you dropped your P-ROC or FAST controller into an existing pinball machine), you can enter the coil number from the operators manual and the P-ROC platform driver code will convert it to the proper driver number. In this case you'd preface them with the letter 'C'. Here's an example from a Williams Judge Dredd machine:

```

trip_drop_target:
  number: C10
  tags:
diverter:
  number: C11
  tags:
trough:
  number: C13
sling_left:
  number: C15
  tags:
sling_right:
  number: C16
  tags:

```

In the case of WPC driver boards, you can also specify coil numbers for flipper coils connected to a Fliptronics board. For example (again from Judge Dredd):

```

flipper_lower_right_main:
  number: FLRM
flipper_lower_right_hold:
  number: FLRH
flipper_lower_left_main:
  number: FLLM
flipper_lower_left_hold:
  number: FLLH
flipper_upper_right_main:
  number: FURM
flipper_upper_right_hold:
  number: FURH
flipper_upper_left_main:
  number: FULM
flipper_upper_left_hold:
  number: FULH

```

pulse_ms:

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.

hold_power:

This setting lets you control how much power is sent to the coil when it's "held" in the on position. This is an integer value from 0-8 which controls the relative power:

- 0: 0% power (e.g. "off")
- 1: 12.5%
- 2: 25%

- 3: 37.5%
- 4: 50%
- 5: 62.5%
- 6: 75%
- 7: 87.5%
- 8: 100% (see the "allow_enable" section below)

Different hardware platforms implement the hold power in different ways, so this 0-8 *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 `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.

allow_enable:

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 `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 *hold_power* setting.

The 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 *hold_power* entry.

If you try to enable a coil that does not have *hold_power* configured or *allow_enabled* set to true, then the coil will not actually be enabled and you'll get a warning in your log file.

Device Control Events

Device control events are events you can use to control devices. They are configured in your machine-wide or mode config with settings that end in *_events*. For example, if a device has a setting for *enable_events:* and you add an event to that setting, then when that event is posted, the device will enable. You can add single events or lists of events to these settings, and you can also configure time-delays for how much time passes between the event being posted and the action to take place. Details are available in the [device control event documentation](#).

Coils make use of the following device control events:

enable_events:

Enabled (hold on) this coil. This requires that *allow_enable* is true or that a *hold_power* setting is configured. Default is *None*.

disable_events:

Disables this coil (meaning that if it's active, it's shut off). Default is *None*.

pulse_events:

Pulses this coil for the default *pulse_ms* milliseconds. Default is *None*.

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

config

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.



This section can be used in your machine-wide config files.



This section can be

used in your mode-specific config files.

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
```

The framework will attempt to convert relative and absolute paths based on your OS, and it can deal with slashes in either direction.

The framework will then open those files one-by-one and merge in 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 files, meaning that one config file can call another which will call another, etc. Any time the framework 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. If that fails then it will try the last known good directory that worked for a config file.

counters

A *counter* is a type of [logic block](#) that tracks the number of times an event is posted towards the progress of a completion goal. You can use optional parameters to specify whether multiple occurrences in a very short time-frame should be grouped together and counted as one hit, the counting interval, and whether this counter counts up or down.



This section can be used in your machine-wide config files.



This section can be used in your mode-specific config files.

Counter logic blocks can be used for things like counting total combos made, counting down progression towards super jets, etc.

Here's an example of a counter you could use to track progress towards super jets:

```
logic_blocks:
  counters:
    super_jets:
      count_events: sw_pop
      event_when_hit: pop_hit
      starting_count: 75
      count_complete_value: 0
      direction: down
      events_when_complete: super_jets_start
```

Some settings for Logic Blocks apply to all logic blocks, so you can [read about them there](#), including:

- The "name" of the counter (e.g. "tilt" or "super_jets" in the examples above)
- enable_events
- disable_events
- reset_events
- events_when_complete
- restart_on_complete
- disable_on_complete
- reset_each_ball

"Counter"-specific settings

In addition to the general Logic Block settings, the Counter type of logic block has the following specific settings:

count_events:

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.)

event_when_hit:

This is the name of the event that will be automatically posted when when this Counter tracks a hit. The name of this event can be whatever you want (since this Counter is the one that's posting it). Note this event will be posted with a parameter "count" which will be set to the integer value of counter, so make sure that any handlers that have registered for this event are expecting that parameter.

count_complete_value:

When the Counter exceeds (or gets below if you're counting down) this value, it will post its "complete" event and be considered complete.

multiple_hit_window:

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 the *multiple_hit_window* is set to *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).

player_variable:

This lets you specify the name of the player variable that will hold the count progress for this counter. If you don't specify a name, the player variable used will be called *<counter_name>_count*.

count_interval:

Let's you specify what the 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.

direction:

This is either *up* or *down* and specifies whether this counter counts up or counts down.

starting_count:

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 a direction of *down*, you'll want to also set this to something more than zero.

credits

The *credits:* section of the config file contains settings for the credits mode.



This section can be used in your machine-wide config files.



This section can be used in your mode-specific config files.

If you include this section in both your machine-wide config and the config for the credits mode, the machine-wide config is loaded first, and then any new or different settings in the credits mode config are merged in.

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: no
  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
  fractional_credit_expiration_time: 15m
  credit_expiration_time: 2h
  persist_credits_while_off_time: 1h
  free_play_string: FREE PLAY
  credits_string: CREDITS
```

switches:

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.

switch:

The name of the switch (from your machine-wide *switches:* section) for the credit switch.

value:

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.

type:

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.*

pricing_tiers:

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.

credit_expiration_time:

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:

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*.

free_play:

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:

The text string that will be used in the `credits_string` machine variable when the machine is in free play. Default is *FREE PLAY*.

fractional_credit_expiration_time:

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*.

max_credits

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.

persist_credits_while_off_time:

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_credit_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.

dmd

The `dmd:` section of your machine configuration file is how you configure your DMD (dot matrix display). Note that you need to do this for both physical DMDs *and* if you just want to have a virtual window-based DMD that looks like an old school physical DMD.



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 of a traditional single-color DMD:

```
dmd:
  physical: yes
  width: 128
  height: 32
  shades: 16
  fps: auto
```

And here's an example of a full-color DMD:

```
dmd:
  physical: yes
  width: 128
  height: 32
  fps: auto
  type: color
```

Specific settings include:

physical:

This is a True / False (or Yes / No) entry which is used to specify whether you have a physical DMD attached. A "physical DMD" in this case is a hardware DMD connected to a P-ROC or FAST Pinball Controller via the 14-pin ribbon cable, or a "SmartMatrix" RGB LED-based DMD.

If you configure this for *no*, then your physical DMD won't work, but you can still configure a virtual dot-like DMD display element for your on screen LCD window (which can be single- or full-color). The default is "No."

width:

The width of your DMD in pixels. If you have a traditional single-color physical pinball hardware DMD connected via the 14-pin DMD cable, this will be 128. If you don't have a physical DMD (and you're just configuring the DMD here so you can have an on screen virtual DMD), then you can set this to whatever you want. (You can even go higher than 128 if you want high res while keeping the look of dots.) The default is 128.

height:

The height of your DMD in pixels. Like `width:` above, if you have a physical pinball DMD attached then this will be 32. If you're setting up your DMD for use as a virtual on screen DMD, then this can be whatever you want. The default is 32.

shades:

This is the total number of shades (intensity levels) your mono-color DMD can support. Most of your favorite 90s Williams/Bally games used 4 shades (loosely called "bright orange," "orange", "dark orange", and "off"), though today's pinball controllers can drive 16 shades (even on old pinball machines with old high-voltage gas DMDs).

If you use a value other than 16 here then you'll have to make sure that any images and animations you import are also set to only use your smaller number of shades. You can also go higher than 16 if you don't have a physical DMD (since the on screen virtual DMD can use as many shades as you want).

If you have a color DMD, this setting is ignored.

The default value is 16.

fps:

How many frames per second you want you DMD to operate at. The default value is `auto` which means it will operate at the same speed as your [Machine HZ](#). Note that you can't

set this higher than your machine Hz. (Well, you can, but it would just update at the machine Hz rate.)

Also some display hardware has speed limitations, so if you're running your machine Hz at a really high rate (say, higher than 60) then you should hard code your DMD fps to something more sane here.

The default is "auto."

type:

This is let's you specify whether you have a mono-color or full-color DMD. The only valid option here is `type: color` (for full-color DMDs). Any other value, or a not including this setting in your DMD config, means that the DMD will be setup as a mono-color DMD.

This setting affects this DMD regardless of whether it's physical or virtual. ("Virtual" meaning it's show in a window on your LCD via the Virtual DMD display element.)

diverters

You create and configure your diverters in the *diverters:* section of your machine configuration file. (Read more about diverters in MPF [here](#).)



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 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_underTopHole, s_underLeftHole,
s_underBorgHole
    targets_when_active: bd_rightCannonVUK
    targets_when_inactive: bd_leftVUK
    feeder_devices: bd_catapult
  subway_bottom_diverter:
    activation_coil: c_underDivertorBottom
```

```

    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

```

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 it's 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 kind 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? :)

<diverter name>

Create a subentry in your *diverters:* section for each diverter you want to create. (Remember that you should create anything that's activated to change the path of the ball as a diverter, including traditional diverters, up/down posts, coil-controlled gates, playfield trap doors, and controlled drop targets which block entrances to devices.)

activation_coil:

The name of the coil that is used to activate your diverter.

deactivation_coil:

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.

type:

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 *hold_power*, then it will use that pwm pattern to hold the coil on. If no *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* to that coil's configuration in the *coils:* section of your machine configuration file.)

activation_time:

This is how long the diverter stays active once it's been activated, entered in the [MPF time string format](#). 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.

activation_switches:

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.

deactivation_switches:

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_switches:

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.

targets_when_active:

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:

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.

feeder_devices:

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.

Device Control Events

Device control events are events you can use to control devices. They are configured in your machine-wide or mode config with settings that end in *_events*. For example, if a device has a setting for *enable_events*: and you add an event to that setting, then when that event is posted, the device will enable. You can add single events or lists of events to these settings, and you can also configure time-delays for how much time passes between the event being posted and the action to take place. Details are available in the [device control event documentation](#).

Diverter make use of the following device control events:

activate_events:

Causes this diverter to activate. Default is *None*.

deactivate_events:

Causes this diverter to deactivate. Default is *None*.

enable_events:

Enables this diverter. (Remember that enabling a diverter is not the same as activating it.)

Default is *None*.

disable_events:

Disables 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.

Default is *None*.

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

driver_enabled

The *driver_enabled:* section of the config files is used to create [driver-enabled devices](#).



This section can be used in your machine-wide config files.



Some of the settings in this section can be used in mode-specific config files.

Here's a sample config from Pin*Bot:

```
driver_enabled:
  playfield_devices:
    number: c23
```

The options are as follows:

<name>:

Each subsection of *driver_enabled:* is the name of the device-enabled device as you'd like to refer to it in your game code. This can really be anything you want, but it's obviously best to pick something that makes sense.

number (and all other settings):

Driver-enabled devices in MPF are based on the regular driver (coil) devices, so all other settings for coils also apply here. So number, tags, etc. are all the same as coils, so you can refer to the [configuration file reference settings for coils](#): to see how to configure them.

drop_targets

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](#).)



This section can be used in machine-wide config files.



This section cannot be used in your mode-specific config files. 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.

What about drop targets with lights?

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.

<name>:

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.

switch:

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.

reset_coil:

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.

knockdown_coil:

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.

Device Control Events

Device control events are events you can use to control devices. They are configured in your machine-wide or mode config with settings that end in *_events*. For example, if a device has a setting for *enable_events:* and you add an event to that setting, then when that event is posted, the device will enable. You can add single events or lists of events to these settings, and you can also configure time-delays for how much time passes between the event being posted and the action to take place. Details are available in the [device control event documentation](#).

Drop targets make use of the following device control events:

reset_events:

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*.

knockdown_events:

Pulses this drop target's knockdown coil. (If this drop target doesn't have a knockdown coil, then these events will have no effect.)

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

drop_target_banks

Once you've configured your individual drop targets, you group them together into banks via the *drop_target_banks*: section of your config file.



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 from *Judge Dredd*:

```
drop_target_banks:
  judge:
    drop_targets: j, u, d, g, e
    reset_coils: c_reset_drop_targets
    reset_events:
      drop_targets_judge_complete: 1s
```

What about drop target banks with lights?

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.

<name>:

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.)

drop_targets:

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.

reset_coil:

The name of the coil that is fired to reset this bank of drop targets.

reset_coils:

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.

Device Control Events

Device control events are events you can use to control devices. They are configured in your machine-wide or mode config with settings that end in *_events*. For example, if a device has a setting for *enable_events*: and you add an event to that setting, then when that event is posted, the device will enable. You can add single events or lists of events to these settings, and you can also configure time-delays for how much time passes between the event being posted and the action to take place. Details are available in the [device control event documentation](#).

Drop target banks make use of the following device control events:

reset_events:

Resets this drop target bank by pulsing this bank's *reset_coil* or *reset_coils*. Default is *machine_reset_phase_3, ball_starting*.

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

event_player

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 section can be used in your machine-wide config files.



This section can be used in mode-specific config files.

Here's an example:

```
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:
    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:

```
event_player:
  mode_shoot_here_started:
    cmd_upper_target_reset
```

fast

The `fast:` section of the machine configuration files is where you configure hardware options that are specific to the FAST Pinball Controller.



This section can be used in your machine-wide config files.



This section *cannot* be used in mode-specific config files.

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
  config_number_format: hex
  baud: 921600
  watchdog: 1s
  default_debounce_close: 10ms
  default_debounce_open: 30ms
```

ports: (required)

This is the list of the virtual COM ports that the FAST controller uses. On Windows machines it's typically something like `com5` or `com6`. On Linux and Macs, it's probably something like `/dev/xxxx`.

If you don't know the port names that your FAST Controller is using, the easiest way is just to plug it in (make sure it's on) and see what port appears. You can confirm this by unplugging it or powering it off and making sure those are the ports that disappear too.

The FAST Core and FAST WPC controllers use three ports—one each for the DMD processor, the NET (main) processor, and one for the LED processor. You want to enter the first three ports into your config.

The FAST Nano controller does not have a DMD processor, so you need to add the middle two ports that do show up.

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.

config_number_format:

This specifies whether you want to specify the addresses of your lights, LEDs, coils, and switches by their integer values or as hex values. Valid options here are `hex` and `int`. Note if you configure your `driverboards: as wpc` or `wpc95` (in the `hardware:` section), then you also have the option of using the original WPC numbers from your operators manual.

baud:

The baudrate for the FAST COM ports. The default is 921600. This setting is included in the `mpfconfig.yaml` default configuration file, so you don't have to add it to your config file unless you're setting a value other than the default.

watchdog:

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.

The purpose of this is to prevent a coil from burning out if MPF crashes or loses communication with the hardware. In those cases the watchdog timer runs to zero and the FAST controller kills the power. 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.

default_debounce_close:

This is the default debounce time that the FAST controller will use for switches when they transition from the open to the closed state. You can add this section to your config to set it to whatever you want, or you can leave it out to use the default which is currently 10ms. Basically if you hit switches or targets and they don't register, that means you need to set a shorter debounce time. And if you get random switch events when a switch isn't actively closed, you need to increase the debounce time.

You can also set the debounce time on a switch-by-switch basis.

default_debounce_open:

This is like `default_debounce_close`, except it controls the timing of the switch moving from the closed to open position. The default is 30ms.

fonts

The `fonts:` section of your machine configuration files is where you map out the names and settings for the fonts your machine uses, including mapping source fonts and setting antialiasing, size, and cropping options.



This section can be used in your machine-wide config files.



This section *cannot* be used in mode-specific config files.

Note that in addition to this reference, we have a [Step-by-Step tutorial guide which explains how to set up fonts](#) in MPF. We also have a [Font Tester tool](#) which comes in handy as you're figuring out the settings you'll enter into your configuration file here.

Example usage:

```

fonts:
  default:
    file: Quadrit.ttf
    size: 10
    crop_top: 2
    crop_bottom: 5
  space title huge:
    file: DEADJIM.TTF
    size: 29
    antialias: yes
    crop_top: 3
    crop_bottom: 3
  space title:
    file: DEADJIM.TTF
    size: 21
    antialias: yes
    crop_top: 2
    crop_bottom: 3
  medium:
    file: pixelmix.ttf
    size: 8
    crop_top: 1
    crop_bottom: 1
  small:
    file: smallest_pixel-7.ttf
    size: 9
    crop_top: 2
    crop_bottom: 3
  tall title:
    file: big_noodle_titling.ttf
    size: 20

```

You enter multiple groups of settings, each with its own name. Then whenever you want to use one of those fonts in your game, you can simply call it by name. For example, in the configuration file above, if you ask for something to be rendered to the display with the font

called "space title", it will use the DEADJIM.TTF font, point size 29, antialiased, and crop three rows of pixels off the top and 3 rows of off the bottom.

Let's look through each of these settings:

<MPF font name>:

The top-level entries in your `fonts:` section are for the font names that you want to use to refer to the fonts in your game. In the above example, we have `default`, `space title huge`, `space title medium`, etc. You can name these whatever you want. Note that each of them is only meant to be used for one size since, though you can use the same font files over and over to create `title_big`, `title_medium`, `title_small`, etc.

If you create an entry for called `default:` (like we have in the first entry above), then MPF will use those settings whenever text is rendered without a font being specified.

size:

The size this font will be rendered at.

file:

The name of the font file that will be used when the game programmer calls for a font in this size. MPF will first look in the fonts folder location in the machine's folder, and then it will look in the MPF system fonts folder.

antialias:

Whether this font should be antialiased.

crop_top:

The number of blank rows that will be removed from above the font after its rendered.

crop_bottom:

The number of black rows that will be removed from the bottom of the font surface after it's rendered.

Alpha blending

In the next few weeks we'll add settings for alpha blends, including controlling whether the background is opaque or transparent, and, when antialiasing is used, whether per-pixel alpha blends are used.

flashers

You can configure your [flashers](#) via the *flashers:* section of your configuration file.



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 from a Williams *Road Show* machine:

```
flashers:
  f_little_flipper:
    number: c37
    label: Flasher above middle left flipper
    tags: white
  f_left_ramp:
    number: c38
    label: Flasher above Bob's Bunker
    tags: yellow
  f_back_white:
    number: c39
    flash_ms: 40
    label: Two white rear wall flashers
    tags: white
  f_back_yellow:
    number: c40
    flash_ms: 40
    label: Two yellow rear wall flashers
    tags: yellow
  f_back_red:
    number: c41
    flash_ms: 40
    label: Two red rear wall flashers
    tags: red
  f_blasting_zone:
    number: c42
    label: Blasting Zone flasher
    tags: white
  f_right_ramp:
    number: c43
    label: Flasher in front of Red
    tags: white
  f_jets_at_max:
    number: c44
    label: Playfield insert in the pop bumpers
    tags: white
```


<FlasherName>:

Each subsection of *flashers:* is the name of the flasher as you'd like to refer to it in your game code. This can really be anything you want, but it's obviously best to pick something that makes sense.

number: (required)

This is the number for the flasher which specifies which driver output the flasher is physically connected to. The exact format used here will depend on whether you're using a P-ROC or FAST controller, and what type of driver board you're using. (Williams WPC, System 11, Stern, P-ROC PD-16, etc.)

Since flashers are connected to driver outputs (just like coils), the number scheme here is identical to coils. Refer to the [number: section of the coils: configuration documentation](#) for details on how to enter these numbers for different platforms. (The example file above is for WPC driver boards, which is why the flasher numbers are in the *Cxx* format.)

flash_ms:

The default time, in milliseconds, that this flasher will flash for when it's sent a "flash" command.

Device Control Events

Device control events are events you can use to control devices. They are configured in your machine-wide or mode config with settings that end in *_events*. For example, if a device has a setting for *enable_events:* and you add an event to that setting, then when that event is posted, the device will enable. You can add single events or lists of events to these settings, and you can also configure time-delays for how much time passes between the event being posted and the action to take place. Details are available in the [device control event documentation](#).

Flashers make use of the following device control events:

flash_events:

Causes this flasher to flash using it's default *flash_ms:* time. Default is *None*.

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

flippers

The *flippers*: section of the config files contains all the settings for the flippers in a pinball machine.



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 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's a lot of options. Read the [flippers section of the documentation](#) for all the details. You should definitely read that first before digging into the configuration options here.

