Erlang / Elixir meetup

Introduction to gen_statem

A gen_server with extra-power

to build Finite state machines

Code |> Tests |> Demo

@antoine_reyt

gen_statem vs gen_server ?

gen_statem is an erlang behaviour (just as gen_server) with additional features:

- separation of state and data
- better handling of timers/timeout
- ability to postpone events
- ability to insert Internal events

It was introduced in OTP 19, and deprecates gen_fsm.

It’s a behaviour designed to build finite state machine.

Plan

- Definition of Finite State machine
- List of features
- Learning the basic syntax
- Overview of the features with code & tests
- Interactive demos via Liveview:
  - **Light**: basic example to introduce the syntax/concepts.
  - **Payphone**: play with timeouts and absolute time.
  - **TrafficLight**: coordinate many FSMs.
- Questions

*A Finite-state Machine is an abstract machine that can be in **exactly one** of a finite number of states **at any given time**.*

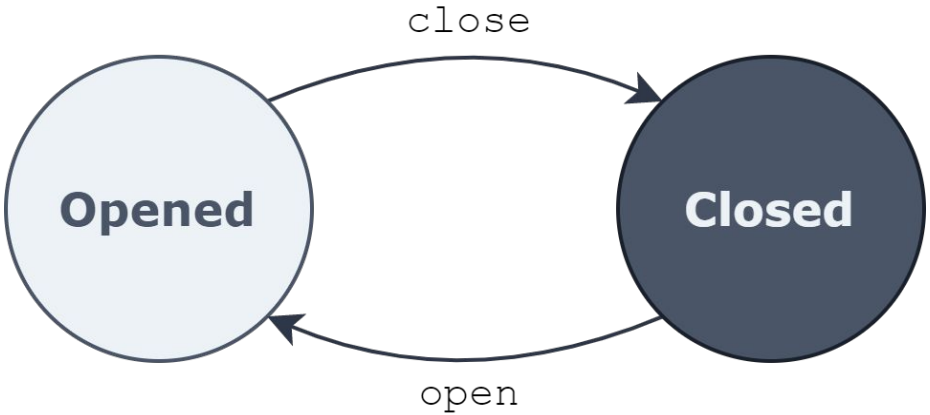
Wikipedia

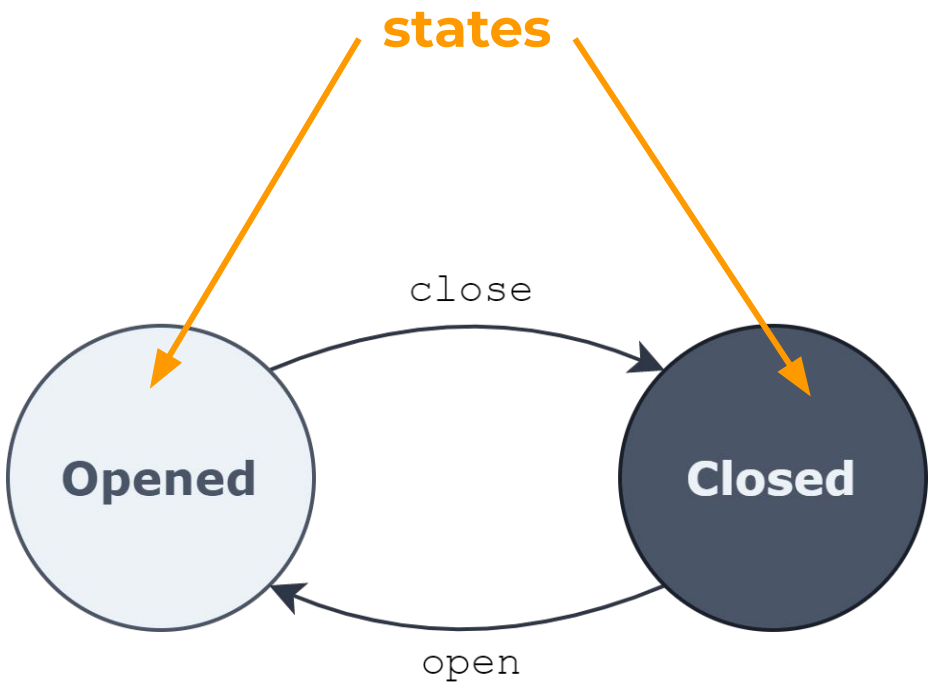
*A Finite-state Machine is an abstract machine that can be in **exactly one** of a finite number of states **at any given time**.*

