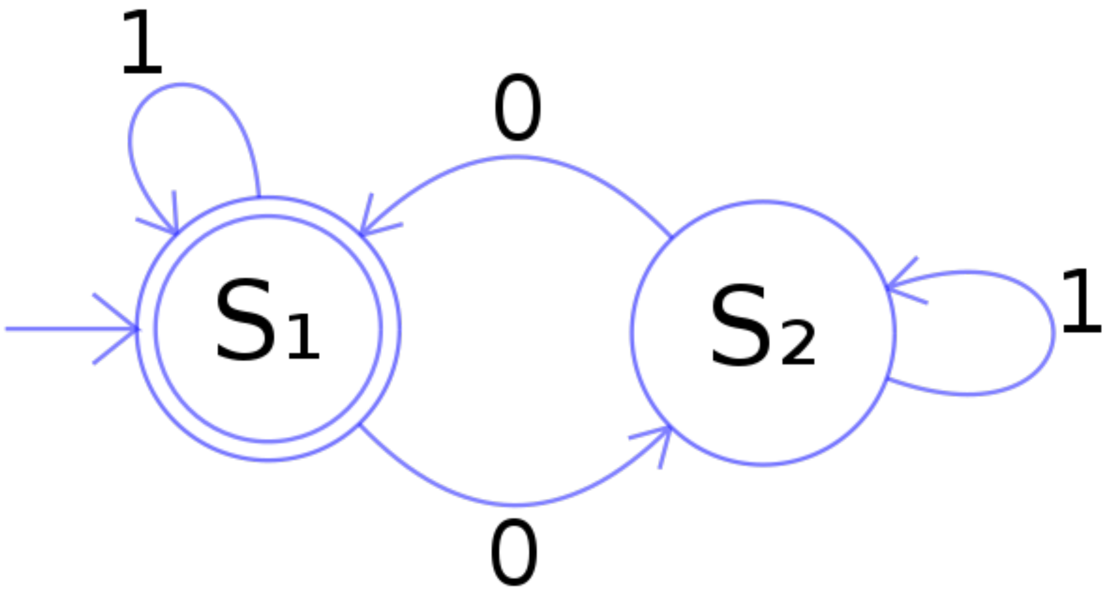
CircuitPython 101: State Machines, Two Ways

Created by Dave Astels

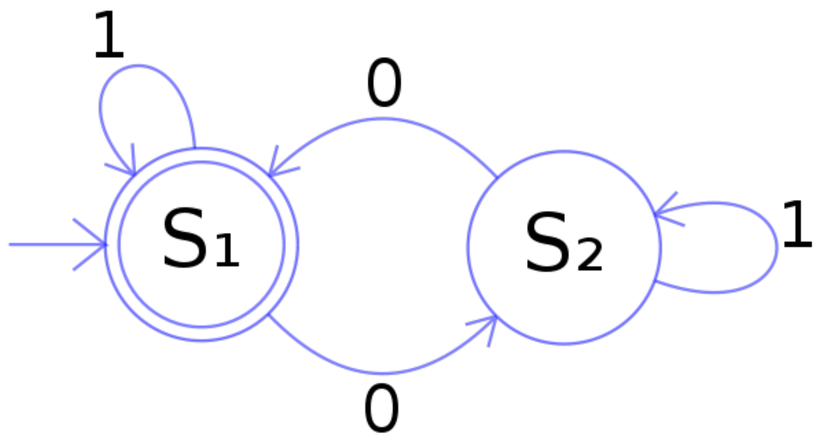


https://learn.adafruit.com/circuitpython-101-state-machines

Last updated on 2023-08-29 03:57:30 PM EDT

Table of Contents

Overview



The author has used state machines in several CircuitPython guides, most recently for the New Years Eve dropping ball.

So what is a state machine?

Accoding to Wikipedia:

> A finite-state machine (FSM) or finite-state automaton (FSA, plural: automat a), finite automaton, or simply a state machine, is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some external inputs; the change from one state to another is called a transition. An FSM is defined by a list of its states, its initial state, and the conditions for each transition.