```
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
  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

```

You configure flippers by putting a *flippers:* entry in your configuration file. Then you create a sub-entry for each flipper. (In the config file above, the flipper sub-entries are named *lower_left*, *lower_right*, *upper_left*, and *upper_right*.)

main_coil:

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.

hold_coil:

The name of the hold coil winding for dual-wound flipper coils.

activation_switch:

The switch that controls this flipper. (i.e. the flipper button)

eos_switch:

If you're using an end of stroke switch with this flipper, enter the switch name here.

use_eos:

True or False, controls whether you're using an EOS switch

Device Control Events

Device control events are events you can use to control devices. They are configured in your machine-wide or mode config with settings that end in *_events*. For example, if a device has a setting for *enable_events:* and you add an event to that setting, then when that event is posted, the device will enable. You can add single events or lists of events to these settings,

and you can also configure time-delays for how much time passes between the event being posted and the action to take place. Details are available in the [device control event documentation](#).

Flippers make use of the following device control events:

enable_events:

Enables this flipper. Default is *ball_started*.

disable_events:

Disabled this flipper. Default is *ball_ending*.

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

game

The `game :` section of the configuration files holds settings related to the game play.



This section can be used in your machine-wide config files.



This section *cannot* be used in mode-specific config files.

This list is very incomplete, but here's what's implemented now:

```
game:
  balls_per_game: 3
  max_players: 4
```

balls_per_game:

How many balls the game is. Typically it's 3 or 3 but it can be anything. The framework doesn't care.

max_players:

Typically this is 4, but the core framework doesn't care what this number is. (When we get to adding the display support, we'll probably have to deal with certain limits, but for now it doesn't matter.)

start_game_switch_tag:

add_player_switch_tag:

gis

The *gis*: section of the config file is for [GI \(general illumination\)](#) settings for hardware that uses them. (This is typically with only used when you're writing new code for an existing pinball machine. Custom games tend to use all individually-addressable lights.)



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 from *Judge Dredd*:

```
gis:
  gi01: # lower backglass
        number: G01
  gi02: # mid backglass and rear playfield
        number: G02
  gi03: # upper left backglass and slings, variable
        number: G03
  gi04: # upper right backglass and deadworld globe, variable
        number: G04
  gi05: # coin slot lights & side cabinet fire buttons
        number: G05
```

number:

The hardware number of the GI string. You can enter the number from the operators manual, starting with the letter "G".

Device Control Events

Device control events are events you can use to control devices. They are configured in your machine-wide or mode config with settings that end in *_events*. For example, if a device has a setting for *enable_events*: and you add an event to that setting, then when that event is posted, the device will enable. You can add single events or lists of events to these settings, and you can also configure time-delays for how much time passes between the event being posted and the action to take place. Details are available in the [device control event documentation](#).

GIs make use of the following device control events:

enable_events:

Enables this GI string. Default is *machine_reset_phase_3*.

disable_events:

Disables this GI string. Default is *None*.

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

hardware

The `hardware:` section of the config files lets you specify options that relate to the specific pinball controller hardware you're using.



This section can be used in your machine-wide config files.



This section *cannot* be used in mode-specific config files.

Here's a sample:

```
hardware:
  platform: p_roc
  driverboards: wpc
```

platform:

Specifies which hardware platform you're using.

Options include:

- `p_roc` (Multimorphic P-ROC)
- `p3_roc` (Multimorphic P3-ROC)
- `fast` (FAST Core or WPC controllers)
- `virtual` (Software-only "virtual" hardware platform that doesn't require any P-ROC or FAST drivers to be installed). This is the default option if you don't add a `platform:` section to your config file.
- `openpixel` (Open Pixel Control-based LEDs)
- `fadecandy` (LEDs connected to a FadeCandy)
- `smartmatrix` (RGB LED color DMD)

driverboards:

Specifies whether you're using Williams, Stern, or other driver boards. Typically this will match the platform of the game you're writing for (if you're modifying an existing game), or if you're building your own game, it will be the custom board type from your hardware vendor.

- `wpc95`
- `pdb` (This is "P-ROC Driver Boards", which means you're using the PD-16, PD-8x8, etc.)
- `fast`

- `wpc`
- `wpcAlphaNumeric` (Note that we have not yet added segmented display support to our media controller)
- `sternSAM` (See our [blog post on getting MPF running with a Stern S.A.M. machine](#) for notes.)
- `sternWhitestar` (Not actually tested but should work since the P-ROC supports it.)

Optional "per device" platform overrides

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 a hardware platform on a device class level (e.g. all switches are P-ROC, all LEDs are openpixel) or on a device-by-device basis (some LEDs are from the FAST controller, others are from the FadeCandy).

To specify that a certain class of devices should use a platform other than the default platform, add an entry for that type of device and the platform you want to use to the hardware: section of your config. For example, if you want to use a P-ROC for most things but a FadeCandy for your LEDs, you would specify it like this:

```
hardware:
  platform: p_roc
  driverboards: pdb
  leds: fadecandy
```

You can specify the following types of hardware in this way:

- coils
- switches
- matrixlights
- leds
- dmd
- gis
- flashers

Further, if you want to override the platform on an individual device basis, you can add a `platform:` setting to the device itself. For example, here are two LEDs, the first using the default hardware platform, and the second using a FadeCandy:

```
leds:
  led00:
```



```

    number: 00
  led01:
    number: 00
    platform: fadecandy

```

high_score

The *high_score*: section of the config file contains settings for the high score mode.



This section *cannot* be used in your machine-wide config files.



This section *can* be used in mode-specific config files.

There's a full [How To guide](#) which walks you through setting up the high score mode, so be sure to read that for the details. This page just contains the settings which control how the high score mode behaves. Here's an example config:

```

high_score:
  shift_left_tag: left_flipper
  shift_right_tag: right_flipper
  select_tag: start
  award_slide_display_time: 4s
  categories:
  - score:
    - GRAND CHAMPION
    - HIGH SCORE 1
    - HIGH SCORE 2
    - HIGH SCORE 3
    - HIGH SCORE 4

```

shift_left_tag:

This is the name of a tag applied to a switch which causes the character picker to shift the highlighted character to the left when a switch with this tag is activated. The default is `left_flipper`, which means in order for this to work, you'd add `left_flipper` to your left flipper button. For example:

```

switches:
  s_lower_left_flipper:
    number: 0A
    tags: left_flipper

```

shift_right_tag:

Just like the `shift_left_tag`, but the name of the tag for the switch that will shift the selected character to the right.

select_tag:

The tag of the switch that, when activated, will select the currently-highlighted character. (Usually this will add the highlighted letter to the list of entered characters, but there are also characters for *back* and *end*, so when this switch is hit, it will do whatever action is highlighted.)

award_slide_display_time:

The amount of time (entered in MPF time string format) which controls how long the award slide is shown on the display after the player enters their name or initials. Once this time passes, the high score mode will move on to collecting the initials for the next player and/or finish the game ending process and move on to the attract mode.

categories:

The list of player variables you want to track for each high score category, with sub-entries under each for the award names (labels) for each position.

images

You use the `images:` section of your configuration file to specify additional non-default settings for any images you want to use in your game.



This section can be used in your machine-wide config files.



This section can be

used in your mode-specific config files.

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 differ from other assets in that asset's folder. (See the [assetdefaults:](#) section for details. Also be sure to read the section on [Managing Assets](#) for an overview of how assets work.)

Here's a example:

```
images:
  jackpot:
    alpha_color: 0
    target: dmd
  happy_face:
    alpha_color: 15
```

Since Images are just a type of Asset in MPF, there are a few config settings that are generic which apply to all types of assets, so read the [reference documentation on Assets](#) for details about how the `name`, `file:`, and `load:` settings work.

Then on top of the defaults, images have two specific settings:

target:

Specifies whether the "target" display for this image will be a DMD or on screen window. This setting tells MPF whether it needs to convert a full-color image (JPG, PNG, etc.) down to the 16-shade palette for use on a DMD.

Use the value "dmd" here (so, `target: dmd`) when you want to load a traditional image file (JPG, PNG, BMP, etc.) for use on a DMD. If you want to load a traditional image and use it on the screen, you don't need to specify a `target`.

You only need to use this setting when you have traditional image file formats you want to display on a DMD. If you load a `.dmd` file, MPF assumes it's going to the DMD.

alpha_color:

This is a color that will be rendered as transparent, allowing any display elements at a lower layer to show through. For DMD files, or for image files that you're targeting for the DMD, this is a shade value from 0-15. If this is a 24-bit image then you can specify the color that will be transparent.

Alpha and blending functionality will be expanded in the future with support for RGBA and proper alpha channels.

info_lights

The `info_lights:` section of the configuration 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 uses lights to tell you things like which player is up, what ball you're on, etc.



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 `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.

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`... If your machine matches by the ones digit, then you'd enter these items at `match_1`, `match_2`, etc.

The `light:` entry in each of these is the light name from the `matrixlights:` section of your config file.

ball_x:

This maps the ball-in-play number to the light.

player_x:

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.

This plugin is pretty basic, but it should meet the needs for EM machines. Right now it only works with Matrix Lights, so if you need it for LEDs then post a request to the forum and we can change it. (Should be pretty simple.)

keyboard

The *keyboard:* section of the config files is used to configure options for how you map computer keyboard keys to pinball machine switches.



This section can be used in your machine-wide config files.



This section *cannot* be used in mode-specific config files.

(An overview of the Keyboard Interface which explains this concept is [here](#).)

```
keyboard:
  z:
    switch: flipperLwL
  slash:
    switch: flipperLwR
  s:
    switch: start
  1:
    switch: trough1
    toggle: True
    start_active: True
  2:
    switch: trough2
```

```

toggle: True
start_active: True
shift-p:
  switch: lockPost
  invert: True
q:
  event: machine_reset
ctrl-shift-4:
  event: advance_reel_test
  params:
    reel_name: score_1p_10
    direction: 1

```

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:

- comma
- period
- equals
- minus
- backquote
- leftbracket
- rightbracket
- slash
- backslash
- asterisk
- plus

You can see a complete list in the [Pygame documentation on key symbols](#). Pretty much you can use anything that's after the "K_" in the list.

Note that you can't use the Escape key because that's currently hard-coded to exit out of MPF when you hit it. (We'll change that in the future so you can configure 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 than 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 four different ones:

- shift
- ctrl
- alt
- meta (Exact meaning depends on the platforms. It's the Command key on a Mac, and the CTRL or Windows key on Windows.)

You can add one (or more) modifier keys by adding the modifier name with a dash (minus sign) before the key, like this:

```
t:
  switch: foo
shift-t:
  switch: tilt
shift-ctrl-t:
  switch: slamTilt
```

Modifier key names are not case sensitive and can be added in any order. (i.e. `shift-ctrl-t` is the same as `ctrl-shift-t`.)

We use modifier keys for lesser-used entries. The tilt is a good example, like in the snippet above. We might use the `T` key during regular game play for some switch, but if we want to hit the plumb bob tilt then we'll hit `CTRL+T`, and if we want a slam tilt then we'll hit `CTRL+SHIFT+T`.

Options for each key & key combination

Once you enter the key 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:

switch:

The switch name of the pinball machine switch you want this key (or key combination) to control.

toggle:

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
  start_active: True
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*.

invert:

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.

event:

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.

mc_event:

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:

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*. This complete entry is the equivalent of the following game code:

```
self.machine.events.post('advance_reel_test', reel_name=score_1p_10,
direction=1)
```

languages

Contains a list of the language translations that are available for on-the-fly replacement of text strings, sound files, and animations.



This section can be used in your machine-wide config files.



This section *cannot* be used in mode-specific config files.

```
languages:
  french
  english
```

The list of languages in this entry represents the current language options and the order they will be searched in whenever a language replacement search is done.

Note that these names are completely arbitrary. They only have to match with the sections you have in your TextStrings configuration. You could call them *adult* and *family_friendly* if you wanted to use different text strings even though they're all in English.

language_strings

The `language_strings:` section of your config file has a list of text strings that will be replaced for a certain language.



This section can be used in your machine-wide config files.



This section *cannot* be used in mode-specific config files.

```
language_strings:
  french:
    press start: APPUYEZ SUR START
    free play: JEU GRATUIT
```

```

player: JOUEUR
ball: BALLE
german:
  press start: START DRÜKEN
  free play: FREISPIEL
  player: SPIELER
  ball: BALL

```

The layout of this section of your config file should be pretty obvious.

By the way we don't speak French *or* German, so we used Google Translate for these just to make examples. They're more to prove the point. :)

Note that in order to support the U with the umlaut in "DRÜKEN" then you need to save the config file as UTF-8. Also you need to make sure that your font has a glyph for those characters. (The current default font we use with MPF does not. It's on our to-do list though.)

light_player

You can use the `light_player:` section of your config files play light shows and scripts based when certain MPF events happen. 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 section can be used in your machine-wide config files.



This section can be used in mode-specific config files.

Here's an example:

```

light_player:
  ball_save_active:
    lights: shoot_again
    script: flash
    tocks_per_sec: 4
    repeat: yes
  tilt_warning:
    led_tags: playfield
    script: flash_red
    key: tilt_warning
    tocks_per_sec: 10
  tilt_warning_expired:
    key: tilt_warning
    action: stop
  shot_ramp_made:
    show: left_side_big_flash
    tocks_per_sec: 10
    repeat: no

```

The Light Player setting above will apply a Light Script called *flash* to the light *shoot_again* when the *ball_save_active* event is posted. It will play that script at 4 tocks per second, and it will repeat forever until it's stopped.

It will also play the script *flash_red* on all the LEDs tagged with *playfield* when the *tilt_warning* event is posted. That script runs at 10 tocks per second and will not repeat.

Finally, it will play a light show called *left_side_big_flash* when the *shot_ramp_made* event is posted. That show will also play at 10 tocks per second and will not repeat.

Specific settings for the `light_player`: are:

<event_name>:

The top-level entries under `light_player`: are the names of the events that will activate the settings underneath them.

show:

The name of a light show that will be played when the *<event_name>* is posted. Light shows contain lists of all the individual lights or LEDs for each step. So if you enter a show here, then you do not need to specify `lights`, `leds`, `light_tags`, or `led_tags`.

lights:

A list of one or more matrix lights (entered in MPF's [config list format](#)) that a light script will apply to.

leds:

A list of one or more LEDs (entered in MPF's [config list format](#)) that a light script will apply to.

light_tags:

A list of one or more tags for lights this light script will apply to.

led_tags:

A list of one or more tags for LEDs this light script will apply to.

script:

The name of the script that will be applied to the lights, leds, light_tags, and/or led_tags in this section. You define your scripts in the [light_scripts:](#) section of your config file.

tocks_per_sec:

How many tocks per second this light script or show should play at.

repeat:

Whether this script or show should repeat.

action:

Specifies whether this option starts the show / script or stops it. Options include:

- `start` - starts the show or script. (This is the default implied value, so if you don't include an `action:` setting then `action: start` is implied.)
- `stop` - stops the show or script. In order to stop a show, you also need to include a `show:` entry with the name of the show you're stopping. In order to stop a script, you also need to include a `key:` entry with the name of the key you specified when you started the script.

Note that when you stop a show or script, you can optionally include `reset: yes` and/or `hold: yes` as settings for that entry to reset the show or script to the beginning and to hold the lights or LEDs in their final states. If you don't specify these entries, the defaults are `reset: yes` and `hold: no`.

key:

You can enter an arbitrary text string as a "key" when you start playing a script which you can then later use to stop that script. This entry from the example above includes a key:

```
tilt_warning:
  led_tags: playfield
  script: flash_red
  key: tilt_warning
  tocks_per_sec: 10
```

Why do you need a key? Unlike light shows, light scripts aren't created for specific lights or LEDs—instead you specify them when you play them. You might have a simple script called "flash" that you use to cause a light to flash. When it comes time to stop that script, how do

you identify which light you want to stop it on? You might have 20 different lights flashing but you only want to stop one. This is where the "key" comes in. When you play a script on a certain light, led, list of lights/leds, or tags of lights/leds, you can optionally specify a "key" that you can use to later stop that script on those lights or LEDs. The key can be anything you want.

In the example above, we play our script called "flash_red" on all the LEDs tagged with "playfield", and we give that a key called "tilt_warning". Then we later stop that flashing when the event `tilt_warning_expired` is posted:

```
tilt_warning_expired:
  key: tilt_warning
  action: stop
```

We could have also just configured this script to play a few times and stop on its own, like this:

```
tilt_warning:
  led_tags: playfield
  script: flash_red
  tocks_per_sec: 10
  repeat: yes
  num_repeats: 3
```

In this case we used the `key`: just to show an example of how it works.

Other show settings:

When you're setting up a light show to play, you can enter any settings for shows you want, including `repeat`, `priority`, `blend`, `hold`, `tocks_per_sec`, `start_location`, and `num_repeats`. See the documentation on [playing shows](#) for details.

light_scripts

You can use the `light_scripts`: section of your machine config files to create "scripts" which control how lights or LEDs behave. Then you can use the `light_player`: section to apply those scripts to specific lights or LEDs.



This section can be used in your machine-wide config files.



This section *cannot* be used in mode-specific config files.

Light scripts are very similar in concept to light shows. The difference is that when you create a light show file, you include the specific lights in the show itself. With light scripts, you create a generic script you can apply to any light (or lights) later on.

For example, you might create a show file that says "turn on *light_shoot_again*, then *light_bonus_2x*, then turn them both off." Every time you run that show, it applies to the lights *light_shoot_again* and *light_bonus_2x*. A script, on the other hand, might say "turn the LED red, then yellow, then off". There's no light or LED name specified in the script, and instead you can later apply that script to any light you want.

So in the `light_scripts:` section of your config file, you actually create a listing of all the different scripts you want to use later in your game. You don't actually apply them at this point. You just create them. For example:

```
light_scripts:
  flash:
    - tocks: 1
      color: ff
    - tocks: 1
      color: 0
  on:
    - tocks: 1
      color: ff

  rainbow_cycle:
    - tocks: 1
      color: ff0000
    - tocks: 1
      color: ff5500
    - tocks: 1
      color: ffff00
    - tocks: 1
      color: 00ff00
    - tocks: 1
      color: 0000ff
    - tocks: 1
      color: ff0099
```

You'll notice that the script format is similar to the [show format](#) except that the script doesn't contain any light names.

The `light_scripts:` section above has three scripts in it: *flash*, *on*, and *rainbow_cycle*. Scripts that you will apply to [Matrix Lights](#) only use `00` (off) or `ff` (on), while scripts which you'll apply to [RGB LEDs](#) have six-character hex color codes.

Notice that each step of the script starts with a dash (and the space between the dash and the word "tocks" is important). The "tocks" represents the relative timing of one step versus the next. (Tocks have nothing to do with real-world time, since when you run a script you can specify how many tocks per second it runs at.)

Once you create your scripts, you can apply them to certain lights based on MPF events with the `light_player:` section of your config. (The `light_player:` section can exist in your machine-wide config file or a mode-specific config file.)

logic_blocks

The `logic_blocks`: section of your config file contains subsections for each type of [logic block](#) in your game, including [accruals](#), [counters](#), and [sequences](#).



This section *cannot* be used in your machine-wide config files.



This section can be used in your mode-specific config files.

You can read about the configuration details for each type of logic block from the links above. However, there are several types of settings that apply to all logic blocks, so we're including them once here instead of copying-and-pasting them into the sections on each logic block.

Settings that apply to all types of Logic Blocks

<name>:

You configure your individual Logic Blocks in the `logic_blocks`: section of your config files. The actual name you pick for each Logic Block doesn't matter. It's just a friendly name that will make sense to you.

events_when_complete:

This is a [list of events](#) (or a single event) which are posted when this logic block is complete.

If you don't have a configuration entry here, then MPF will automatically post an event with the name `logicblock_<your_logic_block>_complete` when the logic block is complete.

enable_events:

This is a [list of events](#) (or a single event) which, when posted, causes this logic block to start watching for events to track progress towards completion. If your logic block is not active, then any events that fire won't count towards progress. Default for this is *None* (meaning you'd have to manually enable this logic block in code).

disable_events:

This is a [list of events](#) (or a single event) which, when posted, causes this logic block to stop watching for events and to stop tracking progress towards completion. If any one of the events is posted while this logic block is disabled, it will be ignored. Default for this is *None* (meaning you'd have to manually disable this logic block in code), or that it will be disabled when the mode this logic block's config is in stops.

reset_events:

This is a [list of events](#) (or a single event) which, when posted, resets this logic block back to its default state. Default is *None*.

restart_on_complete:

Yes/No (or True/False) which controls whether this logic block should reset and restart itself when it completes. Default is *false*.

disable_on_complete:

Yes/No (or True/False) which controls whether this logic block should disable itself when it completes. Default is *true*.

persist_state:

If this is set to *true*, then this logic block will not reset its state each time it's loaded. This means that you have "long running" logic blocks that persist across balls. (Remember though that logic blocks are still tracked on a per-player basis.) Default is *false*.

leds

The *leds:* section of the machine configuration file controls mappings and settings for [RGB LEDs](#) in your pinball machine.



This section *can* be used in your machine-wide config files.



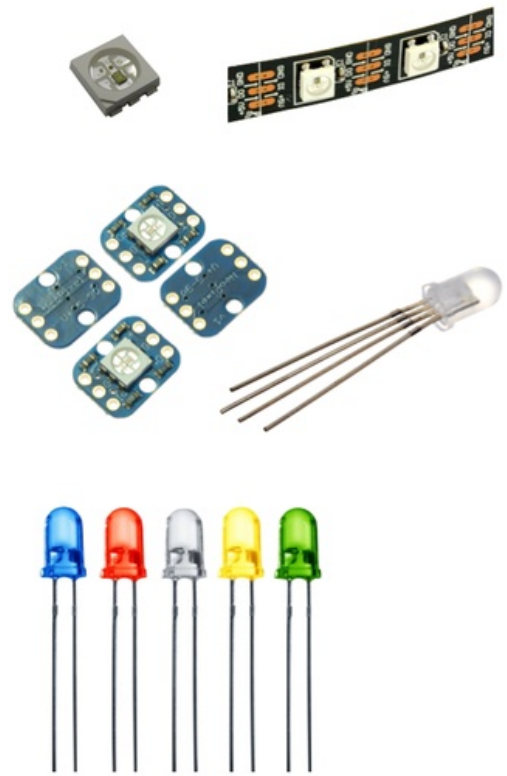
This section *cannot* be used in mode-specific config files.

Note: This section is for LEDs connected to LED controller boards, such as the Multimorphic PD-LED, the FAST Pinball controllers' RGB LED outputs, or a third party LED controller like a FadeCandy. This section is *not* for "drop in" replacement LEDs (like those from CoinTaker) which you put into existing machines, so those are [Matrix Lights](#) as far as MPF is concerned:

“MatrixLights”



“LEDs”



Here's an example *leds:* section from a machine configuration file:

```
leds:  
  l_led0:  
    number: 00  
    brightness_compensation: 0.9  
    tags: playfield  
    fade_ms: 100  
  l_led1:  
    number: 01  
    brightness_compensation: 1.0, 0.9, 1.0  
    tags: playfield  
    fade_ms: 100  
    debug_logging: true  
  l_shoot_again:  
    number: 02  
    tags: lower playfield  
    label: Shoot Again Light
```

<LED Name>:

Just like the other parts of the configuration file, each entry under *leds:* represents the name of the LED as it will be used in the game code.

number:

This is the hardware address of the LED. The exact format depends on what type of LED Controller you have. Click on the links below to go to the how to guides for each platform which have the details of how their numbering works.

- [PD-LED boards \(used with a Multimorphic P-ROC or P3-ROC\)](#)
- [FAST Pinball Controllers](#)
- [FadeCandy \(also can be used with Multimorphic or FAST hardware\)](#)
- Open Pixel Control hardware (can be used in conjunction with Multimorphic or FAST hardware, see below for numbering)

LED numbering with Open Pixel Control LEDs:

You can use MPF's *openpixel/* platform interface to control LEDs via the [Open Pixel Control \(OPC\) protocol](#). For each LED, you need to configure the channel number and LED number. You do this by specifying the channel number, then a dash, then the LED number:

```
l_led0:
  number: 0-0
l_right_ramp:
  number: 0-1
l_space_ship:
  number: 1-47
```

Note that the FadeCandy LED controller is a valid OPC controller, but if you use it via the *fadecandy* interface rather than the *openpixel* interface, you can take advantage of advanced features of the FadeCandy hardware.

brightness_compensation:

Allows you to configure the brightness compensation for this RGB LED. (Read more about brightness compensation in our description of how MPF supports RGB LEDs.) Brightness compensation is configured as a floating point value. You can enter a single value (which will be used for all three R, G, and B elements in this LED, or you can configure three separate values.

```
l_led0:
  number: 0-0
```

```

    brightness_compensation: .85
l_right_ramp:
    number: 2-28
    brightness_compensation: 1.0, 1.0, .9

```

In the above example, `L_led0` is configured for a brightness compensation of `.85`, which means that every LED color command that's sent to `led0` will be multiplied by 0.85. (In other words, this LED will always only run at 85% brightness.)

For LED `L_right_ramp` in the example above, the red and green elements have a brightness compensation of 1.0 (which means they're actually not compensating), while the blue element has a brightness compensation of 0.9. This means the blue value will always only show at 90% brightness as compared to the red and green. (For example, setting the `right_ramp` to color values `[255, 255, 255]` will actually send values of `[255, 255, 230]`.) This is nice for situations where you have an LED package where one of the elements is out of whack and you want to tune it to match the others.

You can also specify brightness compensation values higher than 1.0 (for individual elements and entire LEDs), which can be nice if you have elements that are going bad that you want to crank up a bit, though no matter what MPF cannot send a value higher than 255 to the hardware.

Also remember that there is a global brightness compensation setting (in the configuration files at [led_settings: brightness_compensation](#): which lets you (or the operator) "turn down" all the LEDs in the game. (No more blinding your eyes out in a game with RGB LEDs in a dark bar!) If the global brightness compensation setting is configured, then it will be combined with the brightness compensation settings for each individual LED, meaning you can still tune individual LED elements even if you are using the global setting.

default_color:

You can specify a default color for this LED that will be used when the LED is activated by a command which just turns it "on." This default color is a six-character hex string in `rrggbb` format, like `ff0000` for red, `ffff00` for yellow, etc.

Note that you can override this default by specifying the color of the LED at any time. The default is just used if something asks the LED to turn on so it knows what color to use.

Device Control Events

Device control events are events you can use to control devices. They are configured in your machine-wide or mode config with settings that end in `_events`. For example, if a device has a setting for `enable_events`: and you add an event to that setting, then when that event is posted, the device will enable. You can add single events or lists of events to these settings, and you can also configure time-delays for how much time passes between the event being

posted and the action to take place. Details are available in the [device control event documentation](#).

LEDs make use of the following device control events:

on_events:

Enables this LED with its default color. Default is *None*.

off_events:

Disabled this LED. Default is *None*.

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

led_settings

The `led_settings:` section of your machine configuration files lets you configure settings for RGB LEDs in your machine.



This section can be used in your machine-wide config files.



This section *cannot* be used in mode-specific config files.

This section is included in the `mpfconfig.yaml` file. You only need to add them to your own machine config files if you want to override a default

```

led_settings:
  brightness_compensation: 1.0
  default_led_fade_ms: 100
  gamma: 2.5
  whitepoint: 1.0, 1.0, 1.0
  linear_slope: 1.0
  linear_cutoff: 0.0
  keyframe_interpolation: True
  dithering: True

```

brightness_compensation:

The `brightness_compensation` multiplier that will be used for all LEDs in your machine. This is applied to all LEDs, even if they have their own brightness compensation settings. (So if you have a global brightness compensation setting of 0.85, and an individual LED has a brightness compensation setting of 0.9, then sending that LED a "full" brightness of 255 will actually send it a value of 195. ($255 * 0.85 * 0.9$) This is done on purpose so you can set machine-wide "dimming" here at the global level while still being able to tweak relative brightness levels of individual LEDs.

default_led_fade_ms:

This is the default `fade_ms` that will be applied to individual RGB LEDs that don't have `fade_ms` settings configured. If you configure an individual LED's `fade_ms`, it will override this setting.

Note that this feature currently only works with LEDs connected to a Multimorphic PD-LED board. FadeCandy and FAST controller support will be added in the future.

gamma:

Specifies the [gamma correction](#) value for the LEDs. The default is 2.5.

This setting currently only affects LEDs connected to a FadeCandy LED controller.

whitepoint:

Specifies the white point (or white balance) of your LEDs. Enter it as a list of three floating point values that correspond to the red, blue, and green LED 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.

This setting currently only affects LEDs connected to a FadeCandy LED controller.

linear_slope:

Specifies the slope (output / input) of the linear section of the brightness curve for the LEDs. The default is 1.0.

This setting currently only affects LEDs connected to a FadeCandy LED controller.

linear_cutoff:

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 LEDs 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.

The default value is 1.0.

This setting currently only affects LEDs connected to a FadeCandy LED controller.

keyframe_interpolation:

Boolean value that enables keyframe interpolation to smooth the color transition of LEDs in between machine ticks.

For example, if your machine Hz rate is 30, that means the fastest MPF can update LEDs is 30 times per second. While that's really fast, some players might perceive step "steps" between each step of the color change—especially if LEDs are going from off to full brightness.

Enabling keyframe interpolation means the LED controller hardware will automatically in color frames in between color updates from MPF. For example, when keyframe interpolation is enabled, the FadeCandy LED controller will update all the LEDs 400 times per second, generating in-between color values which make the LEDs look smooth. The results are beautiful.

This setting is enabled by default.

This setting currently only affects LEDs connected to a FadeCandy LED controller.

dithering:

Boolean value that controls whether the hardware LED controller will use dithering to simulate richer and smoother colors than the hardware LEDs can provide. (More details are available in the [FadeCandy documentation](#).)

This setting is enabled by default.

This setting currently only affects LEDs connected to a FadeCandy LED controller.

machine

The `machine:` section of the configuration files holds settings that relate to the physical pinball machine itself.



This section can be used in your machine-wide config files.



This section *cannot* be used in mode-specific config files.

For now this includes:

```
machine:
  balls_installed: 6
  min_balls: 3
  glass_off_mode: true
```

Details about these options:

balls_installed:

This is the number of balls that should be installed in the machine. The operator will get warnings if the number is something else.

min_balls:

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.

glass_off_mode:

This setting controls what happens when a playfield switch is hit when a game is not in progress. In a "final" machine on location, if a playfield switch is hit, MPF will think there's a

live ball on the playfield. This is great functionality to have, but it's super annoying when you're testing your machine.

The problem is that if you hit a switch during attract mode, MPF will think there's a ball on the playfield and you won't be able to start a game until that ball drains. But if you have the glass off and you hit a switch with your finger, you essentially "break" MPF since there was never a ball there (meaning it will never drain), and that's annoying.

So enabling *glass_off_mode* means that if a switch is hit with no ball in play, MPF will not think there's a ball on the playfield and you can still start a game. The default setting is *True*, and you can set this to *False* once you finish your machine.

matrix_lights

The *matrix_lights*: section of the machine configuration file is used to map [lights connected to a lamp matrix](#) (whether incandescent lamps or LEDs) to driver board outputs.



This section can be used in your machine-wide config files.



This section can be used in mode-specific config files.

Here's a snippet of the *matrix_lights*: section of the configuration file for *Judge Dredd*:

```
matrix_lights:
  l_award_sniper:
    number: L64
    label: Sniper Shot Lit
    tags: playfield, white
  l_air_raid:
    number: L65
  l_left_center_feature:
    number: L66
  l_tank_left:
    number: L67
  l_mystery:
    number: L68
  l_drop_target_j:
    number: L71
  l_drop_target_u:
    number: L72
  l_drop_target_d:
    number: L73
  l_drop_target_g:
    number: L74
```


<light_name>:

Each subsection of *matrix_lights*: is the name of the light as you'd like to refer to it in your game code. This can really be anything you want, but it's obviously best to pick something that makes sense.

number: (required)

The way you enter the number for a light depends on the type of pinball controller hardware you're using and what type of pinball machine it is. Refer to the following sections for information about how the numbering works for those platforms:

- [P-ROC with a lamp matrix connected to a PD-8x8](#)
- [P3-ROC with a lamp matrix connected to a PD-8x8](#)
- P-ROC or FAST WPC Controller in a WPC machine (keep reading below)

With WPC hardware, the number is simple. It's just a capital letter "L" followed by the number of the light from the operators manual. The Mission Pinball Framework and the platform drivers handle the hard work of mapping these numbers to actual column and row outputs.

Note that many of the Williams operators manuals have typos here which no one has probably noticed in 25 years. But if you find that the proper light is not lighting up when you think it should, you might have to experiment a bit.

Device Control Events

Device control events are events you can use to control devices. They are configured in your machine-wide or mode config with settings that end in *_events*. For example, if a device has a setting for *enable_events*: and you add an event to that setting, then when that event is posted, the device will enable. You can add single events or lists of events to these settings, and you can also configure time-delays for how much time passes between the event being posted and the action to take place. Details are available in the [device control event documentation](#).

Matrix lights make use of the following device control events:

on_events:

Turns on this light. Default is *None*.

off_events:

Turns off this light. Default is *None*.

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

media_controller

The `media_controller:` section of the config file controls the overall behavior of the MPF media controller process. It's analogous to the `mpf:` section which is used by the MPF core engine. This config section is included in the `mconfig.yaml` file which is the default that's always read in first whenever the media controller starts up. You shouldn't have to change anything here, though you can override specific settings if you want.



This section can be used in your machine-wide config files.



This section *cannot* be used in mode-specific config files.

Here's the default `media_controller:` section from `mconfig.yaml`:

```
media_controller:
  modules:
    modes.ModeController
    language.Language
    display.DisplayController
    show_controller.ShowController
    keyboard.Keyboard
```

```

    sound.SoundController

port: 5050
exit_on_disconnect: yes
display_modules:
  elements:
    - text
    - virtualdmd
    - image
    - animation
    - shape
    - movie
  modules:
    dmd.DMD
  transitions:
    move_out: move_out.MoveOut
    move_in: move_in.MoveIn
  decorators:
    blink: blink.Blink
paths:
  shows: shows
  sounds: sounds
  machine_files: machine_files
  config: config
  fonts: fonts
  images: images
  animations: animations
  movies: movies
  modes: modes

```

modules:

Lists the core modules that will be loaded with the media controller boots.

port:

The TCP port number the media controller will listen on for an incoming BCP connection from a pinball controller.

exit_on_disconnect:

Whether you want the media controller to automatically shut down and exit when the connected pinball controller disconnects. Default is yes.

display_modules:

Lists the various display modules that will be available to the media controller, including display elements, types of displays supported, transitions, and decorators.

paths:

Specifies the default folder names which will be used in a machine's folder to hold different types of media assets.

mode

The `mode:` section of a configuration file is used to specify general settings for a particular mode.



This section *cannot* be used in your machine-wide config files.



This section *can* be used in mode-specific config files.

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 is for the settings for each specific mode in a mode configuration file.)

Let's take a look at an example `mode:` section from a multiball mode:

```
mode:
  start_events: ball_starting
  stop_events: timer_mode_timer_complete, shot_right_ramp
  code: skillshot.SkillShot
  priority: 300
```

Let's look at each of these settings:

start_events:

The name of an MPF event (or a list of multiple events) that, when posted, cause this mode to start. (To enter multiple events, see the [note about adding lists to configuration files](#).) If you add more than one event here, then any one of the events will cause the 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.

stop_events:

Like `start_events`, except a `stop_event` causes 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.

code:

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.

priority:

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 (display shows, light shows, 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 you're 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.

stop_priority:

This optional setting allows you to 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`.

start_priority:

Like `stop_priority`, but it controls the order modes start it. If you have a mode you want to start before others, enter a value here.

use_wait_queue:

True/false setting which 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.

More details about how this works are available in the [How to create a bonus mode](#) guide.

stop_on_ball_end:

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` to the mode: section of a mode's config file. Default is `true`.

restart_on_next_ball:

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 the untracked player variable `_restart_modes_on_next_ball` which maintains a list of the mode to be restarted.

modes

The `modes:` section of your machine configuration files is where you specify which modes will be used in your game. (Read more about how game modes work [here](#).)



This section can be used in your machine-wide config files.



This section *cannot* be used in mode-specific config files.

This section is just a list of modes (based on the folder names in your machine's modes folder. This `modes:` list has nothing to do with whether modes start or stop, or what priority they run at. (Those settings are specified in the [mode: section](#) of each mode's own config file.) Instead this is just a list of all the modes that are available in a game. If you have a mode that you're working on and is not yet ready, don't include it here and it won't be processed or loaded.

```
modes:
- skillshot
- base
- both_ramps_made
- gun_fight
- multiball
- skillshot
- watch_tower
```

Enter your list of modes (again, based on the folder names) like any [config entry list](#).

multiballs

The `multiballs:` section of the config file 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.