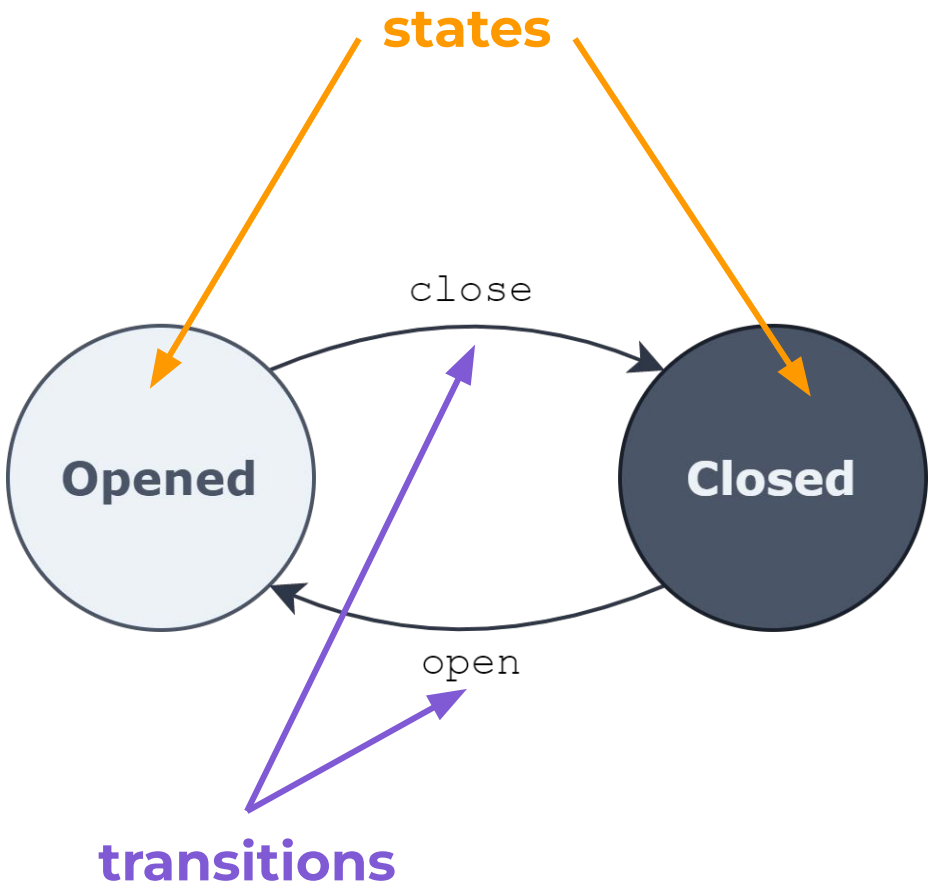
Wikipedia

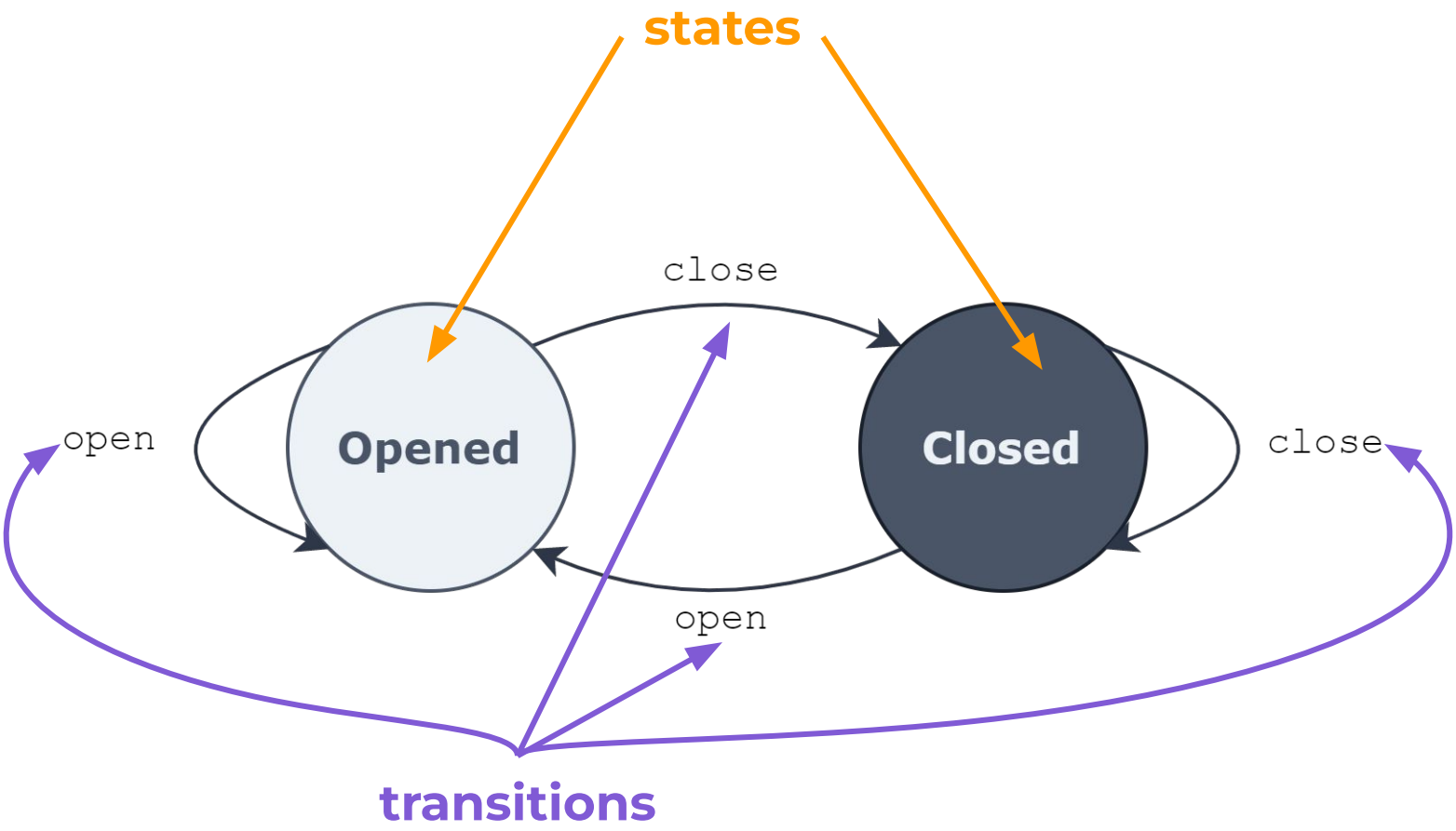
A Finite-state Machine is an abstract machine
that can be in exactly one of a finite number of
states at any given time.