So what's all that mean? Simply put, there's some number of states and a way to move between them.

What's a state? Let's use the example of the states of matter: solid. liquid, and gas (we'll ignore plasma to keep this simple). There are well defined transitions to move between these states:

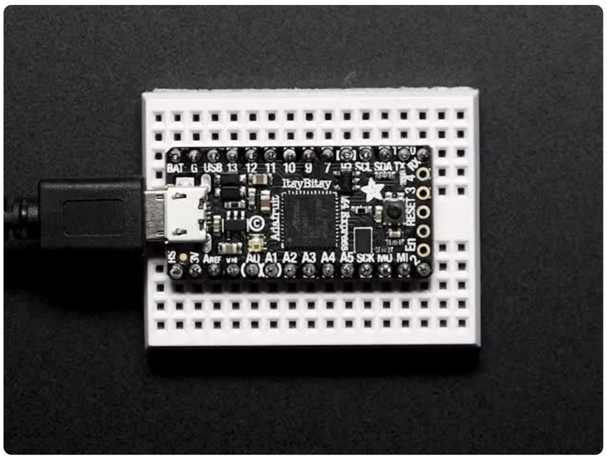
- melting transitions from solid to liquid,
- freezing transitions from liquid to solid,
- evaporating transitions from liquid to gas,
- condensing transitions from gas to liquid, and
- sublimating transitions from solid to gas.

Each of these have conditions in terms of material, temperature, and pressure that control whether or not the transition can happen. A simplified example of this is that when its temperature is less than 0C (32F), water will freeze transitioning it from a liquid state to a solid state.

The state machine in the diagram at the top of this page processes a stream of 1s and 0s (as label the transitions). When the machine is in S1 it has processed an even number of 0s, and an odd number when in S2.

A huge advantage of working with a modern, capable MCU like the SAMD51 with it's ARM Cortex-M4 core and spacious memory is that we can make use of more advanced programming techniques that require overhead that smaller, simpler, slower MCUs simply can't handle.

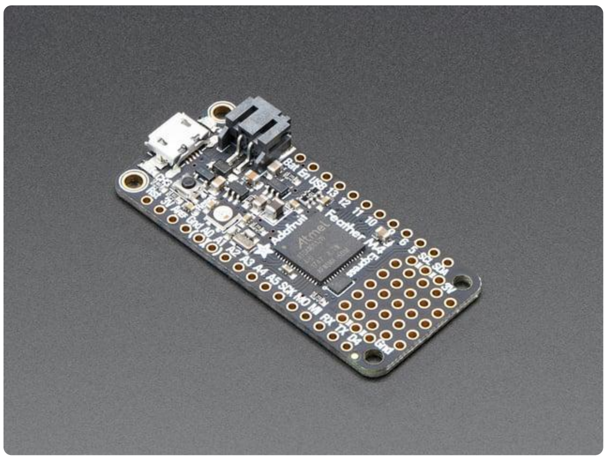
Featured Parts



Adafruit ItsyBitsy M4 Express featuring ATSAMD51

What's smaller than a Feather but larger than a Trinket? It's an Adafruit ItsyBitsy M4 Express featuring the Microchip ATSAMD51! Small,...

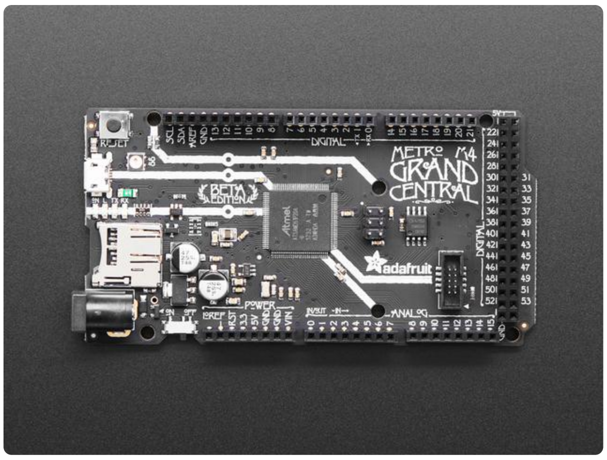
https://www.adafruit.com/product/3800



Adafruit Feather M4 Express - Featuring ATSAMD51

It's what you've been waiting for, the Feather M4 Express featuring ATSAMD51. This Feather is fast like a swift, smart like an owl, strong like a ox-bird (it's half ox,...