```
multiballs:
  main_multiball:
    ball_count: 3
    shoot_again: 5s
    ball_locks: bunker
```

<name>:

ball_count: (required)

Integer value of the number of balls this multiball will be.

source_playfield:

The name of the playfield these balls will be added to. If you only have one playfield (which is most games), you can leave this setting out. Default is the playfield called *playfield*.

shoot_again:

A time period where drained balls will automatically be re-launched. (Kind of like a multiball-specific ball save). Default is *10 seconds*.

ball_locks:

A list of one or more [ball lock devices](#) that will be used as the source for the balls for this multiball. When multiball begins, it will first try to eject balls from these ball locks to get the required number of balls in play. If these devices don't have enough balls, then it will add balls from the default device that adds balls into play. (Typically the trough / plunger lane.)

Device Control Events

Device control events are events you can use to control devices. They are configured in your machine-wide or mode config with settings that end in *_events*. For example, if a device has a setting for *enable_events*: and you add an event to that setting, then when that event is posted, the device will enable. You can add single events or lists of events to these settings, and you can also configure time-delays for how much time passes between the event being posted and the action to take place. Details are available in the [device control event documentation](#).

Multiball devices make use of the following device control events:

enable_events:

Enables this multiball device. (It must be enabled before it can be started.) Default is *ball_started*.

disable_events:

Disables this multiball device. Default is *ball_ending*.

reset_events:

Resets this multiball device. Disables it, cancels *shoot_gain*, and resets the balls ejected count to 0. Default is *machine_reset_phase_3, ball_starting*.

start_events:

Starts this multiball by adding the configured number of balls into play. Default is *None*.

stop_events:

Stops this multiball by stopping shoot again. Note that short of disabling the flippers, you can't really stop a multiball. Default is *None*.

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

mpf

The `mpf` section of the machine configuration file contains settings that affect the actual framework itself (as opposed to game-specific settings). In most cases you will never need change any of these settings, but they are available to you if you want.



This section can be used in your machine-wide config files.



This section *cannot* be

used in mode-specific config files.

Here's the default MPF section that the Mission Pinball Framework uses:

```
mpf:
  system_modules: !!omap # order is important here
    - timing: mpf.system.timing.Timing
    - events: mpf.system.events.EventManager
    - switch_controller: mpf.system.switch_controller.SwitchController
    - ball_controller: mpf.system.ball_controller.BallController
    - modes: mpf.system.modes.ModeController
    - light_controller: mpf.system.light_controller.LightController
    - bcp: mpf.system.bcp.BCP
    - shots: mpf.system.shots.ShotController
    - logic_blocks: mpf.system.logic_blocks.LogicBlocks
    - scoring: mpf.system.scoring.ScoreController
    - target_controller: mpf.system.target_controller.TargetController

  device_modules:
    driver.Driver
    switch.Switch
    matrix_light.MatrixLight
    led.LED
    gi.GI
    autofire.AutofireCoil
    ball_device.BallDevice
    drop_target.DropTarget
    drop_target.DropTargetBank
    target.Target
    target.TargetGroup
    flipper.Flipper
    diverter.Diverter
    score_reel.ScoreReel
    score_reel.ScoreReelGroup
    flasher.Flasher
    playfield_transfer.PlayfieldTransfer

  plugins:
    auditor
    ball_save
    ball_search
    info_lights
    osc
    socket_events
    switch_player

  paths:
    scriptlets: scriptlets
    shows: shows
    audits: audits/audits.yaml
    machine_files: machine_files
    config: config
    modes: modes

  default_pulse_ms: 10
  default_flash_ms: 50
  auto_create_switch_events: True
  switch_event_active: "%_active"
  switch_event_inactive: "%_inactive"
  switch_tag_event: sw_%
```

```

device_events:
  autofires:
    enable: ball_started
    disable: ball_ending, tilt, slam_tilt
  flippers:
    enable: ball_started
    disable: ball_ending, tilt, slam_tilt
  drop_targets:
    reset: ball_starting
    enable:
    disable:
  targets:
    enable: ball_started
    disable: ball_ending, tilt, slam_tilt
    rotate_left:
    rotate_right:

```

system_modules:

This is a list of the system modules that MPF will load from the /system folder.

device_modules:

This is the list of the device modules that MPF will load from the /devices folder. (Note that not all device types are used in every machine, and MPF will only load device modules that have corresponding devices specified in the machine configuration files.

plugins:

The list of plugins that MPF will attempt to load when it boots. Note that some plugins require certain sections to exist in the config files, and if they're not there, then they don't load. (For example, if you don't have a `socket_events:` section in your config, then the `socket_events` plugin won't load.)

paths:

This is the subfolder location where where MPF looks for certain things in each machine's configuration folder. `scriptlets:` `scriptlets` means that MPF looks for scriptlets in the /`scriptlets` folder, etc. You can override this in your own configuration files if you want.

default_pulse_ms:

Number of ms to use to pulse coils for the coils in your configuration file where you have not specified a default. (You can always pass a parameter to a coil's pulse method too in order to pulse it at whatever you want, like `coil.pulse(10)` or whatever.

default_pulse_ms:

The default pulse time, in ms, of coils.

default_flash_ms:

The default pulse time, in ms, of flashers.

auto_create_switch_events:

True/False which controls whether MPF will automatically post events when switches open and close.

switch_event_active:

The name of the switch event that's posted when a switch is activated. The percent sign is a placeholder that is replaced with the switch name. The default of "`%_active`" means that when a switch called *target1* is activated, an event called *target1_active* will be posted. (Note that "`%_active`" has quotation marks around it because a value starting with a percent sign confuses the YAML processor, so we wrap it in quotes to let it know that it's a string.)

This feature requires that `auto_create_switch_events:` is true.

switch_event_inactive:

Same as *switch_event_active* above, except it controls the name of the event that's posted when a switch goes inactive.

switch_tag_event:

This specifies the name of events that are posted when a switch is activated based on switch tags. The percentage sign is replaced with the tag name. MPF will post one event for each tag. For example, if you have a switch that's tagged with `home`, `start`, `yes`, and `switch_tag_event: sw_%`, then when that switch is hit, MPF will post the events *sw_home*, *sw_yes*, and *sw_start*.

device_events:

This section lets you map which control events will be created for various devices, as well as lets you specify the default events that are created if a particular device doesn't have any events specified in its configuration.

open_pixel_control

You can use the `open_pixel_control:` section of your machine config file to control settings for MPF LED hardware connections that use the Open Pixel Control (OPC) protocol (such as the [FadeCandy](#)).



This section can be used in your machine-wide config files.



This section *cannot* be used in mode-specific config files.

Here's the `open_pixel_control:` section of the `mpfdefault.yaml` configuration file. You don't need to add these to your own config files unless you want to override any of these defaults.

```
open_pixel_control:
  host: localhost
  port: 7890
  connection_required: no
  connection_attempts: -1
  number_format: int
```

host:

The host name of the OPC server MPF will connect to. This is almost always *localhost* since in most cases, the OPC server is running as a separate process on the computer running MPF.

port:

The TCP port the OPC server is listening on.

connection_required:

Boolean yes/no true/false which controls whether MPF needs a connection to the OPC server to run. If you enable this, then if MPF boots and can't connect to the OPC server, MPF will exit. Also if it's enabled and MPF does connect but then it loses its connection, MPF will exit.

connection_attempts:

The number of attempts MPF will make to try to connect to the OPC server. A value of 0 or less (such as the -1 default) means "unlimited" and MPF will continuously try to connect in the background. (The OPC client built into MPF runs as a separate thread, so it's able to keep trying to reconnect in the background without affecting performance.)

number_format:

Specifies whether the numbers for your Open Pixel LEDs in your config file are hex or integer format. (Use the values `hex` or `int`.)

OSC

You can configure the settings for MPF's OSC interface in the `osc:` section of your machine config file.



This section can be used in your machine-wide config files.



This section *cannot* be used in mode-specific config files.

These settings are in the default `mpfconfig.yaml` file, so they will be applied by default if you enable the OSC interface. You can modify or override them in your own machine configuration files.

```
osc:
  machine_ip: auto
  machine_port: 8000
  approved_client_ips: any
  client_port: 9000
  debug_messages: false
  client_updates:
    switches
    lights
    data
```

machine_ip:

The IP address of your pinball machine. A value of `auto` means that it will automatically use whatever the machine's IP address is. `auto` works fine in most cases, but if you have more than one network interface in your machine you might have to specify which one you want to use.

machine_port:

The UDP port of your pinball machine, i.e. the OSC interface will listen on. (In your OSC clients, this is usually called the "outgoing" port since it's the port they're sending their OSC messages to.)

approved_client_ips:

A list of approved IP addresses for OSC clients. This feature is not yet implemented, so just leave it to `any` for now.

client_port:

The UDP port the OSC interface will send client messages to. This is usually called something like the "incoming" port on your OSC client.

debug_messages:

Set this to `true` to enable debug logging all (incoming and outgoing) OSC messages. This is great when you're building your OSC clients so you can see exactly what the machine is receiving and what it's sending to the client. If you use this with light or switch updates then this will be a *lot* of lines in your log file, so it's disabled by default.

client_updates:

Specifies which types of activities you'd like to update on your OSC client. As you can imagine, it's possible that updating every light and every switch on your client is a lot of data, so if you don't use them then you can safely remove them from this list.

playfields

The *playfields:* section of the machine-wide config file is where you configure [playfields](#).



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 of a machine with two playfields:

```
playfields:
  playfield:
    tags: default
  lower_playfield:
    tags: lower
```

You don't have to include a *playfields:* section in your machine-wide config since the *mpfconfig.yaml* default config contains the settings for the default playfield, like this:

```
playfields:
  playfield:
    tags: default
```

Note that there are no settings that are specific to playfields. All you have to do is add them to the list. However, since devices in MPF require at least one setting entry, each of the playfields in the example above contain a *tags:* entry just so they have something to make the syntax correct. The tags aren't actually used for anything though.

If you have multiple playfields, it's important to for one of them to be named *playfield*. That one will be the default playfield that other ball devices will eject their balls to if you don't configure them with eject targets.

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

random_event_player

You can use the *random_event_player:* section of your config files to cause a random event from a list to be posted when a specific event is posted. This is very similar to the [event_player](#). 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 section can be used in your machine-wide config files.



This section can be

used in mode-specific config files.

Here's an example from Indiana Jones where we use it to randomly advanced one of the top lane shots to light when the skillshot starts:


```

random_event_player:
  mode_skillshot_started:
    - indy_i_advance
    - indy_n_advance
    - indy_d_advance
    - indy_y_advance

```

The random event player is very basic. One of the events from the list will be chosen and posted at random when the event entry above it is posted.

score_reels

The *score_reels*: section in the machine configuration files holds settings for mechanical score reels for EM-style pinball machines. Details about how we support mechanical score reels can be found in the [Score Reel device section](#) of this documentation.



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 of two score reel configurations from the config file:

```

score_1p_10k:
  coil_inc: player1_10k
  switch_0: score_1p_10k_0
  switch_9: score_1p_10k_9
  rollover: True
  limit_hi: 9
  limit_lo: 0
  repeat_pulse_time: 200ms
  hw_confirm_time: 300ms
score_1p_1k:
  coil_inc: player1_1k
  switch_0: score_1p_1k_0
  switch_9: score_1p_1k_9
  rollover: True
  limit_hi: 9
  limit_lo: 0
  repeat_pulse_time: 200ms
  hw_confirm_time: 300ms

```

Each section is headed with the name you'd like to refer to this score reel as in your game code. In the above example, the score reels are named *score_1p_10k* and *score_1p_1k*. It's helpful if you name each reel in some way that makes sense to you. In this example, you can infer that these reels are the 10,000s digit and the 1,000s digit from the Player 1 score reel group. (Note that the names here are arbitrary.) What's important to understand is that these entries make up the individual movable reels themselves. Then you use the [Score Reel Group section](#) to group multiple individual reels into logical groups.

Specific configuration options for score reels include:

coil_inc:

The name of the coil that's pulsed to increment the reel. The actual coil pulse time is specified in the [coils: section](#) of the config files.

coil_dec:

The name of the coil that's pulsed to decrement the reel. The actual coil pulse time is specified in the [coils: section](#) of the config files.

switch_x:

The name of the switch that is active when the reel is in the value of the "x" in the name. In the example configuration above, *switch_0* is active when the reel is showing 0, and *switch_9* is active when the reel is showing 9. All EM reels have switches in the zero position. Most have switches for 9 also, and some reels have switches for all the positions to handle replay scores and match features.

You must specify at least one switch, though it doesn't have to be at the zero position. When the Mission Pinball Framework sets the initial position of the reels, it will pulse them continuously until it learns of their positions via the activation of one of these switches. Then it "knows" (well, "assumes") the position of the reel based on how many pulses it did since the last position that was confirmed via a switch.

It's not really necessary to have more than one position switch per reel, though if you do then they provide a certain redundancy since the machine could still function if one of the switches breaks. It is definitely not necessary to wire up all ten switches for each reel. That would be overkill which doesn't really buy you anything, and it would take forever and waste a lot of switch inputs on your hardware controller.

Note that whether this switch is normally open or normally closed is specified in that [switch's settings in the machine configuration files](#). The reel itself only looks for "active" or "inactive," so if you have a switch that is open when the reel is in one position and closed for all others, be sure to configure your switch as "NC" (normally closed).

rollover:

A value of True or False that indicates whether this reel is capable of rolling over, that is, whether it can hit its limit and then take another pulse to roll back over to zero. In typical EM games, the scoring reels would have their rollover set to True, but the credit wheel would have a rollover of False.

limit_hi:

This is the numeric value of highest position on the reel. For scoring reels it would be 9. For credit reels it's whatever the highest number is on the reel.

limit_lo:

This is the lowest number on the reel, in most cases zero.

repeat_pulse_time:

This is time that the machine will wait between each "pulse" of the reel. For example, if you set this to *200ms*, and the machine needs to pulse the reel from 1 to 6, it will fire off 5 pulses all 200ms apart. Note that this is *not* the same value as the [coil's pulse ms](#). The coil's pulse ms is how many milliseconds that coil is pulsed for. This *repeat_pulse_ms* is how many milliseconds the machine waits between pulses. It's important that your *repeat_pulse_time* is significantly longer (perhaps 2-to-1) than your coil pulse time so that the coil has enough time to de-energize, allowing the reel to ratchet into its next position.

You can enter values here in seconds or milliseconds. See the full explanation of the time duration formats [here](#).

hw_confirm_time:

This is the time, in ms, that the machine will wait after a reel coil pulse before reading the switches to see what position the reel is in. The default is 300ms, though you might have to test your machine to dial-in the exact values. Many reels advance in two steps, with the initial coil stroke releasing a lever that advances the reel half-way, and then a ratchet spring that locks the reel in the rest of the way after the coil de-energizes. This means that if you have a coil pulse time of 100ms, you have to wait at least 100ms before the reel starts ratcheting into its new position. Then you have 20-30ms for that movement to occur, followed by another 100ms or so of "jitter" on the switch as it bounces around before locking into its final position.

Our experimentation with mid-70s Gottlieb Decagon reels yielded the best performance with coil pulse times of around 100ms, but in most cases the position switches didn't settle down until 240-260ms after that initial pulse. So in our machine we set this *hw_config_ms* to 300ms just to be safe. (You definitely want to err on the side of caution, since this is the delay before the machine reads the switches after a reel is fired. If that switch is still jittering and your machine just happens to read it as its "open" when in fact the reel has moved, the machine will think the reel didn't advance correctly and will try to correct it.)

You can enter values here in seconds or milliseconds. See the full explanation of the time duration formats [here](#).

confirm:

This option lets you specify how you'd like the machine to confirm that the score reels are in their proper positions. This is necessary because most individual reels do not have switches in every position, so it's possible that a reel advance could misfire and the reel could be physically out of sync with where the machine thinks it should be. This option lets you control what happens in those cases.

There are three options you can choose from:

- **lazy** - The default setting of "lazy" means that the machine does not check the value of the reel as it's advancing them, but after the reel group stops advancing, the machine will check to make sure the hardware switches make sense with where it thinks the reel should be, and it will reset them if they're not right. It's important to understand that the machine doesn't always know every position each reel is in. For example, if your score reels only have switches in the 9 and 0 positions and your player's score is 782, then all the machine can do is make sure that each reel is *not* in the 9 or 0 position. But really it doesn't know if the score reel group is showing 782 or 682 or 222—all it knows is that a 9 or 0 is either showing or not. However, if the machine thinks the score reel group should be displaying 782, but it sees that the ones digit reel has the 0 position switch active, then it will advance the ones reel twice to move it to what it assumes is the 2 position. The same will happen if the machine wants the score to be 780 but it does not see an active 0 switch for the ones. In that case it will advance the ones reel until the zero switch is active.
- **strict** - In "strict" mode the machine will check to make sure the switches are in the proper positions after *every single pulse*. The upside to this is that your score reels have a better chance of always being right, but the downside is that you slow down the speed your reels operate at. (In strict mode, the fastest a reel can be advanced is based on the value of the reel's *hw_confirm_ms* setting which might be something like 300ms, whereas in *lazy* or *none* mode it can advance the reels at their *repeat_pulse_ms* time which might be around 200ms.
- **none** - In "none" mode, the machine will never check the status of the reels (after the initial positioning to find an active switch when the machine is reset). At first you might think this is a horrible idea, but really it's how classic EMs have worked all these years. (When "none" is used, the machine will still use each reel's switches when it resets them to a new position, such as when the game restarts.) Also keep in mind that even if "none" is used, the machine still knows what each player's actual correct score is—regardless of what's displayed on the reels.

Device Control Events

None.

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

score_reel_groups

The *score_reel_groups*: section in the machine configuration files holds settings for groupings of mechanical score reels for EM-style pinball machines. Details about how score reel groups work can be found in the [Score Reel Groups device section](#) of this documentation.



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:

```
score_reel_groups:
  player1:
    reels: score_1p_10k, score_1p_1k, score_1p_100, score_1p_10,
None
    tags: player1
    max_simultaneous_coils: 2
    chimes: None, chime1, chime2, chime3, None
    confirm: lazy
  player2:
    reels: score_2p_10k, score_2p_1k, score_2p_100, score_2p_10,
None
    tags: player2
    max_simultaneous_coils: 2
    chimes: None, chime1, chime2, chime3, None
    confirm: lazy
```

Each section is headed with the name you'd like to refer to this score reel group as in your game code. In the above example, the score reel groups are named *player1* and *player2*, which obviously refer to the Player 1 and Player 2 score reel groupings in the machine's backbox. There are several subsections for each `score_reel_group`:

reels:

This is a comma-separated list of individual [score reels \(based on the names you gave them\)](#) that are grouped together to show the entire value displayed in this group. The number of items in this list equals the number of digits in your score, and the position of each individual reel represents its position in the group, starting with the left-most digit. Note that if your score reel group has one of those fake plastic inserts a given position, then you need to enter the word `None` in this list.

The configuration sample above is for a 5-digit score reel group with a fake 0 in the ones column, so that's why there are four reels defined in the ten thousands, thousands, hundreds, and tens position, with *None* as the placeholder in the ones position.

max_simultaneous_coils:

This is the maximum number of coils you want to fire at the same time in the score reel group. This is needed because most machines don't have enough power to fire too many coils at once. This usually comes into play when the score reel group is resetting itself to zero. The default value is 2 which should work fine in most situations and is a fairly accurate representation of how classic EM games operated. You can experiment with different values to see how well your specific machine can handle them.

chimes:

This is a list of coils that are pulsed each time a score reel advances when its performing a scoring action. This list should have the same number of items as your `reels:` configuration, and the position of each chime coil corresponds to the score reel group coil that will cause it to fire. In the example above, the line `chimes: None, chime1, chime2, chime3, None` means there is no chime for the ten thousands reel, *chime1* is used for the thousands, *chime2* is used for the hundreds, *chime3* is used for the tens, and no chime is used for the ones.

Note the actual chime coil pulse times are controlled by each coil's setting in the [Coils section of the machine configuration file](#).

([See our note](#) for how you can enter lists into the MPF config files for proper formatting.)

Device Control Events

None

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

Tags for score reel groups work like tags for other devices. You can use them to group and address score reel groups together. Certain tags also have special meanings.

- **playerX**- Used to tell the machine that you want to map this score reel group to the player "X" in the tag name. You can add multiple player tags to a single score reel group and then the game will use that group for both players. For example, if you have a machine with two score reel groups but you want to have 4 player games, you can give one group the tags `player1`, `player3` and the other `player2`, `player4`, and the game will automatically reset each reel group to each player's score when it becomes their turn.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

scoring

The `scoring:` section of the configuration files lets you increase (or decrease) the value of a [tracked player variable](#) when some event takes place. In most cases, you'll use the scoring section to add a value to the player's score variable, but technically this section can be used to add to any player variable. (So if you're tracking the total number of loops a player gets via a player variable called `loops`, you would use the scoring section to increase the value of the `loops` player variable by one whenever a loop is made.) See the [documentation on the Score Controller](#) for more details on this.



This section *cannot* be used in your machine-wide config files.



This section *can* be used in mode-specific config files.

Here's an example scoring section:

```
scoring:
  inlane_hit:
    score: 1070
  outlane_hit:
    score: 930
  sw_playfield_active:
    score: 10
    total_switches: 1
  right_ramp_made:
    score: 1000000|block
  ramps: 1
```

Inside your scoring section, there are basically only two things you have to configure:

<event_name>:

In your `scoring:` section, create an sub-entry for each event that you'd like to use to trigger a player variable to change value.

<player_variable>:

Then under each `event_name:` entry, create a sub-entry for one or more player variables and add the numeric value you'd like added to (or subtracted from) that player variable. For example:

```
scoring:
  outlane_hit:
    score: 930
```

In the example above, whenever the event `outline_hit` is posted, the player variable called `score` will be increased by 930.

```
scoring:
  sw_playfield_active:
    score: 10
    total_switches: 1
```

In the example above, whenever the event `sw_playfield_active` is posted, the player variable called `score` will be increased by 10, and the player variable `total_switches` will be increased by 1.

Note that you can have as many player variables listed for an event as you want, and you can also enter negative values if you want that event to subtract from the associated player variable.

Blocking score events in lower-priority modes

The `scoring:` section of your config file is a mode-based configuration setting. This means that you'll likely have `scoring:` sections in most (if not all) of your modes, and each mode's `scoring:` section will contain the scoring-related configuration for that mode only. (It also means that the scoring events you enter are only processed if the mode whose config they're in is running.)

There are situations where you might want a scoring event from one mode to block a scoring entry from a lower-priority mode. For example, you might have scoring for your pop bumpers set to 75,000 in your base mode. But after 50 pop bumper hits, you might want to enable a super jets mode where each shot to a pop bumper is worth 1,000,000 points instead. If you just entered `score: 1000000` in your super jets mode config, then each pop bumper hit would actually be worth 1,075,000 points since the scoring entry would be processed by both your base and your super jets modes. To address this, you can add `|block` to the end of a scoring entry. Doing so will prevent that event from affecting that player variable in a lower priority mode.

For example, if you have this config in your base mode:

```
scoring:
  pop_bumper_hit:
    score: 75000
    total_pops: 1
```

And then this config in your super jets mode:

```
scoring:
  pop_bumper_hit:
    score: 1000000|block
```

Now when your super jets mode is active, the super jets scoring of 1000000 will be added and it will prevent the `score: 75000` from the base mode from being added. Note that this `block` configuration is player variable-specific, so in the above configuration example, the base mode's `total_pops` value will still be increased by one when the super jets mode is active since the super jets `scoring` config was only set to block the `score`, not the `total_pops`.

Mode-based scoring

Whenever a tracked player variable changes (which is what player variables in the `scoring` section of your config create), an event is posted with the details of the change. You can use

this to show slides based on the new score or the score change. See the [score controller documentation](#) for details.

Also since all scoring is done in mode-based configurations, the score controller also tracks the total change of each player variable in a mode and posts mode-specific scoring events. (This means you can set a mode that, for example, ends when you get a certain number of points in that mode. Or you can base bonus scores on specific score values achieved in modes.) Again see the [score controller documentation](#) for details.

scriptlets

The `scriptlets:` section of your machine configuration files is where you specify the various scriptlets you've installed into your game. (What's a scriptlet? [Read here to find out.](#))



This section can be used in your machine-wide config files.



This section can be used in your mode-specific config files.

```
scriptlets:
  attract.Attract
  bonus.Bonus
  modes.Multiball
  modes.InstantDeath
```

You can add as many scriptlet as you want, with one per line. MPF will look for scriptlet in the `/scriptlets` folder in your machine files location. The part of the name before the dot is the module name (i.e. the file name), and the part after the dot is the class name in the scriptlet file. (Note you can have more than one scriptlet in the same file, with each one being its own class. Just add them to the list multiple times, one for each class.)

For example, the first entry of `attract.Attract` will load the `Attract()` class from the `/scriptlets/attract.py` file.

sequences

Sequences are a type of Logic Block where you can trigger a new event based on a series of one or more other events. They are a key part of implementing [game logic](#) in MPF.

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.

```
logic_blocks:
  sequences:
    light_special:
```

```

events:
  - shot_target
  - balldevice_right_popper_ball_enter
  - sw_leftStandup, sw_rightStandup
events_when_complete: start_multiball
enable_events: ball_started
disable_events: ball_ended
reset_events: ball_ended

```

Some settings for Sequence Logic Blocks apply to all logic blocks, so you can [read about them there](#), including:

- The “name” of the counter (e.g. “tilt” or “super_jets” in the examples above)
- enable_events
- disable_events
- reset_events
- events_when_complete
- restart_on_complete
- disable_on_complete
- reset_each_ball

These other settings are specific to the Sequence logic blocks:

events:

This is where you configure the actual events that make up the "steps" of your Sequence Logic Block. The real power of Sequence Logic Blocks is that you can enter more than one events for each step, and *only one* of the of the events of that step has to happen for that step to be complete.

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.)

The steps of a Sequence Logic Block must be completed in order. (If you want a Logic Block where the steps can be completed in any order, that's what the Accrual Logic Blocks are for.)

Read [our note about how to enter lists of lists](#) into the config files to make sure you get the configuration right.

In the example above, the `shot_target` event has to fire first, then a ball has to enter the right popper, then the player has to hit either the right or left standup

player_variable:

This lets you specify the name of the player variable that will hold the progress for this logic block. If you don't specify a name, the player variable used will be called `<sequence_name>_status`.

shots

The *shots*: section of your config files is where you setup the [shots](#) in your machine. A *shot* is a switch (or a series of switches that have to be hit in order), typically with a light or LED that reflects the current state of the shot (unlit, lit, complete, flashing, etc.). Shots can be things like standup targets, rollover lanes, drop targets, ramps, loops, orbits, etc.



This section can be used in your machine-wide config files.



This section can be used in mode-specific config files.

Shots can have different *shot profiles* applied to them. A shot profile controls what happens when a shot is hit. For example the 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 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.

You can add shots to your machine-wide config file or to a mode-specific config file. Shots in your machine-wide config will always be available, and shots in mode-specific configs will be created and then removed as the mode is active or inactive. (Shots can be enabled and disabled by game logic, so just because a shot exists in your machine-wide config doesn't necessarily mean that it's always enabled.)

You can also choose to create the shots in your machine-wide config, but then to add entries for them in mode configs to modify their behavior while that mode is active. (Per-mode shot profiles and control events, for example.)

Here's a sample *shots*: section from a config file:

```
shots:
  lane_1:
    switch: lane_1
    light: lane_1
  lane_a:
    switch: lane_a
```

```

    light: lane_a
lane_n:
    switch: lane_n
    light: lane_n
lane_e:
    switch: lane_e
    light: lane_e
upper_standup
    switch: upper_standup
    led: led_17, led_19
right_ramp:
    switch_sequence: right_ramp_enter, right_ramp_made
    time: 2s
left_orbit:
    switch_sequence: left_rollover, top_right_opto
    time: 3s
weak_right_orbit:
    switch_sequence: top_right-opto, top_center_rollover
    time: 3s
full_right_orbit:
    switch: top_right_opto, left_rollover
    time: 3s

```

Let's look at each of these configuration options in detail.

<name>:

Create one entry in your `shots:` section for each shot in your machine. Don't worrying 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.

switch:

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 setup 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.

switch_sequence:

A *switch_sequence* is where you configure your shot so that multiple switches have to be hit, in order, for the shot to be registered as hit. You can optionally specify a time limit for these switches (i.e. the sequence must be completed within the time limit) with the `time:` setting.

When the first switch in a sequence is activated, the 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".

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 hit, 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.

time:

This is the time limit these switches have to be activated in, from start to finish, in order for the shot to be posted. 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.)

cancel_switch:

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_switch:

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.

If you use this with a single switch shot, then the time must pass before the shot will count. If you use this with a `switch_sequence`, then the time must pass before a new sequence will start to be tracked.

Enter this switch with a time value (in seconds or ms), like this:

```
shots:
  mode_start:
    switch: mode_start
    delay_switch:
      rear_entry: 1.5s
  rear_entry_mode_start:
    switch_sequence: rear_entry, mode_start
    time: 1.5s
```

The example above illustrates a typical use for this where you have a single switch which you can hit from the front, and then also a rear entry where a rear switch is hit then the main switch. Setting up the switch sequence for the rear entry is easy, but without the `delay_switch` on the front entry, then a ball going in the rear entry would trigger a hit event for the front shot too.

light:

The name of the lamp matrix-based light associated with this shot. This can be a list of multiple lights if you want to have multiple lights tied to this shot. (Note: You'll likely only use `light:` or `led:`, but not both, depending on whether your machine uses a lamp matrix or direct controlled LEDs.)

led:

The name of the LED associated with this shot. This can be a list of multiple LEDs if you want to have multiple LEDs tied to this shot.

profile:

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.

Only one profile can be active at a time. (Though if you're really crazy and you need to track multiple profiles, you can configure the same switch and same light to be part of two different shots, and each one of them can have its own shot profile.)

Device Control Events

Device control events are events you can use to control devices. They are configured in your machine-wide or mode config with settings that end in *_events*. For example, if a device has a setting for *enable_events*: and you add an event to that setting, then when that event is posted, the device will enable. You can add single events or lists of events to these settings, and you can also configure time-delays for how much time passes between the event being posted and the action to take place. Details are available in the [device control event documentation](#).

Shots make use of the following device control events:

enable_events:

A list of one or more events that will 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.)

Default is *None*.

disable_events:

A list of one or more events that will 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.)

Default is *None*.

reset_events:

A list of one or more events that will reset this shot. Resetting a shot means that it jumps back to the first state in whatever [shot profile](#) is active at that time.

Default is *None*.

hit_events:

A list of one or more events that will 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.

Default is *None*.

advance_events:

A list of one or more events that will 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).

Default is *None*.

remove_active_profile_events:

A list of one or more events that will cause the active shot profile to be removed, and the next-highest priority profile to be applied.

Default is *None*.

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

shot_groups

Once you define all your [shots](#), you can group them together via the *shot_groups:* section of your config file.



This section can be used in your machine-wide config files.



This section can be used in mode-specific config files.

For example:

```
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>`). 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.

<name>:

Create one entry in your *shot_groups:* section for each group of shots in your machine. This name can be whatever you want, and it will be the name for this shot group which is used throughout your machine.

shots:

The list of shots (from the *shots:* section of your config file) that make up this shot group. Note that 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.

profile:

The name of the [shot profile](#) that will be applied to all the shots in this shot group.

- If you're editing a **machine-wide config file**, then the profile name specified here will be the default profile for each shot in the group 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 to each shot in this group 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.

Device Control Events

Device control events are events you can use to control devices. They are configured in your machine-wide or mode config with settings that end in `_events`. For example, if a device has a setting for `enable_events`: and you add an event to that setting, then when that event is posted, the device will enable. You can add single events or lists of events to these settings, and you can also configure time-delays for how much time passes between the event being posted and the action to take place. Details are available in the [device control event documentation](#).

Shot groups make use of the following device control events:

rotate_left_events:

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:

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.

rotate_events:

This list of events that, when posted, will rotate the current shot states in the direction specified in the profile's `custom_rotation`: setting. This can be a simple list of events or a [time-delayed list](#).

This lets you implement custom left-right patterns (like four left, then four right) like the moving lit shot skillshot on the bank of five targets in T2.

enable_rotation_events:

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.)

disable_rotation_events:

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:

A list of one or more events that will enable this shot group. (Enabling a shot group will also enable all of the individual shots that make up this group.) This can be a simple list of events or a [time-delayed list](#).

If a shot group is not enabled, then it will not post hit events and shot rotation is disabled.

If you do not specify any *enable_events*, then MPF will automatically create enable events based on the list in the `config_validator: shot_groups: enable_events:` section of your machine-wide config. (By default that's *ball_started*, meaning your shot groups are automatically enabled when a ball starts.)

If you specify any *enable_events* in your machine-wide config, then none of the default enable events will be added. (i.e. if you also want to include the default *enable_events*, you will have to add them here too.)

If you specify any *enable_events* in a mode-specific config, then those events are only active during that mode. Mode-specific *enable_events* are in addition to machine-wide *enable_events*.

disable_events:

A list of one or more events that will disable 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*.)

If you specify any *disable_events* in your machine-wide config, then none of the default *disable_events* will be added. (i.e. if you also want to include the default *disable_events*, you will have to add them here too.)

If you specify any *disable_events* in a mode-specific config, then those events are only active during that mode. Mode-specific *disable_events* are in addition to machine-wide *disable_events*.

reset_events:

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.

advance_events:

A list of one or more events that will advance all the shots in this shot group one step in the active profile. This can be a simple list of events or a [time-delayed list](#).

Advancing a shot does not post hit events and therefore does 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.

remove_active_profile_events:

A list of one or more events that will cause the active shot profile to be removed from every shot in the group, and the next-highest priority profile to be applied. This can be a simple list of events or a [time-delayed list](#).

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

A list of one or more tags that apply to this device. Tags allow you to access groups of devices by tag name.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

shot_profiles

You can use the `shot_profiles:` section of your config files to configure the settings for various *shot profiles* that you can then apply to your shots and drop targets.



This section can be used in your machine-wide config files.



This section can be used in your mode-specific config files.

```
shot_profiles:
  hit_me:
    states:
      - name: active
        light_script: flash
        repeat: yes
        tocks_per_sec: 5
      - name: complete
        light_script: "off"
    player_variable: laser
  default:
    states:
      - name: unlit
        light_script: "off"
      - name: lit
        light_script: "on"
  drop_shot:
    states:
      - name: up
        light_script: "off"
      - name: down
        light_script: "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 three profiles, called *hit_me*, *default*, and *drop_shot*. (All three of these profiles are contained in the system-wide `mpfconfig.yaml` file, so you don't actually have to create these exact ones in your game.)

player_variable:

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.

loop:

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.

state_names_to_rotate:

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:

```
shot_profiles:
  mode_tracker:
    rotation_events: pop_bumper_active
    state_names_to_rotate: unlit, active
    states:
      - name: unlit
        light_script: unlit
      - name: lit
        light_script: flash
        tocks_per_second: 5
      - name: complete
        light_script: "on"
```


state_names_to_not_rotate:

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.

rotation_pattern:

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:

```
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.

advance_on_hit:

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.)

lights_when_disabled:

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.)

block:

True or false value which 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.

states:

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. See our reference about [how to create list items in config files.](#))

The following sections explain the settings for each state:

(state) name:

This is the name of the step. In other words, it's what "state" the shot is in when this profile step is active.

(state) light_script:

This is the name of the light script that will be run on any lights or LEDs that are configured for the shot or drop shot that has this profile applied when the shot is hit and advances to this step. This light script should be a [script that you've created](#) somewhere else in your config files.

(state) hold:

Whether the light or LED should "hold" its final state if your light script ends. Default is true.

(state) reset:

Whether the light or LED should reset itself if your light script ends. Default is false.

(state) repeat:

Whether the light script should repeat. Control the number of times it repeats with the num_repeats: setting. Default is true.

(state) blend:

Whether the light script should blend with whatever else that light is doing at a lower priority. Default is false.

(state) tocks_per_sec:

How fast in terms of "tocks per second" the light script should run. Default is 10.

(state) num_repeats:

How many times the light script should repeat. A value of 0 means it loops forever. Note that repeat must be true in order for this to have an effect.

(state) sync_ms:

A sync_ms value you can use to synchronize this light script with other scripts that might be running. (This is how you can ensure that all the lights in your machine are flashing together.) See the documentation on light shows for details about how this works.

show_player

The `show_player:` section of your configuration file is used to automatically play and stop shows based on different [MPF events](#). (Remember that Shows are YAML files that contain synchronized sequences of lights, display slides, sounds, events, and coil actions. More information is available in the [Shows](#) section of this documentation.)



This section can be used in your machine-wide config files.



This section can be

used in your mode-specific config files.

Here's an example

```
show_player:
  attract_start:
    - show: random_blinking
      repeat: yes
      tocks_per_sec: 4
    - show: attract_display
      repeat: yes
      tocks_per_sec: 1

  attract_stop:
    - show: random_blinking
      action: stop
    - show: attract_display
      action: stop

  shot_jackpot:
    show: jackpot
    tocks_per_sec: 30
```

To configure shows to automatically play and stop, you create a `show_player:` entry in your config file. Then create a sub-entry for each MPF event where you'd like (one or more) shows to play or stop.

If you want to enter more than one show for each event, you have to [enter them as a list](#). The easiest way to do that is to add a dash ("-") and a space before each show entry while keeping them at the same level of indentation. (See the `attract_start:` section in the example. Notice there are two sub-entries, one which starts with `- show: random_blinking`, and one which starts with `- show: attract_display`.)

Notice that you can enter any settings for shows you want, including `repeat`, `priority`, `blend`, `hold`, `tocks_per_sec`, `start_location`, and `num_repeats`. See the documentation on [playing shows](#) for details.

If you want to an event to stop a show, add an setting `action: stop`, as shown in the example above in the `attract_stop:` entry. Stopping a show can also take settings `reset` and `hold`.

If you add a `ShowPlayer:` section to a mode-specific config files, shows that are playing as part of that mode are automatically stopped when the mode ends.

shows

You use the `shows:` section of your configuration file to specify additional non-default settings for any [show](#) you want to use in your game. (This section can be in your machine configuration files for machine-wide assets and/or a mode-specific config file for mode-specific assets.)

Note: You do *not* have to have an entry for every single show you want to use, rather, you only need to add individual assets to your config file that have settings which differ from other assets in that asset's folder. (See the [AssetDefaults:](#) section for details. Also be sure to read the section on [Managing Assets](#) for an overview of how assets work.)

Here's a example:

```
shows:
  extra_ball:
    load: preload
```

Since shows are just a type of Asset in MPF, there are a few config settings that are generic which apply to all types of assets, so read the [reference documentation on assets](#) for details about how the `name`, `file:`, and `load:` settings work.

There are no additional settings for shows apart from the generic settings that apply to all types of assets.

slide_player

The `slide_player:` section of your config file lets you configure various [display elements](#) and [slides](#) that will automatically play based on certain MPF events. This section is very similar to the `sound_player:` and `show_player:` sections, except this one is for displays.



This section can be used in your machine-wide config files.



This section can be used in your mode-specific config files.