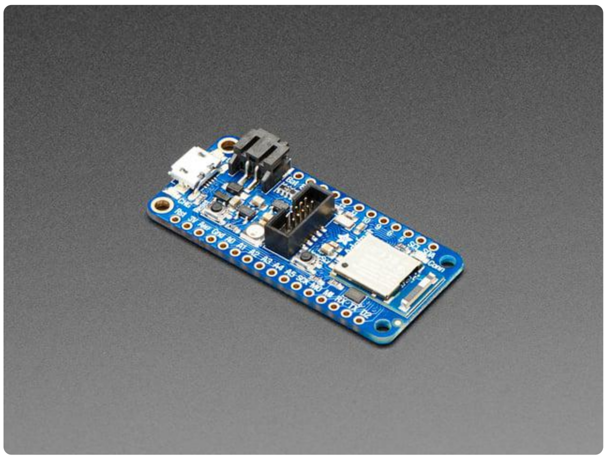
https://www.adafruit.com/product/3857



Adafruit Grand Central M4 Express featuring the SAMD51

Are you ready? Really ready? Cause here comes the Adafruit Grand Central featuring the Microchip ATSAMD51. This dev board is so big, it's not...

https://www.adafruit.com/product/4064



Adafruit Feather nRF52840 Express

The Adafruit Feather nRF52840 Express is the new Feather family member with Bluetooth Low Energy and native USB support featuring the nRF52840! It's...

https://www.adafruit.com/product/4062

Code

We'll be using CircuitPython for this project. Are you new to using CircuitPython? No worries, there is a full getting started guide here ().

Adafruit suggests using the Mu editor to edit your code and have an interactive REPL in CircuitPython. You can learn about Mu and its installation in this tutorial ().

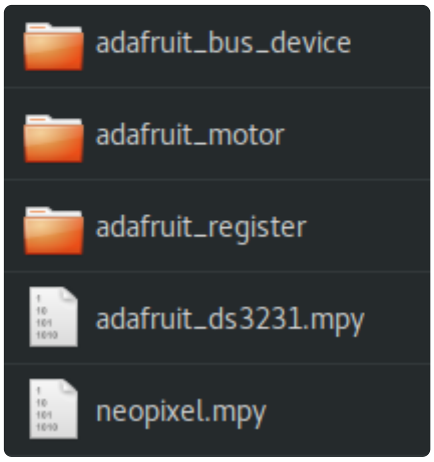
Install the latest release of CircuitPython, version 4 (). Follow the instructions here () using the appropriate CircuitPython UF2 file.

Get the latest 4.0 library pack (), unzip it, and drag the libraries you need over into the /lib folder on CIRCUITPY.

For this project you will need the following libraries:

- adafruit_bus_device
- adafruit_motor
- adafruit_register
- adafruit_ds3231.mpy
- neopixel.mpy

When these are installed, your CIRCUITPY/lib directory should look something like:



```
adafruit_bus_device
adafruit_motor
adafruit_register
adafruit_ds3231.mpy
neopixel.mpy
```