Here's an example:

```
slide_player:
  player_score:
    - type: text
      text: "%score%"
      number_grouping: true
    - type: text
```

```

    text: "%player|number%"
    v_pos: bottom
    h_pos: left
    size: 5
  - type: text
    text: %player|ball%"
    v_pos: bottom
    h_pos: right
    size: 5
  tilt_hit:
    type: text
    text: WARNING
  player_add_success:
    type: text
    text: PLAYER %num% ADDED
    expire: 2s
  ball_live_added:
    type: text
    text: LET'S GO!
    expire: 2s
  balldevice_deadworld_ball_enter:
    type: text
    text: "BALLS LOCKED: %balls%"

```

First you create an entry called `slide_player:`. Then you create subsections in there for MPF events, followed by one or more entries for specific [display elements](#) (and their settings) that you'd like to be displayed when that event is posted.

Take a look at the following section for a simple example:

```

  tilt_hit:
    type: text
    text: WARNING

```

What this does is when the event `tilt_hit` is posted, a display element of type "Text" will be put on the display, and the text will say "WARNING". Since no other settings are defined, the defaults (also configurable in your configuration files) will be used. There are several options for each display element, including:

type:

The `type:` can be any of the display element options [covered here](#). The current options are *text*, *shape*, *image*, *animation*, *movie*, *character_picker*, and *entered_chars*. Refer to the link for details about the options you can use for each different type of element. Note that *text* display elements can also include percent signs to use dynamically-generated text. (See the section below on dynamically-generated text for details.)

expire:

An [MPF time string](#) (such as `2s` or `500ms`) that causes this slide to automatically remove itself. Whatever next-highest priority slide is will be displayed in its place. Note that if this slide enters via a transition, the timer will start after the transition is complete.

slide_name:

A string name that you'd like to call the slide that is created. If a slide already exists with this name and `clear_slide:` is set to `False`, then these elements will be added to the existing slide. If you don't specify a `slide_name`, then a new slide will be created with a random name.

slide_priority:

The priority this slide will have. This has to be the same or greater than whatever slide is currently showing or else this slide won't show up. (It will still be created and will be shown later if the current higher priority slide is removed.)

If this `ShowPlayer:` section is being configured as part of a mode config, the `slide_priority` entered here will be added to the mode's base priority. (e.g. If the mode is running at priority 300 and you specify `slide_priority: 1`, the slide created will have a priority of 301.)

clear_slide:

True/False as to whether the slide should be cleared (erased) before these elements are added to it. This only applies when you specify a slide name and when a slide already exists with that name.

clear_slides:

Clears all the slides created by the mode where this `slide_player:` entry is.

By default, MPF keeps at least one slide per mode unless the slides were configured to expire or the mode ends. But what if you want to clear all the slides from a mode but you don't want to end that mode? That's where the `clear_slides` option comes in.

To use it, just add `clear_slides: yes` to an event entry in `slide_player:` section of a mode config file:

```
slide_player
  some_event:
    clear_slides: yes
```

In the example above, when the event *some_event* is posted, all of the slides that mode created will be removed.

persist_slide:

True/False as to whether this should persist after it's no longer being shown. If True, then you can show this slide again by calling it by name.

display:

String name of the display that you'd like this slide to be shown on. (Such as *DMD* or *window*.) If you don't specify a display then this slide will be built for the default display.

transition:

If you want to apply a transition into this slide, you can add a `transition:` entry with sub-settings that control the transition itself.

Specifying multiple display elements for one event

If you want to add two elements to the display, note that you can enter multiple items as long as they are all indented the same amount and you add a dash ("-") to signify when the next item starts. For example:

```
ball_started:
- type: Text
  text: "%score%"
  min_digits: 2
  number_grouping: true
- type: Text
  text: PLAYER %number%
  v_pos: bottom
  h_pos: left
  size: 5
- type: Text
  text: BALL %ball%
  v_pos: bottom
  h_pos: right
  size: 5
  transition:
    type: move_in
    direction: top
```

This entry puts three separate display elements on the display when the `ball_started` event is posted.

- A Text element with text that shows the current score, using the default size and position. Note that "%score%" is wrapped in quotation marks since its value starts

with %, and that confuses the YAML parser. So we wrap it in quotes to tell the parser that the value is a string.

- A Text element with text that says "PLAYER %number%" in the bottom left position, rendered at font size 5.
- A Text element with text that says "BALL %ball%" in the bottom right position, rendered at font size 5.

On-the-fly text replacement with % signs

The *slide_player* entries in your config file are triggered by events. Many events have keyword/value parameters passed along with event itself when it was posted, and you can access the values of those parameters with the text display elements in your *slide_player*.

To access event parameters in a text display element, just add the parameter name surrounded with percent signs, like this:

```
slide_player:
  player_add_success:
    type: text
    text: PLAYER %num% ADDED
```

Since the *player_add_success* event includes a parameter *num* which is the player number, this will print *PLAYER 1 ADDED* (or whatever the number of the new player is) when the *player_add_success* event is posted.

You can include % variables in-line with other text or by themselves on their own lines. Note though that if you include them on their own lines, a YAML value cannot start with a percent sign, so you need to put it in quotes, like this:

```
type: "%num%"
```

You can also include the values of player variables and machine variables in your *slide_player* entries. See the documentation on the [text display element](#) for more details.

smartmatrix

The `smartmatrix:` section of your machine-wide config contains the settings for a SmartMatrix RGB LED DMD controller. (See our [How To guide on these](#) for details of how to set one up.)



This section can be used in your machine-wide config files.



This section *cannot* be used in mode-specific config files.

```
smartmatrix:
  port: com12
  use_separate_thread: yes
```

port:

This is the name of the serial port that your Teensy uses. If you don't know it, just look at your serial ports, then plug in your Teensy and see which port appears. On a Mac or Linux, this will have a Unix-style port name.

There is no default setting.

use_separate_thread:

This controls whether the code that communicates with the Teensy will run as part of your machine MPF loop or whether it will run in a separate thread. Options are yes/no or true/false.

The correct setting will depend on the specifics of your hardware and what size display you're running. For example, on one test system (MPF running in a Windows VM on a MacBook), it didn't matter what this was set to—the experience was the same either way. Another user running a larger size 128×64 display, had to set `use_separate_thread: no` to get good performance. So basically try it with both settings and see which one works better.

Default is *true*.

snux

The *snux* section of your machine config file lets you configure settings for the Snux driver board for System 11 machines. (What's the Snux driver board? See our [How To guide for System 11 machines](#) for details.)



This section can be used in your machine-wide config files.



This section *cannot* be used in mode-specific config files.

This *snux* configuration is included in the *mpfconfig.yaml* template file and shouldn't have to be included in your own machine's config.

```
snux:
  flipper_enable_driver_number: c23
  diag_led_driver_number: c24
```

flipper_enable_driver_number:

The driver output that's mapped to the flipper enable relay. (Note that the actual devices the flipper enable relay enables varies depending on the particular System 11 machine it's connected to.)

diag_led_driver_number:

The driver output that's mapped to the "diag" LED on the Snux driverboard.

sound_player

You can use the `sound_player:` section of your config file to configure sounds to automatically play based on certain [MPF events](#).



This section can be used in your machine-wide config files.



This section can be used in your mode-specific config files.

Here's an example:

```

sound_player:
  intro loop:
    sound: intro_loop
    start_events: ball_starting
    stop_events: ball_live_added
    duration:
    loops: -1
    priority:
    fade_in: 0
    fade_out: 0
    volume: 1
  this level entry can be whatever you want:
    sound: ball_launch_motorcycle
    start_events: balldevice_shooterLaneR_ball_eject_attempt
  slingshot:
    sound: crime 12
    start_events: shot_Slingshot
  anything you want:
    sound: main_loop
    start_events: player_eject_request
    loops: -1
    stop_events: ball_ending

```

To use the Sound Player, add a `sound_player:` section to your config file. Then create sub-entries from each sound/event combination you want. Note that the actual name of each sub-entry doesn't really matter. What matters is the sound and events in the entry.

So for each sub entry, there are two required settings:

sound: (required)

The name of the sound object that will play. Note that this is *not* a file name, rather, it's the name of the sound in MPF.

start_events: (required)

A list of one (or more) MPF events that will cause this sound to play. You can separate them with commas, or enter one event on each line if you start each line with a dash. More details on how to enter lists into config files is [here](#).

stop_events:

A list of events that will cause this sound to stop playing.

duration:

Not yet implemented.

loops:

The number of times this sound will loop. (So 0 means it plays once, 1 means it plays twice because it loops once, etc.) Enter a value of -1 to have it just loop forever until one of the `stop_events` is posted.

Default is 0 which means it just plays once and stops.

priority:

The relative priority this sound will be played at. For `sound_player: settings` in mode config files, this priority will be added to the mode's base priority.

The priority is only used if the track the sound is playing on is currently playing its max number of simultaneous sounds. If so, the new sound will replace one of the existing sounds if the the new sound has a higher priority than the lowest priority current sound. Otherwise the new sound will be added to the queue and played when the next available slot is free.

By the way, if you find that you have too many sounds cutting off other sounds, then you can increase the `max_simultaneous_sounds` value for that track.

fade_in:

Not yet implemented.

fade_out:

Not yet implemented.

volume:

A value of how loud this sound should play, between 0 and 2. The volume you configure here is mixed in with the track and overall MPF volume.

A value of 1 (the default) plays the sound at the full normal volume. 0 is mute. 0.1 would be 10%, 1.5 would be 150%, etc. Max value is 2.

sounds

You use the `sounds:` section of your configuration file to specify additional non-default settings for any sounds you want to use in your game. (This section can be in your machine configuration files for machine-wide assets and/or a mode-specific config file for mode-specific assets.)



This section can be used in your machine-wide config files.



This section can be used in your mode-specific config files.

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 differ from other assets in that asset's folder. (See the [asset_defaults:](#) section for details. Also be sure to read the section on [Managing Assets](#) for an overview of how assets work.)

Here's an example:

```
sounds:
  jackpot:
    volume: 2
    track: voice
  ball_launch:
    volume: 1.1
```

Since Sounds are just a type of Asset in MPF, there are a few config settings that are generic which apply to all types of assets, so read the [reference documentation on Assets](#) for details about how the `name`, `file:`, and `load:` settings work.

Then on top of the defaults, sounds have several sound-specific settings: (These are all default settings for each sound. Many can be overridden when the sound is actually played.)

track:

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.) You can specify the name of a standard track or the name of your streaming track.

volume:

The volume offset of this sound, expressed at a number between 0 and 2. 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. A value of 1 (the default) means this sound plays at normal volume. 0.5 means it plays at 50%, 1.5 means it plays at 150%.

The max value here is 2. Anything higher than 2 will be reset to 2.

max_queue_time:

An [MPF time string](#) which 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 `max_simultaneous` 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.

max_simultaneous_playing:

Not yet implemented.

fade_in:

Not yet implemented.

fade_out:

Not yet implemented.

loops:

Not yet implemented.

start_time:

Not yet implemented.

end_time:

Not yet implemented.

sound_system

The `sound_system:` section of the machine config file controls the general settings for the machine's sound system.



This section can be used in your machine-wide config files.



This section can be used in your mode-specific config files.

Here's an example:

```

sound_system:
  buffer: 512
  frequency: 44100
  channels: 1
  initial_volume: 1
  volume_steps: 20
  tracks:
    voice:
      volume: 1
      priority: 2
      simultaneous_sounds: 1
      preload: no
    sfx:
      volume: 1
      priority: 1
      preload: no
      simultaneous_sounds: 3
  stream:
    name: music
    priority: 0

```

buffer:

This is the size of your sound buffer. It must be a power of 2. If you leave this setting out (or set it to ``auto``), then MPF will automatically pick a buffer size.

The exact value you use will take some trial-and-error. A bigger buffer means that there's less chance of skipping and dropout, but it also means that sounds can take longer to play since the buffer has to fill first. We use the value of 512 on the BeagleBone Black and it seems fine. Sounds are still instant as far as we can tell, and we don't have problems with skipping. Some people have 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.

bits:

This is the number of bits (8-bit for 16-bit) for your sound. You need to enter the number as negative for signed values. Probably you should never use this, which is why it's not included in the example config file above. Just leave it out.

frequency:

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 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.)

channels:

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.

initial_volume:

The initial MPF overall volume, represented as a number between 0 and 1. 1 is max volume, 0 is off, .9 is 90%, etc. Note that this only controls the volume of the MPF app, not the host

OS's system volume. So you still need to make sure that the host OS is not on mute and that the volume is turned up.

volume_steps:

This is the number of steps of volume you want when you present the volume controls to the operator or player. For example, a value of `20` means that the volume of a machine at full will show to the player as "20", and then as they turn it down it will step through, 19, 18, etc.

Note that this setting does not affect the internal volume. (That's still always a value between 0 and 1.) The `volume_steps:` setting only affects how it's displayed to the player.

tracks:

Every sound that's played in MPF is played on a track. Each track can have it's own settings, and you can set volume on a per-track basis. The example above shows two tracks, called *voice* and *sfx*. The idea (in case it isn't obvious) is that you play all your 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, `voice:` and `sfx:` in the example. Then create one or more of the following settings for each track:

volume:

This is the volume offset for this track, as a value between 0 and 1. A value of 1 (the default) means that this track plays at full volume. Note that each track's volume will be combined with the overall system volume. So if your MPF overall volume is set to .8 (80%) and you have a track set to .5 (50%), sounds on that track will play at 40% overall volume (50% of 80%).

priority:

The relative priority of this track. This feature is not really implemented yet, but it will be used in the future to control tracks automatically getting quieter (called "ducking" in the audio world) when higher priority tracks are playing. (For example, you might want the sound effects track to lower its volume when voice callouts are playing.)

simultaneous_sounds:

This sets the maximum number of simultaneous sounds that can be played on this track. The example config file above shows the *voice* track with a max of 1 simultaneous sound

playing, since if you have two 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.

preload:

Controls whether the sound files for this track should be preloaded into memory when MPF starts up. Preloading sounds means they're available to play instantly, but it also means that those sounds will always be consuming memory. Whether you preload sounds depends on how many sounds you have and how much memory your computer has.

This preload setting only sets the default for the track. You can choose to override it on a sound-by-sound basis. (So if you set preload to "False" (or "no") here, you can still choose to preload a few key sounds when you set up your sounds.

Note that if a sound is not preloaded, it can still play—it's just that MPF has to load it from the disk first. Again, whether that's bad or not depends on your computer. If you have an SSD you can probably get away without preloading any sounds and they'll all play fine.

stream:

This setting lets you configure a "stream" track which is a special kind of track which streams sound files from disk as they're being played instead of loading the complete sounds into memory. The catch is that you can only stream a single sound file at a time. (This is a limitation of SDL which is what MPF's sound system is based on.)

In practice, the stream track works great for background music since those files are long (and would therefore take up a lot of memory), and you only ever have one background sound playing at a time.

Specific settings for the stream track include:

name:

The name of the streaming track which you'll use later when you want to control which track each sound plays on.

priority:

The relative priority of this track. Again, this setting doesn't really do anything right now.

switches

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.



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:

```
switches:
  flipperLwR_EOS:
    number: SF1
  flipperLwR:
    number: SF6
  fireR:
    number: S12
    tags: plunger
  start:
    number: S13
    tags: start
  plumbBob:
    number: S14
    tags: tilt
  outlaneL:
    number: S16
    tags: playfield_active
    debounce: True
  inlaneL:
    number: S17
    tags: playfield_active
    debounce: True
  trough1:
    number: S81
    type: 'NC'
  shooter_lane:
    number: S82
    activation_events: ball_in
    deactivation_events: ball_out
```

Each subsection of `switches:` is a switch name, which is how you refer to the switch in your game code. Then there are several parameters for each switch:

number:

The number is the physical switch input number. The exact format if this will vary depending on what type of pinball controller you're using and how the switch is connected. You can refer the the how to guides for each hardware platform for details:

- [P-ROC switch numbering](#)
- [P3-ROC switch numbering](#)
- [FAST Pinball controller switch numbering](#)
- WPC switch numbering (keep reading below)

If you're using a Williams WPC machine (with either FAST or P-ROC, when your *machine: driverboards* configuration is set to *wpc*), you can use the switch numbers printed in the game's operators manual and the P-ROC or FAST platform interface will translate them into the actual numbers behind-the-scenes.

The numbering scheme is as follows:

- **SFx** - This is a switch connected to a Williams Fliptronics interface board, labeled *Fx* in the Williams Operators Manual. These types of switches would be entered into the config file like `number: SF1`.
- **SDx** - This is a dedicated switch, labeled *SDx* in the Williams manual. Use them like `number: SD21`.
- **Sx** - This is a matrix switch, just labeled with a switch number in the Williams manual. Use them like `number: S12`.

activation_events:

A list of one or more names of events that MPF will post when this switch goes active. Enter the list in the MPF [config list format](#). These events are posted exactly as they're entered, in addition to any events that are posted based on the switch's tags.

deactivation_events:

A list of one or more names of events that MPF will post when this switch goes inactive. Enter the list in the MPF [config list format](#).

type:

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.

Note: This `type:` entry will probably go away in the future and we'll just add a `tags:` option for NC, since really there's no need for a separate `type` parameter here.

debounce:

Lets you configure whether a switch should be debounced or not. Each hardware platform handles debounce differently, so check out the how to guides for your specific platform for details:

- [P-ROC switch debounce settings](#)
- [P3-ROC switch debounce settings](#)
- [FAST Pinball controller switch debounce settings](#)

Device Control Events

Device control events are events you can use to control devices. They are configured in your machine-wide or mode config with settings that end in `_events`. For example, if a device has a setting for `enable_events:` and you add an event to that setting, then when that event is posted, the device will enable. You can add single events or lists of events to these settings, and you can also configure time-delays for how much time passes between the event being posted and the action to take place. Details are available in the [device control event documentation](#).

Switches make use of the following device control events:

Settings that apply to all device types

There are some settings that apply to all types of devices that also apply here.

tags:

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 `DropTargetBank1` 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, including:

- **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 to validate the playfield as well as to reset the ball search timer. (i.e. as long as switches with the `playfield_active` tag are being activated, the ball search timer won't start.)
- **start** - Used to tell the game that the player has hit the start button.

label:

The plain-English name for this device that will show up in operator menus and trouble reports.

debug:

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*.

switch_player

You use the `switch_player:` section of your machine configuration file to enter a "script" of switch events that will automatically be played back to simulate your machine being used (for troubleshooting and testing purposes).



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 from our *Demo Man* game. This script puts a ball in the trough, hits start, moves the ball from the trough to the plunger lane, hit's launch, moves the ball out of the plunger lane, and hits a few playfield targets.

```
switch_player:
  start_event: machine_reset_phase_3
  start_delay: 1s
  steps:
    - time: 100ms
      switch: s_trough_1
      action: activate
    - time: 600ms
      switch: s_start
      action: hit
    - time: 100ms
      switch: s_trough_1
      action: deactivate
    - time: 1s
      switch: s_shooter_lane
```

```
    action: activate
  - time: 1s
    switch: s_ball_launch
    action: hit
  - time: 100ms
    switch: s_shooter_lane
    action: deactivate
  - time: 1s
    switch: s_right_inlane
    action: hit
```

There are a few configuration settings:

start_event:

This is the MPF event that will trigger this script to begin. If you don't add this section, it will automatically begin when the `machine_reset_phase_3` event is posted.

start_delay:

This is an MPF time string entry which specifies how soon the script kicks off after the `start_event` is posted.

steps:

This is a series of steps, each with three settings:

time:

An MPF time string entry of how long to wait before performing the action in this step.

switch:

The switch name (from the `Switches:` section of your config file) for the switch whose state you're setting.

action:

What action you want to be applied to the switch. There are three options:

- **activate** - Sets the switch to active. This is the "logical" setting of the switch which *does* take into consideration a switch's "NC" setting. (So for example, an opto switch going active technically means the switch hardware opens, but since it's an NC switch, it's inverted. "Active" means the game thinks there's a ball there.)