Example Project

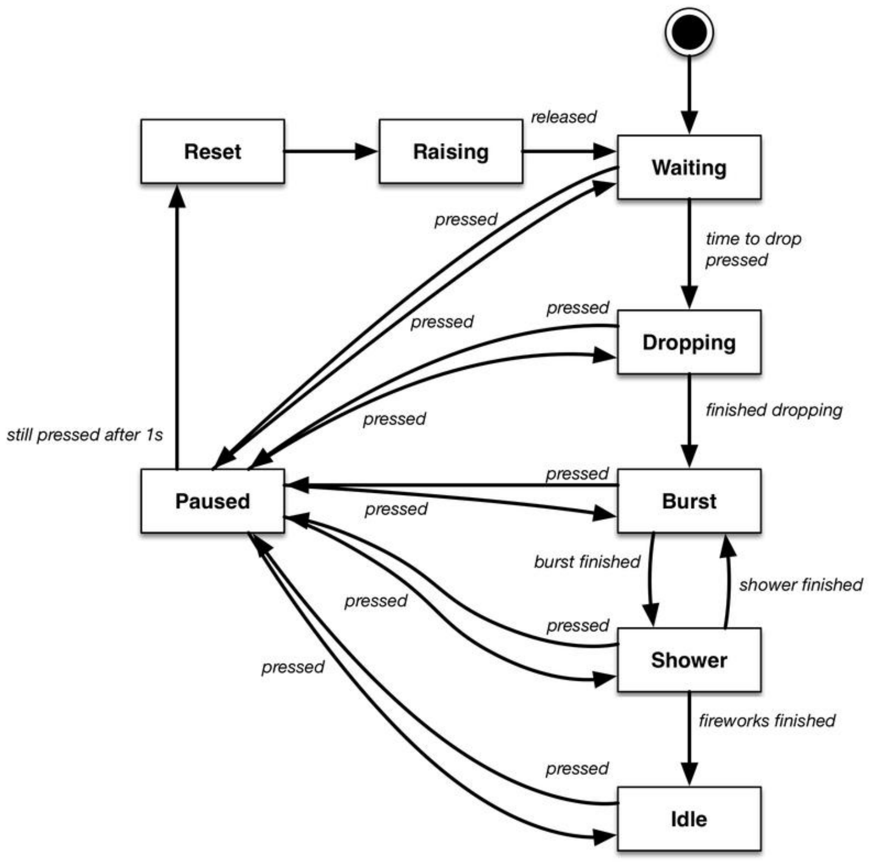
The hardware this code is meant to run on is described in the NYE Ball Drop project () by the Ruiz Brothers.

There are two versions of the codebase, each in it's own subdirectory. You can load it all using the download Project Zip link below.

State Machine example code

The NYE Ball Drop Machine

There's a standard way to draw state machines that can get quite detailed. Here we'll use a fairly basic form. Below is the state machine from the NYE ball drop guide. We'll explore a couple ways to implement it.



The rectangles are states, the lines between them are the transitions. Each conditional transition is labeled with the condition.

States are:
Waiting - The machine sits here waiting until the time read from the real time clock hardware reaches 10 seconds to midnight on Dec 31, or the switch being pressed.
Dropping - The ball is dropping, its NeoPixels are color cycling in a rainbow pattern, and the countdown sound clip is playing. Once the ball has finished dropping, Auld Lang Syne plays.
Burst - A random color is set on the NeoPixels and grows from black to full brightness, stops there for a bit before fading back to black. The effect is mean to look like a fireworks explosion.
Shower - A NeoPixel effect is portrayed that is meant to look like the shower of sparks falling after the initial explosion (using the same color).
Idle - Once the ball has dropped and the fireworks are over, the machine sits here indefinitely.

Paused - This is an interesting state. It is entered from most other states when the switch is pressed. It is exited, returning from whence it came when the switch is pressed again. However, if the switch is held pressed for a over a second the Reset state is entered.
Reset - various machine state variables are reset in preparation for running through the sequence again.
Raising - The ball is raised again in preparation to be dropped. This continues for as long as the machine is in this state. Once the switch is released, the Waiting state is entered.

Some of transition conditions are based on hardware: pressed, released, have to with the switch. Others are a bit more abstract, e.g. fireworks finished which is true when the fireworks effect has run for the required amount of time.

We're going to look at two approaches to implementing state machines using this one as an example. It has some interesting challenges that make it a good case study.

Brute Force

Implementing this project, we have a large conditional structure in the main `while True` loop with an `if` clause for each state. Each time through the loop, only the clause corresponding to the current state is executed.

There's some oddity around the switch. Specifically the PAUSED state is handled separately: if the switch is pressed (and therefore its value fell) we return to the state which was paused, as well as resuming any audio that was playing and servo motion.

If it hasn't been just pressed we see if its still being held down and has been for a second. In that case we transition to the RESET state.

Since the pause operation applies to all but the WAITING state, it's handled outside any state: state, audio, and servo motion are saved (and paused/stopped) and we transition to the PAUSED state. This happens when the switch is pressed, in all but the waiting state (or the paused state as it was already considered previously).

The rest of the states are handled more conventionally. There is an if clause for each state that determines what happens while in that state, as well as when to move to another state and what happens as we do.

For example, in the WAITING state, nothing happens until the switch is pressed or the time reaches 10 seconds to midnight on Dec. 31. When either of those happen the countdown clip starts playing, the servo starts turning, the rainbow effect is initialized, and the time to stop the drop is set. This last bit is required since the switch can be used to start the drop at any time. Finally, the DROPPING state is transitioned to.

The remaining states differ in the details, but the general structure is similar.

```
while True:
    now = time.monotonic()
    t = rtc.datetime
    switch.update()

    if state == PAUSED_STATE:
        log("Paused")
        if switch.fell:
            if audio.paused:
                audio.resume()
            servo.throttle = paused_servo
            paused_servo = 0.0
            state = paused_state
        elif not switch.value:
            if now - switch_pressed_at &gt; 1.0:
                state = RESET_STATE
        continue

    if switch.fell and state != WAITING_STATE:
        switch_pressed_at = now
        paused_state = state
        if audio.playing:
            audio.pause()
        paused_servo = servo.throttle
        servo.throttle = 0.0
        state = PAUSED_STATE
        continue

    if state == WAITING_STATE:
        log("Waiting")
        if switch.fell or (t.tm_mday == 31 and
                           t.tm_mon == 12 and
                           t.tm_hour == 23 and
                           t.tm_min == 59 and
                           t.tm_sec == 50):
            start_playing('./countdown.wav')
            servo.throttle = DROP_THROTTLE
            rainbow = rainbow_lamp(range(0, 256, 2))
            log("10 seconds to midnight")
            rainbow_time = now + 0.1
            drop_finish_time = now + DROP_DURATION
            state = DROPPING_STATE

    elif state == DROPPING_STATE:
        log("Dropping")
        if now &gt;= drop_finish_time:
            log("***Midnight")
            servo.throttle = 0.0
            stop_playing()
            start_playing('./Auld_Lang_Syne.wav')
            reset_fireworks(now)
            firework_stop_time = now + FIREWORKS_DURATION
            state = BURST_STATE
            continue
        if now &gt;= rainbow_time:
            next(rainbow)
            rainbow_time = now + 0.1

    elif state == BURST_STATE:
        log("Burst")
        if burst(now):
            state = SHOWER_STATE
            shower_count = 0

    elif state == SHOWER_STATE:
        log("Shower")
        if shower(now):
            if now &gt;= firework_stop_time:
                state = IDLE_STATE
            else:
                state = BURST_STATE
                reset_fireworks(now)

    elif state == IDLE_STATE:
        log("Idle")

    elif state == RESET_STATE:
        log("Reset")
        strip.fill(0)
        strip.brightness = 1.0
        strip.show()
        if audio.playing:
            audio.stop()
        servo.throttle = RAISE_THROTTLE
        state = RAISING_STATE

    elif state == RAISING_STATE:
        log("Raise")
        if switch.rose:
            servo.throttle = 0.0
            state = WAITING_STATE
```

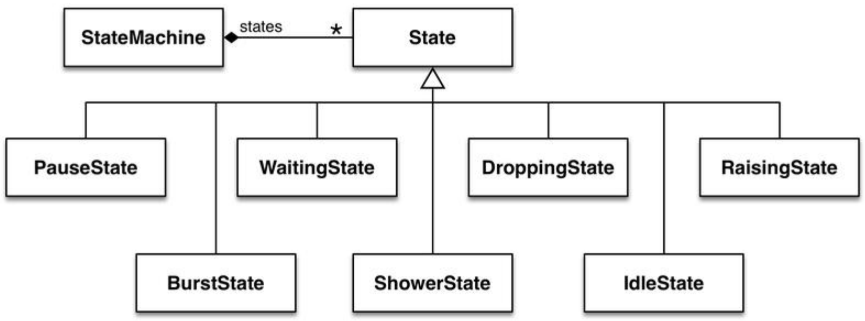
Notice that at first it looks like there's no way out of the IDLE state, but remember that near the top of the loop is the way to get into the PAUSED state by pressing the switch. This is the way out of IDLE: press and hold the switch to reset the system and raise the ball. That will result in the machine getting into the WAITING state.