- **deactivate** - Sets the switch to inactive. (i.e. no ball activating it.)
- **hit** - Does a quick "hit" of the switch which activates and then immediately deactivates the switch, like when a ball rolls over a target or a player hit's the launch button.

Launching Switch Player scripts via the command line

Remember that in MPF you can have as many machine configuration files as you want, and you can list additional config files in other config files. This means if you have a game that you sometimes want to test with an automatic script and other times you don't, you can create a simple standalone config file for the switch test without having to duplicate all your settings.

For example, imagine you have your "main" configuration file in `config.yaml`. You launch it like this:

```
python mpf.py your_machine_folder
```

You don't need to specify any configuration folder option here because MPF automatically looks for a file called `config.yaml` at `your_machine_folder/config/config.yaml`.

Now what if you wanted to create an automatic switch script for this machine? In that case, create a second configuration file called `autotest.yaml` and set it up like this:

```
config:
  - config.yaml

switch_player:
  <...>
```

To run your game with this switch player script, use a command like this:

```
python mpf.py your_machine -c autotest
```

The `-c` option tells MPF that you want to use the configuration file `your_machine/config/autotest.yaml` instead of `your_machine/config/config.yaml`. If you look at that file from above, notice that it contains the settings for your Switch Player, it contains the link to the switch player plugin, and it contains a link to the main `config.yaml` which it merges in when you run it. So this is how you can have one main config file but then still have the option to use or not use your automated switch playback scripts.

system11

The *system11:* section of the machine-wide config file contains settings which control how MPF works with System 11 machines. See the [How To guide for System 11 machines](#) for details of this interface.



This section can be used in your machine-wide config files.



This section *cannot* be used in mode-specific config files.

```
system11:
  ac_relay_delay_ms: 75
  ac_relay_driver_number: c14
```

ac_relay_delay_ms:

The number of milliseconds MPF waits before and after flipping the A/C select relay to allow for it to fully switch positions. Default is *75ms*.

ac_relay_driver_number:

The driver (with a "C" added to it) from the operator's manual of your System 11 machine which controls the A/C select relay. Note that this relay is labeled differently in different manuals. Sometimes it's called the *A/C Select Relay*, and sometimes it's called the *Solenoid Select Relay*.

tilt

The *tilt:* section of your config lets you specify the settings that the tilt mode will use to process tilts.



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:

```
tilt:
  tilt_warning_switch_tag: tilt_warning
  tilt_switch_tag: tilt
  slam_tilt_switch_tag: slam_tilt
  warnings_to_tilt: 3
  reset_warnings_events: ball_ended
  multiple_hit_window: 300ms
  settle_time: 5s
  tilt_warnings_player_var: tilt_warnings
```

Let's look at each of these settings in more detail:

tilt_warning_switch_tag:

The tag of the switch (or switches) that will cause a tilt warning to occur. When a switch tagged with this tag is activated for the *warnings_to_tilt* value, a tilt will occur. The count that keeps track of these warnings is specified in the *tilt_warnings_player_var*.

tilt_switch_tag:

The tag of the switch (or switches) that will cause a tilt to occur. Switches with this tag do not use the tilt warnings, rather they cause a tilt to occur immediately. So don't use this for the plumb bob (use the *tilt_warning_switch_tag* for that). Instead this might be for an older machine that has the roller ball tilt that detects the front of the machine being lifted off the ground.

slam_tilt_switch_tag:

The tag of the switch (or switches) that, when activated, will cause a slam tilt to occur.

warnings_to_tilt:

The number of warnings that will generate a tilt. The tilt is the result of the last warning here, so a value of 3 here means the player will get 2 warnings and then tilt on the 3rd.

reset_warnings_events:

A event (or list of events, in [MPF device control event format](#)) that will reset the number of warnings for the player. For example, if you configure your machine for three warnings to tilt, if *reset_warnings_events* is *ball_ended* then that means the player will get two warnings per ball before the tilt. If you change it to *game_ended*, that means they'll get two warnings per game.

multiple_hit_window:

The amount of time (in [MPF time string format](#)) where multiple hits to the tilt warning switch will be counted as a single warning.

settle_time:

The amount of time (in [MPF time string format](#)) that has to pass before the next ball will be served after a tilt.

tilt_warnings_player_var:

The name of the player variable that will be used to track the number of tilt warnings.

timers

You can configure timers via the `timers:` section of a mode's config file.



This section *cannot* be used in your machine-wide config files.



This section *can* be used in your mode-specific config files.

A timer is kind of like a logic block and posts events every "tick" and counts up or down. You can configure how fast the timer ticks and whether it counts up or down, and you can also stop, start, pause, extend, reset, or change the speed of the timer.

Timers don't really do anything on their own, rather they just post events (at each tick and on completion) that you can use for other parts of your game logic, like starting or stopping modes, hurry up scores, display and sound updates, etc.

Here's an example of a timer that's used to stop a timed skillshot mode:

```
timers:
  mode_timer:
    start_value: 5
    end_value: 0
    direction: down
    tick_interval: 1.5s
    control_events:
      - event: sw_pop
        action: pause
        value: 2
      - event: balldevice_playfield_ball_enter
        action: start
```

<name>:

Each timer has a sub-entry in the `Timers:` section of the mode config file. You can name this timer whatever you want (and it's name it based on whatever the sub-entry is). The name is used to construct the various MPF events this timer posts. (The timer in the above example's name is "mode_timer.")

start_value:

This is the default value the timer starts at (and the value it goes to when it's reset). The default is 0. If the timer is configured to count down, then set this `start_value` to something other than 0.

start_running:

True/False (or yes/no) as to whether this timer should start running as soon as the mode starts. If no, then this timer won't start until one of the control events is posted to start it. Default is False.

end_value:

When the timer hits this value, it will post a `timer_<name>_complete` event. The default setting for the `end_value` is 0, so if you have a count down timer then it will post its complete event when it gets to 0. If you have a count up timer and no `end_value` specified, it will count up forever (or until you stop it or the mode ends).

direction:

Specifies whether this timer counts up or counts down. The default is to count up, but you can enter `direction: down` to create a timer that counts down.

tick_interval:

An [MPF time string](#) which specifies how long each "tick" of the timer is. The default is 1 second, but you can set this to whatever you want. This is an easy way to set timers to run in what we call "pinball time" where a countdown modes count much slower than real-world time. (For example, you might have a 30-second hurry up mode that starts a countdown timer on the display from 10 to zero, but really each "tick" is 3 seconds.)

control_events:

This is a list of event/action/value pairs that control the behavior of the timer. Looking at the example above, when the event `balldevice_playfield_ball_enter` event is posted, the timer will start. When the event `sw_pop` is posted, the timer will pause for 2 seconds.

Notice that you enter multiple control events in MPF's "list of lists" format with a minus sign and space denoting each new group of settings.

Settings for control events include:

event:

The name of the event that will cause this control action to take place.

action:

The name of the action type this event will cause. Valid options include:

- `add` - Adds time (specified in the "value" entry below) to the timer.
- `subtract` - Removes time (specified in the "value" entry below) to the timer.
- `jump` - Sets the timer to a specific time (specified in the "value" entry below).
- `start` - Starts the timer. Posts the MPF event `timer_<name>_started`. Use this to start a timer that was previously stopped or paused.
- `stop` - Stops the timer. Posts the MPF event `timer_<name>_stopped`.
- `pause` - Pauses the timer for the amount of time specified in the "value" entry below. Posts the MPF event `timer_<name>_paused`. If you don't specify the pause time, it pauses for an indefinite amount of time. (This is essentially the same as stopping the timer, with the only difference being the names of events that get posted.)
- `set_tick_interval` - Sets a new value for the tick interval.
- `change_tick_interval` - Adds or subtracts the new time value to the current tick interval.

value:

The optional value that goes along with the action above.

restart_on_complete:

If True, automatically restarts this timer when it hits the end.

timing

The `timing:` section specifies how many "ticks" per second the MPF game engine runs at. If you're using the MPF media controller, this setting will also control it.



This section can be used in your machine-wide config files.



This section *cannot* be used in mode-specific config files.

This setting is set to 30 (i.e. 30Hz) in the `mpconfig.yaml` file, so you only need to add it if you want your machine to run at a speed other than 30Hz. Setting it is simple:

```
timing:
  hz: 30
```

More details about the machine Hz are available in the [Timers & Machine Tick Speed documentation](#) in the MPF Core Architecture section.

hz:

This setting controls how many "loops" per second your game code runs at. (Another way to phrase this is it's how many times per second MPF "wakes up" to do things.) It also represents the fastest speed any periodic updates can occur in your machine, so if you want to run your display at 60fps then set your *hz* to 60.

Note that when switches are hit, they will "wake up" MPF immediately, so having a low hz setting does not mean that MPF will not be responsive. See the document about Machine HZ & Loop Rates for details.

The default value is 30.

hw_thread_sleep_ms:

This is how many milliseconds MPF will sleep between polling the hardware for switch changes. The default value is 1 which should be fine for most cases. If you set this to 0 then MPF will poll the hardware as fast as possible, but this is not advised since it means MPF would consume 100% of your CPU which means it will run hotter and your overall system will not be responsive.

triggers

The `triggers:` section of a config file is used to send BCP trigger commands based on MPF events.



This section can be used in your machine-wide config files.



This section can be used in your mode-specific config files.

The BCP specification dictates that triggers are sent in the format `trigger?name=foo`. They can optionally include additional parameters, such as `trigger?name=foo&some_param=whatever`.

To configure triggers, create a `triggers:` section in your config file. Then create a sub-section with a name that matches an MPF event, specify the name that you want to be passed via BCP as well as (optionally) any additional parameters. For example:

```
triggers:
  sw_plunger:
    bcp_name: plunged
    params:
      how_many: 1
      this_many: 2
  shot_ramp:
    bcp_name: ramp_made
```

In the above example, the BCP command `trigger?name=plunged&how_many=1&this_many=2` will be sent when the MPF event `sw_plunger` is posted and the BCP command `trigger?name=ramp_made` when the MPF event `shot_ramp` is posted. You can optionally use % signs like with text display elements of the [Slide Player](#) to substitute event parameter text with player variables and/or parameters attached to MPF events.

videos


You use the `videos:` section of your configuration file to specify additional non-default settings for any video you want to use in your game.



This section can be used in your machine-wide config files.



This section can be

used in mode-specific config files.  Note: The precompiled Mac binaries for Pygame do not include the movie module MPF needs to play videos. Yet another reason we need to move away from Pygame.

Note: A video in MPF is an MPEG-1 video that can be displayed in an on screen window. Typically this is either a background loop or a cut scene. Videos are similar to Animations in MPF, though animations tend to be for foreground elements and support alpha transparencies.

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 differ from other assets in that asset's folder. (See the [assetdefaults:](#) section for details. Also be sure to read the section on [Managing Assets](#) for an overview of how assets work.)

Here's a example:

```
videos:
  background_fire:
    load: preload
```

Since videos are just a type of Asset in MPF, there are a few config settings that are generic which apply to all types of assets, so read the [reference documentation on Assets](#) for details about how the `name`, `file:`, and `load:` settings work.

There are no additional settings for videos apart from the generic settings that apply to all types of assets.

virtual_platform_start_active_switches

The `virtual_platform_start_active_switches:` section lets you configure of switches that will automatically start as "active" when you launch MPF using the virtual hardware platform. (i.e. when you use the `-x` command line option). This is used to make testing your machine easier since you can configure the trough to be "full" without having to manually hit the keyboard keys to activate each trough switch before you can start a game.



This section can be used in your machine-wide config files.



This section *cannot* be

used in mode-specific config files.

The settings are pretty simple—just a list of switch names. For example:

```
virtual_platform_start_active_switches:
  s_trough1
  s_trough2
  s_trough3
  s_trough4
  s_trough5
```

This section is ignored when you run MPF while connected to a physical pinball machine.

volume

The `volume:` section of your machine config file specifies several things, including:

- The names of the audio tracks in your machine.
- The default volume settings for each track.
- How many "steps" you want between 0 (off) and full volume.



This section can be used in your machine-wide config files.



This section *cannot* be

used in mode-specific config files.

There's a default `volume:` configuration in the machine-wide `mpfconfig.yaml` default file. This should be fine for most cases, though you can change it here if you want to do something different.

```
volume:
  tracks:
    master: 20
```



```

voice: 20
sfx: 20
music: 20
steps: 20

```

Note that all audio is controller via the standalone media player. So these MPF core engine volume settings only control what MPF sends to the media controller via BCP. The media controller handles the actual processing of the volume changes.

tracks:

This is a list of track names along with the default (initial power-on) volume settings for each. By default, MPF uses four tracks:

- `master`: the overall volume of the entire machine
- `voice`: a track which plays all voice callouts
- `sfx`: a track which plays sound effects
- `music`: a track which plays the continuous background music

steps:

This is simply how many steps you want between the volume at zero (off, or mute) and full volume. Internally MPF uses 100 steps, but you probably don't want to actually have 100 because it would be really annoying to have to hit the volume up or volume down key so many times. 20 is probably fine for most cases.

window

Controls the on-screen window that MPF creates, including properties of the window itself as well as the initial set of display elements you'd like to show up in that window.



This section can be used in your machine-wide config files.



This section *cannot* be used in mode-specific config files.

```

window:
  width: 800
  height: 600
  title: Mission Pinball Framework
  resizable: yes
  frame: yes
  fullscreen: no
  fps: auto
  quit_on_close: True
  elements:

```

```
- type: VirtualDMD
  width: 512
  height: 128
  h_pos: center
  v_pos: center
  pixel_color: ff5500
  dark_color: 220000
  layer: 1
  pixel_spacing: 2

- type: Text
  font: tall title
  text: JUDGE DREDD
  h_pos: right
  v_pos: bottom
  y: -10
  x: -20
  size: 60
  antialias: yes
  layer: 1
  color: ee9900

- type: Shape
  shape: box
  width: 514
  height: 130
  layer: 2
  thickness: 2
  v_pos: center
  h_pos: center
  color: 00ff00

- type: Movie
  movie: torches
  repeat: True
  layer: 0
```

width:

Width of the window in pixels. Default is 800.

height:

Height of the window in pixels. Default is 600.

title:

Text title of the window. Default is "Mission Pinball Framework".

resizable:

True/False which controls whether this window is resizable.

frame:

True/False which controls whether the window has a "frame" around it (including the title bar, etc).

fullscreen:

True/False which enables full screen mode for the on screen window. If True, the width and height settings are ignored.

fps:

The number of frames per second that the on screen window updates itself. (The "frame rate," or "refresh rate," in other words.) A value of "auto" means that it refreshes at the same rate as the [Machine Hz](#). On some systems (and depending on how many and what type of display elements you have in your window), the system can get bogged down trying to refresh the window too fast, so you can enter a lower frame rate here. You should try out the "auto" setting though. Our main test system in a Windows 7 virtual machine with a 640 x 160 on screen virtual DMD, and we can run 60fps no problem even while we're driving a physical pinball machine. And that's using a VM!

quit_on_close:

Whether MPF should quit when the window is closed. Default is yes. (Note this feature has not yet been implemented.)

elements:

A list of one or more display elements that will be shown as the initial content in the window. See the sections on [Display Elements](#) for the specifics of the different types of elements that you can use, and see their related entries in the [configuration file reference](#) for details of their settings.

11. List of Machines MPF has been used with

MPF has been tested on several different existing machines. In doing so, we have baseline MPF hardware configurations built for the following machines. Note that these are not complete games. In fact some of these were only built in a few hours and have nothing more than the hardware configuration. But if you're interested in using MPF on any of these machines, contact us and we can send you the relevant configuration files you can use as a starting point.

Also if you've used MPF on any other machines and you're willing to share your configuration files, please let us know!

Williams WPC

1. Congo
2. [Demolition Man](#)
3. Hurricane
4. Indiana Jones: The Pinball Adventure
5. [Judge Dredd](#)
6. Red & Ted's Road Show
7. Star Trek: The Next Generation
8. T2

Stern S.A.M.

1. Star Trek

Williams System 11

1. Jokerz
2. Pin*Bot

Other/custom:

1. Aztec (1976 EM rebuilt with modern hardware)
2. Big Shot (1974 EM conversion to modern electronics)
3. Bulleye 301 (Grand Products playfield replacement for early Bally solid-state machines)

There are also several completely custom machines in progress, including:

1. Elemental (Chris Dana)
2. [Minions](#) (The Pinball Amigos from Rotterdam)
3. [The Muppets](#) (Jack Bridges)
4. [Nightmare Before Christmas](#) (Mark Incitti)
5. [Peanuts Pinball](#) (Aaron Davis)
6. [Spaceballs: The Pin](#) (John Marsh)
7. [Tattoo Mystique](#) (Brian Cox)

12. Road Map & Current Status

The current "master" (stable) version of MPF is **0.21.3**. ([download](#))

- Complete list of current MPF features is [here](#).
- List of new features & bug fixes we're working on for the next release is [here](#).
- Long-term goals to get to our "version 1" release are listed below.
- Discuss all this in the [MPF Development forum](#).

Near Term Goals to get us to "v1"

Here's a list of everything we have to get finished up to get us to what we'll call our "1.0" version. (Our goal is to release MPF 1.0 at some point in 2015) :

- Create a single effects controller to handle synchronization of light shows, sounds, and display content. (Right now these are all handled separately.)
- Segmented displays
- Operator mode / service menu
- Bad switch handling
- Status display (show info when both flipper buttons are held in)
- Flipper codes
- Match
- Bonus Mode (We have a how to guide which shows how to do it with code. We need to create a version based on config files.)
- Servo / stepper / motor support
- A proper installer and inclusion in PyPI
- Move our graphics and sound off of Pygame and onto SDL2.
- A graphical-based "pinball builder" tool which lets you setup MPF graphically.
- A graphical playfield tool that lets you see lights and activate switches on your computer screen.

If you have ideas you'd like us to add, please [contact us](#). (Or feel free to write them yourself. We're happy to help you with this. :)

If you want to see our vision for MPF beyond 1.0, check out our blog post on our [long term roadmap and vision for the future of pinball](#).

Changes in MPF v0.30

The next version of MPF will be v0.30. It's currently in active development and is expected to be released in March with a preview available in a few weeks. ([Details here.](#))

The latest code for MPF v0.30 is in the `dev` branch in GitHub which is available [here](#). The latest `dev` code for MPF_MC (the MPF media controller) is available [here](#). Neither of these two projects are ready for public use, but you can check the commits to see what we're working on. We have been very busy, with lots of commits every day to both projects.

This document was last updated **Mar 4, 2016**

Release Schedule

We hope to have a preview release of MPF 0.30 available by March 7. We're hoping to get the final release out by March 31. Of course these schedules may change depending on what comes up, but those are our goals at this point.