Above is just the main loop that implements the core of the machine. The rest of the code can be found in the zip that you can download on the Code page ().

Discussion

This method has pros and cons. It's simple for small, simple machines but as the machine's size and complexity increases so does the code length and complexity. Even for this machine, while not being overly large or complex, the code is getting a bit obtuse.

Using Classes



If we make use of CircuitPython's object oriented (OO) capabilities we can make a far cleaner implementation. Each state becomes a class. We can use inheritance to create a simple machine that manages transitions. The machine just needs to track it's current state, not caring what exactly it is. As before, the pause feature in this project adds a little wrinkle to this.

For simplicity, everything before the main loop stays more-or-less the same (of course we don't need the state constants). The main loop will be replaced by what's described below.

State

Before we go further we need to look at the abstract `State` base class. This defines the interface that all state conform to (in Python this is by convention, in some languages this is enforced by the compiler). It also contains any functionality that is common to all states.

Looking at the class below we see 5 pieces of a state's interface:

constructor - This is expected. While the implementation here does nothing , it could. The concrete state classes call this before doing any of their own initialization.
name - this property returns the name of the class. Here it returns an empty string since the State class is abstract and should never be instantiated. Subclasses return

their name.
enter - This is called when a state becomes the active state (i.e. as it is entered).
exit - This is called when a state ceases to be the active state (i.e. when it is exited).
update - This is called on the active state each time through the main loop. Each state's update method calls this first, before doing it's own processing. You can see that is how pausing is implemented. When the active state is updated, this is executed first, checking for a switch press and pausing if it has been detected. Notice how a boolean is returned. That signals whether pausing has happened. If so, the "active" state is no longer active and should skip the rest of its update. States that are not pausable (paused and waiting) do not call this in their update method.

```
class State(object):

    def __init__(self):
        pass

    @property
    def name(self):
        return ''

    def enter(self, machine):
        pass

    def exit(self, machine):
        pass

    def update(self, machine):
        if switch.fell:
            machine.paused_state = machine.state.name
            machine.pause()
            return False
        return True
```

The State Machine

A state machine implementation is a class that manages states and the transitions between them. Using inheritance in OO and especially in a dynamically typed language like Python, the machine can be happily unaware of what the states are as long as they have a consistent interface. The way the paused state works in this project muddies that slightly in that the machine is responsible for the state interactions with the paused state. Because of that the paused state can be seen as almost part of the machine. As such they'll be discussed together.

The machine has a constructor that simply initializes some instance variables. Some are machine related, some are used to implement pausing, but a few are here because it's a common place to coordinate between states. In this case we have some variables used to coordinate between the fireworks states. A better way would be to use a composite state for this: a state that has a state machine embedded in it. That adds a little more complexity than desired for a fairly introductory guide.

Similarly, there are methods for adding states to the machine (`add_state`), transitioning to a different state (`go_to_state`), and updating each time through the loop (`update`). There are two methods for pausing (`pause` and `resume_state`), as well as a fireworks support method (`reset_fireworks`).

```
class StateMachine(object):

    def __init__(self):
        self.state = None
        self.states = {}
        self.firework_color = 0
        self.firework_step_time = 0
        self.burst_count = 0
        self.shower_count = 0
        self.firework_stop_time = 0
        self.paused_state = None
        self.pixels = []
        self.pixel_index = 0

    def add_state(self, state):
        self.states[state.name] = state

    def go_to_state(self, state_name):
        if self.state:
            log('Exiting %s' % (self.state.name))
            self.state.exit(self)
        self.state = self.states[state_name]
        log('Entering %s' % (self.state.name))
        self.state.enter(self)

    def update(self):
        if self.state:
            log('Updating %s' % (self.state.name))
            self.state.update(self)

    # When pausing, don't exit the state
    def pause(self):
        self.state = self.states['paused']
        log('Pausing')
        self.state.enter(self)

    # When resuming, don't re-enter the state
    def resume_state(self, state_name):
        if self.state:
            log('Exiting %s' % (self.state.name))
            self.state.exit(self)
        self.state = self.states[state_name]
        log('Resuming %s' % (self.state.name))

    def reset_fireworks(self):
        """As indicated, reset the fireworks system's variables."""
        self.firework_color = random_color()
        self.burst_count = 0
        self.shower_count = 0
        self.firework_step_time = time.monotonic() + 0.05
        strip.fill(0)
        strip.show()
```