What's new & changed in MPF 0.30

Big Changes

- Moved to Python 3.4 (from Python 2.7)
- Separated the media controller component (`mpf-mc`) into a separate project. This way you won't have to download all the media controller prereqs if you're using the Unity media controller.
- Completely rewritten media controller (like, from scratch), based on Kivy. (No more Pygame.) This is able to leverage the GPU, play modern video formats, and is all-around faster, better, and nicer than before.
- Created a proper installer and registered it with PyPI, so you can install MPF simply by running `pip install mpf`.
- Completely rewritten config file migrator that can now save comments and white space. This means the config migrator can now do everything.
- Changed the way MPF starts. "mpf" is registered with your system, so you just switch to your machine folder and just run "mpf" to load that machine in MPF (or "mpf mc" to run the media controller, or "mpf migrate" to migrate your config files, etc.)
- Light scripts have been combined into shows, and now shows are powerful and can be used to control anything (lights, LEDs, slides, sounds, coils, GI, etc.) Also you can put placeholder "tokens" anywhere in a show which are replaced in realtime with the parameters you pass to the show to run. (That's how shows replace light scripts,

except they're much more powerful now because they work with anything, not just lights and LEDs.)

Other changes

- New RGB Color class that supports other color layouts (HSV, HSL) and hardware LEDs that are not in RGB order (like BGR). Also named colors are supported, and you can redefine color names dynamically.
- LED smooth fades which can be done in hardware, software, or both.
- New MPF clock class replaces the existing timer_tick, events, and tasks in a way that is much more flexible and "smooths out" the CPU processing across the main loop.
- Shows are changed from "ticks" to real-world time, with show steps that can be in absolute time or delta time (with mix-and-matching in shows)
- All graphics can now use the GPU which gives us massive performance improvements and lowers CPU utilization.
- Completely rewritten audio interface which supports all the important pinball concepts like ducking with custom per-sound ramps.
- Completely rewritten asset manager which includes support for asset pools (one asset can have multiple files under the hood, so a single sound request can randomly return a sound from a pool, for example).
- Display widgets can be added to existing slides, making display components easier to re-use (rather than having to build completely new slides all the time.)
- Support for accelerometer-based tilts.
- Ball search.
- Servo support.
- Improved ball device logic which makes it harder for MPF to get confused about where balls are.

13. Version History

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.)

0.21

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

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 Linux shell 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

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 cause MPF 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

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

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 "ticks"
- Created a centralized config processing module

0.15

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

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

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

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
- Added the 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

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

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

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 were 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 an 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

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 titl 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

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

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

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

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

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 virtual hardware platform for virtual / software only testing that does not require P-ROC or FAST drivers.

0.2

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

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.

14. MPF 0.30 Release Notes

Version 0.30 is the biggest change to MPF yet in terms of changes to a single version. We knew we had to make some big changes at some point to some internal architecture things, so we decided to "rip the Band Aid off" and do it all at once. The good news is that most of the fundamental concepts of MPF did not change, (it's mostly internal architecture stuff). Also, we have a migration tool which will migrate your existing config files and show files to the new format, so you should be able to get your game up and running quickly on the new version.

Note that this document merely lists the big new features and changes to MPF 0.30. All of these changes will be incorporated into the general MPF documentation as we get that documentation updated. If you have an existing machine project on MPF 0.21, refer to our MPF 0.21 to MPF 0.30 migration guide for details on how to migrate your machine code and configs to MPF 0.30.

Here's an overview of the changes to MPF in v0.30. Not every change is listed here, but these are the big ones:

(1) MPF now requires Python 3

Prior versions of MPF used Python 2.7. Python 2.7 is now about 8 years old, and no longer receives new features. Python 3 was released in 2008 and is currently on version 3.5.

MPF 0.30 requires at least Python 3.4. If you plan to use the MPF Media Controller on Windows (more on that in a bit), you need to use Python 3.4. If you're on Mac or Linux, or you're on Windows but you plan to use the Unity Media Controller, you can use Python 3.4 or 3.5. We recommend you use the latest patch release of Python you can. (Currently 3.4.4 or 3.5.1)

MPF (and the MPF Media Controller) can run in 32-bit or 64-bit mode. We recommend that you run the version that matches your OS.

(2) Mac OS X is officially supported

Prior versions of MPF leveraged a multimedia library called "Pygame" that was very buggy on the Mac, leading us to recommend that if you are a Mac user, you run MPF in a VM running either Ubuntu or Windows. MPF 0.30 no longer uses Pygame, so you can use MPF on a Mac natively.

(3) The Media Controller is now a separate package from MPF

Starting in MPF 0.30, the MPF Media Controller (also referred to as the "Media Controller" or simply the "MC") is now a separate package from MPF. This means there are two packages in GitHub: [mpf](#) (the MPF core engine) and [mpf-mc](#) (the media controller). We did this for several reasons:

- It allows us to tie the pre-reqs for the MC to the MC package, meaning if you're not using the MPF-MC (perhaps because you're using the Unity-based MC or because you have an EM machine that doesn't have graphics and sound), now you don't have to mess with all the pre-reqs for the MC which you won't use just to use MPF.
- It lets us separate the release cycles of the two projects and handle patches and updates separately.

From an installation standpoint, it's still really easy to install both MPF and the MPF-MC at the same time, so really the fact that they're two separate packages doesn't impact you, but it's important to know.

(4) The MPF-MC has been completely rewritten from scratch

The new MPF-MC component has been completely rewritten from scratch for MPF 0.30. The reasons for this are covered [here](#), but the main reason is the fact that the MPF-MC for v0.30 uses a multimedia library called "Kivy" instead of Pygame. We had many problems and limitations with Pygame, including:

- Pygame is not maintained anymore. The last release was in 2009, which means it only works with the versions of Python that were current in 2009.
- Bugs that have been found since 2009 have not been fixed. (For example, the Mac version of Pygame has a memory leak which is why we can't support MPF on a Mac.)
- The audio system in Pygame is very basic. It only allows 2-channels (no surround sound), 8 simultaneous voices, and a single music track.
- Pygame does all of its graphical processing in the CPU, meaning it does not leverage the GPUs in modern computers. This leads to both lower performance and higher CPU usage. (Things that a GPU can do almost instantly take a lot of processing power if they're done on a CPU.)
- Pygame only supports video if it's encoded into MPEG-1 format with very obscure settings which does not lead to high quality videos. (And Pygame on the Mac doesn't support video at all!)

- Pygame cannot precisely control where the graphical window pops up on the screen, which is a problem for machines that have an LCD in the backbox that is partially obscured.

The [Kivy](#) multimedia library is modern and updated often. It leverages modern base multimedia libraries (like [SDL2](#) and [GStreamer](#)), and it can fully leverage the GPU. It's also licensed via the MIT license (which is the license MPF uses), and Kivy apps can also be built for iOS and Android (in addition to Windows, Mac, and Linux), opening up future possibilities for MPF.

(5) Proper Python package installers, and inclusion in PyPI

Starting in MPF 0.30, we have created "proper" Python installers (rather than the hand-built scripts we used in the past). We have also uploaded the MPF source packages into the Python Package Index ([PyPI](#), also called the "Cheese Shop"), which is sort of like an app store for Python packages.

This means that installing MPF is now as simple as running `pip install mpf` from a command line, and it will automatically connect and download all the prerequisites you need. (Actually if you run `pip install mpf-mc`, that will install the MC which itself includes MPF as a prereq, so that single command will give you everything you need.) The packages in PyPI are pre-compiled for Windows and Mac, so installation is fast and easy and nothing has to be compiled.

This change also means that MPF and MPF-MC are installed into your Python site-packages folder, and updating them is as simple as running another pip command.

(6) Change to how MPF is launched

Now that MPF has a proper installer and is installed from PyPI, the way you actually launch MPF has changed. Starting in MPF 0.30, you simply open a command prompt and switch to your machine folder, then simply run `mpf` to launch the `mpf` core engine, or `mpf mc` to launch the MC. (On Windows you can alternately run `mpf both` to launch both.)

(7) Config files are now version 4

MPF 0.30 changes the `#config_version` setting in your machine config files to `#config_version=4`. (Again, the migration tool will migrate your existing config files to the new format. See our How To guide on migrating from MPF 0.21 to 0.30 for details.)

(8) Changes to show files and formats

The biggest change to MPF in 0.30 (besides moving from Pygame to Kivy) is that the internal processing of shows has completely changed. There are several changes here, so let's look through them one-by-one:

(A) All shows are driven by MPF

Prior to MPF 0.30, it was kind of confusing because there were actually two different show players. One of them ran on the "MPF side" of things and was responsible for shows that included machine hardware, like lights, LEDs, coils, flashers, etc. The other ran on the "MC side" of things and handled shows which included sounds and display stuff.

This was bad for several reasons:

- It was confusing.
- It was difficult (or impossible) to create single shows that included MPF and MPF-MC components. (e.g. how do you make a show that includes synchronized lights and sounds?)
- Starting and stopping shows was hard because what if MPF started a show, and MPF-MC tried to start a show too, but the MPF-MC was busy so it had to queue that show, but the MPF show was already started, so you had them out of sync, then you tried to cancel one... it was a mess.

In MPF 0.30, there is only one show controller and one type of show. They all run on the MPF side of things. This single show has the ability to send commands (via BCP) for things that should be run on the MC side, so a show can still include sounds or slides and MPF will send those instructions, when that show step comes up, via BCP to the MC where the MC can process and play them.

(B) Show content is "played" by the standard config_players

Another weird thing about shows in prior versions of MPF is that the way shows "played" each step was different from the way a config_player would play a config. (A "config_player" is a section of the config that "plays" something, like the light_player, led_player, sound_player, slide_player, etc. It's basically what maps some MPF event to some kind of action to be played.)

So in the old version of MPF, in a show, you might have a *lights:* section which contained instructions for what lights to "play" in that step, and then you might also have a *light_player:* section of a mode or machine config file which also contains instructions for what lights to "play" when a certain event is posted. The problem was that those were two completely sections of code (one was in the show controller and the other was in the light_player). This

was bad because the exact formats of the options were not identical between the two different types of players, and different ones had different features, etc.

So in MPF 0.30, we said, "Why do these two types of players have to be different? If we have something called a "light player", should that same code be used to play whatever is in the *lights:* section in a show step and also to play whatever is in the *lights_player:* section of a config file when that even is posted?

These are what are known as "config players" in MPF 0.30. (Note that the word "player" here has nothing to do with a human player of a pinball game.)

MPF 0.30 has several built-in config players, including:

- bcp_player
- coil_player
- event_player
- flasher_player
- gi_player
- led_player
- light_player
- random_event_player
- show_player
- trigger_player

Also, the *config_player* functionality of MPF allows for plugins which can play their own types of configs, and if you have the MPF-MC installed, it registers three additional config players:

- slide_player
- sound_player
- widget_player

Again, the beauty here is that each of these players is responsible for that player's section of the config file and also for that player's section of a show step. So the "flasher_player" handles both the *flasher_player:* of the machine or mode config file and also the *flashers:* section of a show step.

This is also great for reliability and testing. Now each player doesn't have to have its own logic for registering and deregistering events and everything—instead that's all shared from a base class and each individual player only needs to know how to deal with its own type of devices.

(C) Shows become playlists

Prior versions of MPF were very hit-or-miss when it came to putting together playlists, again because every component had to implement its own type of playlist. In MPF 0.30, since `config_players` are responsible for the actual processing of each step of a show, the show files and the show controller essentially become "dumb" containers. (This is a good thing.)

It also means that you can start, stop, and advanced other shows within the steps of a show (since the `show_player` is now just a `config_player` like anything else). This also means that you can use a show like a playlist, adding whatever you want to each step of a show (again, including other shows). In other words, MPF 0.30 now supports playlist of anything.

We will also be adding universal transition support, so each `config_player` can implement its own transitions which can be used within shows. (Currently transitions are manually implemented for sounds and slides. We'll make those more generic and universal soon.)

(D) Tocks: are gone, shows now operate on real-world time

The concept of *tocks*: has been removed from show files. Now, each step of a show is driven by a *time*: setting (which can be like any time string in MPF, including milliseconds, seconds, minutes, etc.).

MPF 0.30 also adds the option for time settings for individual show steps to be specified in relative terms (time after the previous step) or in absolute terms (time since the beginning of the show). Absolute time steps was needed when synchronizing lights and LEDs with sounds and videos.

Even though show steps are configured based on time, you can still set the playback speed of a show to play a show faster or slower, and you can still change the playback speed of a running show.

Show in MPF <= 0.21:

```
- tocks: 1
  leds: ...
- tocks: 2
  leds: ...
```

Show in MPF 0.30:

```
- time: 0
  leds: ...
- time: +1
  leds: ...
- time: +2
```

The *time:* entries in shows are [standard MPF time strings](#). The default is seconds, but you can enter ms, minutes, etc. Also note the + before the time in the example above. Time values that start with plus are *incremental*, meaning they are the time since the previous step ended. If you don't enter a plus sign, that means the time is *absolute* from the beginning of the show. (Absolute values make it easy to sync show steps with audio or video.)

Also note that the time values for show steps in MPF 0.30 are shifted "down" one position. In other words, shows in MPF 0.30 always start with a time: 0, and they always end with an empty final step which specifies the time after the last step when the show ends.

This may seem kind of confusing at first, but it's necessary for the absolute times to work. If you think about it, in old versions of MPF, the *ticks:* value was technically the time when the following step started (since ticks specified how many ticks that step lasted). So in order to make absolute times work in MPF 0.30, we had to change it so the *time:* value of a step was the time when that step *started*, rather than the time when that step *ended*.

Note that the migration tool will add quotation marks around time values that start with +. This isn't actually necessary, it's just something the migration tool does.

(E) Light scripts are gone, replaced by placeholder "tokens" in shows

Prior versions of MPF included light and led scripts, which were like shows except that instead of specifying which lights or LEDs each step would apply to, you passed a list of lights or LEDs when the script was started. (And then the act of playing a script would build up a temporary show with the proper light or LED names inserted into it.)

The problem with this was that light scripts were very specifically written only to cover lights and LEDs, and they were not very flexible. (Even though shows had lots of options, only a subset of those options were exposed to light scripts.)

So in MPF 0.30, we completely removed the concept of light scripts and instead added a placeholder "token" concept to shows. For example, in MPF 0.30, anything that's in parenthesis in a show file will now be replaced (in realtime) with key/value pairs that are passed to that show when it starts.

For example, you could have a show file like this:

```
- time: 0
  (leds): ff0000
- time: +1s
  (leds): 000000
```

Then when you play that show, you could pass a value of `leds=led1` (or a list of values, like `leds=[led1, led2, led3]`), and the `(leds)` sections in the show file will automatically be replaced by the values you pass. The actual names of the tokens can be anything you want.

For example, you could have a line called `(banana) : ff0000` in your show, and then pass `banana=led1` when the show plays, and that will be fine too.

So this is how tokens with placeholder tokens replace light scripts.

This is very powerful for two reasons:

- Since shows now use those universal config players, this means that what used to be called light scripts (which are now these shows with tokens), now these shows can be used with *anything*. You can now have dynamically-replaced placeholder tokens which can work with lights, LEDs, other shows, flashers, events, coils, slides, sounds, etc.
- You can put your placeholder tokens anywhere in a show. So instead of having the `(leds)` token as the list of LEDs in the example above, you could add `(leds) : (color1) and (leds) : (color2)`, and then you could pass `leds=led1, color1=ff0000, color2=000000` to create a show which could dynamically flash any led (or list of leds) between whatever two colors you wanted.

(F) Additional of `#show_version=4`

Since shows now pull so much of their config options from the associated `config_players`, starting in MPF 0.30 you now need to add a `#show_version=4` as the first line of show show YAML file. (This is similar to the `#config_version=4` setting you add to your config files.) The migration utility that comes with MPF 0.30 will automatically add this (as well as converting your existing shows with `tocks:` format to the new `time:` format), and since shows now have the version number in them, future versions of the migrator will continue to be able to migrate your show files if anything ever changes in the future.

(9) Named colors

MPF 0.30 now supports specifying colors by name instead of by hex value. This works anywhere that colors are specified, including in shows and for commands that are executed directly, and it applies everywhere colors are used (for LEDs, display widgets, etc.).

By default, MPF includes a built-in list of the [standard W3C web colors](#). You can also define your own colors and add them to the list.

What's really cool is that the lookup process which converts a color name to a color value is done in every time a color is used, so you can actually redefine or update the values for colors dynamically in your game and the colors will start using those values from there are out. (This is cool for operator settings like white balance and also for using the same effect for different modes where you could have a bunch of effects written to use a color called `modecolor` and then you just keep updating that color value depending on which mode is running.)

(10) Hardware accelerated LED fades

Previous versions of MPF did LED fades and color transitions by repeatedly sending incremental color commands with each "tick" of the MPF clock. That meant that if you were running MPF at 60Hz, a 100ms fade would actually take place in six "steps" that were 16ms apart. (100ms / 60Hz)

In v0.30, MPF can send fade commands to supported hardware to "smooth out" the fades. This means that even though MPF is only sending updates every 16ms, it can tell the hardware LED controller that it wants to fade from one color to the next, and the hardware controller can do several sub-step fades in-between MPF steps.

This is currently supported on FAST LED controllers, and will soon be added to Multimorphic PD-LED and FadeCandy controllers.

(11) Asset Pools

MPF now supports grouping assets into asset "pools" which is where you have multiple physical asset files that are combined together into a single asset name. For example, instead of just playing the same slingshot sound over and over every time a sling shot is hit, you could actually create four different sounds, and the asset manager will play a different sound each time. You can control how each sound is selected too. (Random, weighed random, round robin, random but play them all before repeating any, etc.)

(12) Ball Search

MPF 0.30 now includes a proper ball search feature, with advanced options to control things like timing between searches, order devices are searched, and what happens during multiple phases of the search. (Maybe if there's a device holding a ball, you don't fire that device's eject coil during the initial ball search rounds, but if after 5 search rounds it still hasn't found the ball, you can try pulsing that coil too.) This is all exposed via the config files.

(13) Accelerometer-based tilts

If you're using MPF with a hardware platform that supports accelerometers, you can now configure the tilt to be based on the accelerometer with g-force settings.

(14) Servo Support

MPF 0.30 now includes built-in support for servo devices accessible via several hardware platforms.

(15) Text Strings

When you specify text for the display (either the on screen display or via a DMD), you can now use a dollar sign to specify a text string lookup instead of adding the actual text to your config.

For example, in a text widget config:

```
- type: text
  text: $welcome
```

And then in your config file:

```
text_strings:
  welcome: Welcome!
```

This gives you the flexibility to use different config files with different sets of text strings. For example, you might have a "mature" and "family-friendly" versions of text strings, or you could have different text strings for different languages.

Note that these text strings are just for static text lookup. You can still access player variables, machine variables, and event parameters in text widgets the same way you always could (though now variables can contain text strings and vice-versa).

(16) Placeholder variables %var% -> (var)

The placeholder variables for player, machine, and event parameters in text strings have been changed from percentage signs to parenthesis.

MPF <= 0.21:

```
- type: text
  text: BALL %ball%
```

MPF 0.30:

```
- type: text
  text: BALL (ball)
```

Not only is this cleaner visually, it also means you don't have to put quotation marks around values that start with %.

Deprecations

MPF 0.30 removed the following features:

- .DMD formatted files are no longer supported. (We think they're not necessary but can add support for them if needed.)
- External shows (This is temporary. They'll come back in 0.31.)