Paused State

When the paused state is entered, it grabs the time the switch was pressed (since it can only be entered in response to the switch being pressed, we know it was just pressed). This is used in `update` to determine how long the switch has been held. Audio and servo are stopped.

In `update` (note that the base class's `update` is NOT called) a switch press is checked for. If it's found audio and servo are resumed and the machine is told to go back to the state that was paused (notably without entering it). If the switch wasn't just pressed but is pressed, it must still be pressed from the entry to the paused state. If it's been a second since then, the system is reset by transitioning to the `raising` state.

```
class PausedState(State):

    def __init__(self):
        self.switch_pressed_at = 0
        self.paused_servo = 0

    @property
    def name(self):
        return 'paused'

    def enter(self, machine):
        State.enter(self, machine)
        self.switch_pressed_at = time.monotonic()
        if audio.playing:
            audio.pause()
        self.paused_servo = servo.throttle
        servo.throttle = 0.0

    def exit(self, machine):
        State.exit(self, machine)

    def update(self, machine):
        if switch.fell:
            if audio.paused:
                audio.resume()
            servo.throttle = self.paused_servo
            self.paused_servo = 0.0
            machine.resume_state(machine.paused_state)
        elif not switch.value:
            if time.monotonic() - self.switch_pressed_at &gt; 1.0:
                machine.go_to_state('raising')
```

Now we can look at each other state in turn.

Waiting State

This state is responsible for doing nothing. Until, that is, the user presses the switch or the time reaches 10 seconds to midnight on New Year's Eve. At that point it causes a transition to the dropping state.

```
class WaitingState(State):

    @property
    def name(self):
        return 'waiting'

    def enter(self, machine):
        State.enter(self, machine)

    def exit(self, machine):
        State.exit(self, machine)

    def almost_NY(self):
        t = rtc.datetime
        return (t.tm_mday == 31 and
                t.tm_mon == 12 and
                t.tm_hour == 23 and
                t.tm_min == 59 and
                t.tm_sec == 50)

    def update(self, machine):
        if switch.fell or self.almost_NY():
            machine.go_to_state('dropping')
```

Dropping State

The dropping state has quite a bit going on.

It starts with a constructor that allocates a couple variables for the rainbow effect, as well as one to hold the time at which to stop the drop.

The `enter` method is responsible for starting the countdown audio file, turning on the servo to drop the ball, starting the rainbow effect, and setting the time to end the drop.

The `exit` method stops the audio and servo, as well as setting up for the fireworks effect.

Finally, `update` checks if it's time to end the drop. If so it has the machine transition to the burst state. Otherwise, if it's time to advance the rainbow effects it does so.

```
class DroppingState(State):

    def __init__(self):
        self.rainbow = None
        self.rainbow_time = 0
        self.drop_finish_time = 0

    @property
    def name(self):
        return 'dropping'

    def enter(self, machine):
        State.enter(self, machine)
        now = time.monotonic()
        start_playing('./countdown.wav')
        servo.throttle = DROP_THROTTLE
        self.rainbow = rainbow_lamp(range(0, 256, 2))
        self.rainbow_time = now + 0.1
        self.drop_finish_time = now + DROP_DURATION

    def exit(self, machine):
        State.exit(self, machine)
        now = time.monotonic()
        servo.throttle = 0.0
        stop_playing()
        machine.reset_fireworks()
        machine.firework_stop_time = now + FIREWORKS_DURATION

    def update(self, machine):
        if State.update(self, machine):
            now = time.monotonic()
            if now &gt;= self.drop_finish_time:
                machine.go_to_state('burst')
            if now &gt;= self.rainbow_time:
                next(self.rainbow)
                self.rainbow_time = now + 0.1
```

Burst and Shower States

These two go together to make the fireworks effect reminiscent of an explosion followed by a shower of twinkling sparks.

Burst is pretty simple. Nothing happens when it's entered, but when it exits, it sets the count for `Shower`. Each time through the loop its `update` method calls `burst` to advance the effect (same as the brute force implementation). When `burst` eventually returns `True`, it triggers a transition to the shower state.

Shower is somewhat similar. On exit it resets the fireworks system (which will generate a different random color). In `update`, `shower` is called to advance the effect. When `shower` eventually returns `True`, the machine transitions to one of two states: idle if it's time for the fireworks effect to finished, back to burst if it's not.

These states bounce back and forth until the fireworks has gone on long enough: explosion, shower, repeat.

```
class BurstState(State):

    @property
    def name(self):
        return 'burst'

    def enter(self, machine):
        State.enter(self, machine)

    def exit(self, machine):
        State.exit(self, machine)
        machine.shower_count = 0

    def update(self, machine):
        if State.update(self, machine):
            if burst(machine, time.monotonic()):
                machine.go_to_state('shower')


# Show a shower of sparks following an explosion

class ShowerState(State):

    @property
    def name(self):
        return 'shower'

    def enter(self, machine):
        State.enter(self, machine)

    def exit(self, machine):
        State.exit(self, machine)
        machine.reset_fireworks()

    def update(self, machine):
        if State.update(self, machine):
            if shower(machine, time.monotonic()):
                if now &gt;= machine.firework_stop_time:
                    machine.go_to_state('idle')
                else:
                    machine.go_to_state('burst')
```

Idle State

As we saw above, once the fireworks effect is finished, the machine moves into the `idle` state. This state does absolutely nothing beyond calling the base class' methods. State's `update` method will kick the machine into paused (and possibly raising after a second) when the switch is pressed.

```
class IdleState(State):

    @property
    def name(self):
        return 'idle'

    def enter(self, machine):
        State.enter(self, machine)

    def exit(self, machine):
        State.exit(self, machine)

    def update(self, machine):
        State.update(self, machine)
```

Raising State

This state is transitioned to when the switch is held for at least a second and remains the active state until it is released at which time the machine moves to the waiting state to start all over again.

The `enter` method resets the NeoPixel strip, stops the audio, and turns on the servo to pull the ball back up. On exit, the servo is stopped. The `update` method simply watches for the switch to be released, at which point the machine moves to the `waiting` state.

```
class RaisingState(State):

    @property
    def name(self):
        return 'raising'

    def enter(self, machine):
        State.enter(self, machine)
        strip.fill(0)
        strip.brightness = 1.0
        strip.show()
        if audio.playing:
            audio.stop()
        servo.throttle = RAISE_THROTTLE

    def exit(self, machine):
        State.exit(self, machine)
        servo.throttle = 0.0

    def update(self, machine):
        if State.update(self, machine):
            if switch.rose:
                machine.go_to_state('waiting')
```

The Main Loop

All that's left is to set up the machine and start looping.

```
machine = StateMachine()
machine.add_state(WaitingState())
machine.add_state(DroppingState())
machine.add_state(BurstState())
machine.add_state(ShowerState())
machine.add_state(IdleState())
machine.add_state(RaisingState())
machine.add_state(PausedState())

machine.go_to_state('waiting')

while True:
    switch.update()
    machine.update()
```

Discussion

By using classes to split the code into chunks specific to each state, and managing them in a separate machine class we've broken the code into small pieces, each of which is quite simple and understandable. Furthermore, separating out entry, exit, and update into separate methods cuts things into even smaller and more understandable pieces. What happens when a state is entered? Look in its class' entry method. This regularity of structure also helps make managing the states similarly simple and understandable.

It's also far easier to add states to the machine.