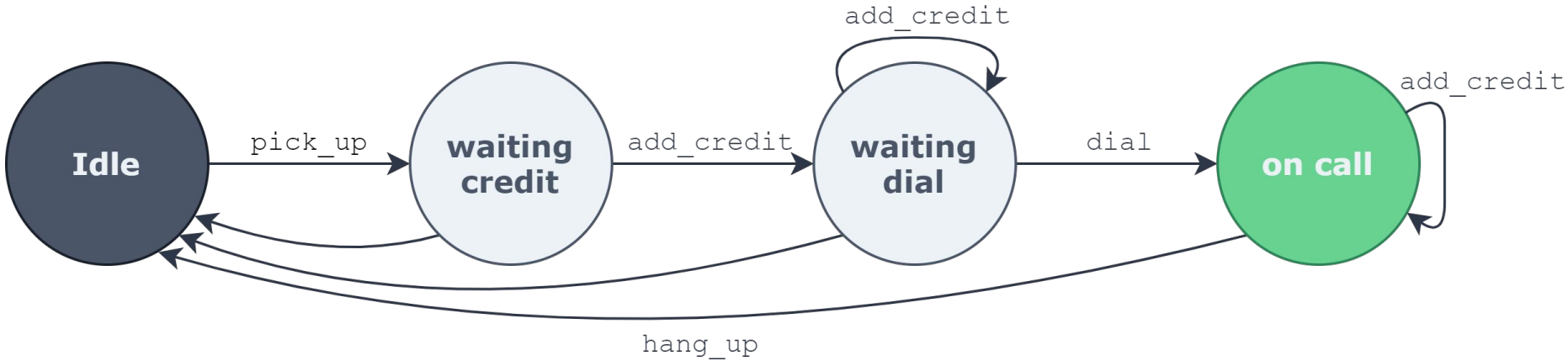
Let’s go graphical: use state-charts


close
Opened
Closed
open

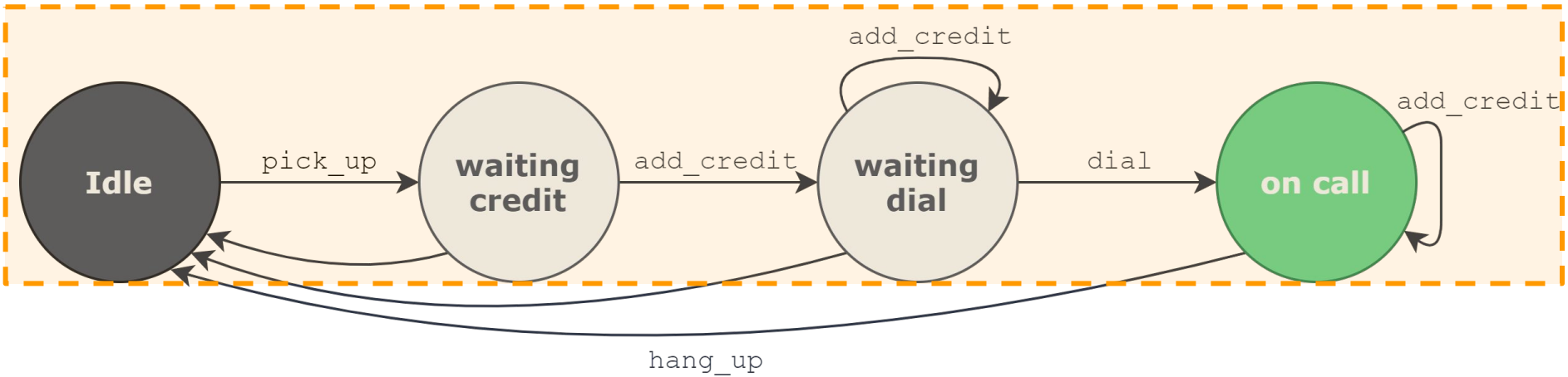

states
close
Opened
Closed
open

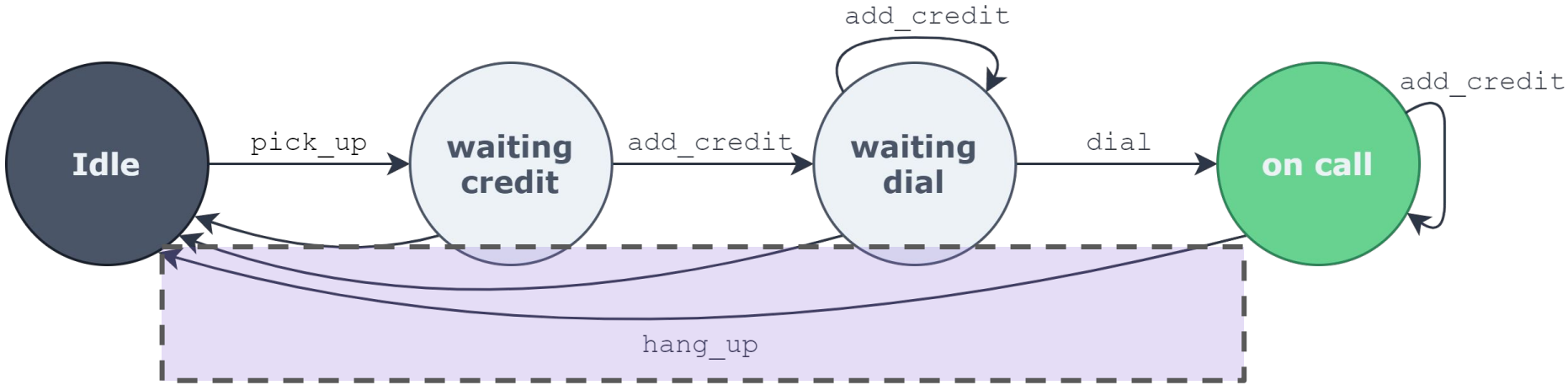

states
close
Opened
Closed
open
transitions


states
close
open
Opened
Closed
close
open
transitions


add_credit
Idle
pick_up
waiting credit
add_credit
waiting dial
dial
on call
add_credit
hang_up

Happy path


add_credit
add_credit
Idle
pick_up
waiting credit
add_credit
waiting dial
dial
on call
hang_up


add_credit
add_credit
Idle
pick_up
waiting credit
add_credit
waiting dial
dial
on call
hang_up

Exit path

Properties

- starts in its **initial state**
- can only be in a **fixed set of states**
- can only be in **one state at a time**
- changing from one state to another is called a **transition**

Summary

An FSM is composed of:

- States (including the initial state)
- Transitions

Benefits:

- Very simple to reason about using statechart
- Iterate with non technical people

Features

Features

- Timeouts
- state_enter
- Postpone
- Internal actions

Features - Timeouts

State Time-Out

There is one State Time-Out that is automatically cancelled by a state change.

Generic Time-Outs

There are any number of Generic Time-Outs differing by their Name. They have no automatic cancelling.

Event Time-Out

There is one Event Time-Out that is automatically cancelled by any event. Note that postponed and inserted events cancel this time-out just as external events.

Features - state_enter

When enabled, it’s called each time the state changes.

Depending on how your state machine is specified, this can be a very useful feature: factorize code.

You can also repeat a state: when entering the same state, you can explicitly say that you want to trigger the state_enter function again .

Features - postpone

If you decide to postpone an event, then it will not be tried again while in the current state.

After a state change, the queue restarts with the previously postponed events, and then continues with the new external event.

Features - internal_event

Internal events can only be generated by the state machine itself.

You can schedule many internal events for one incoming external event.

They are inserted in the queue as the next event to process before any already queued events.

Features - summary

The FSM reacts to:

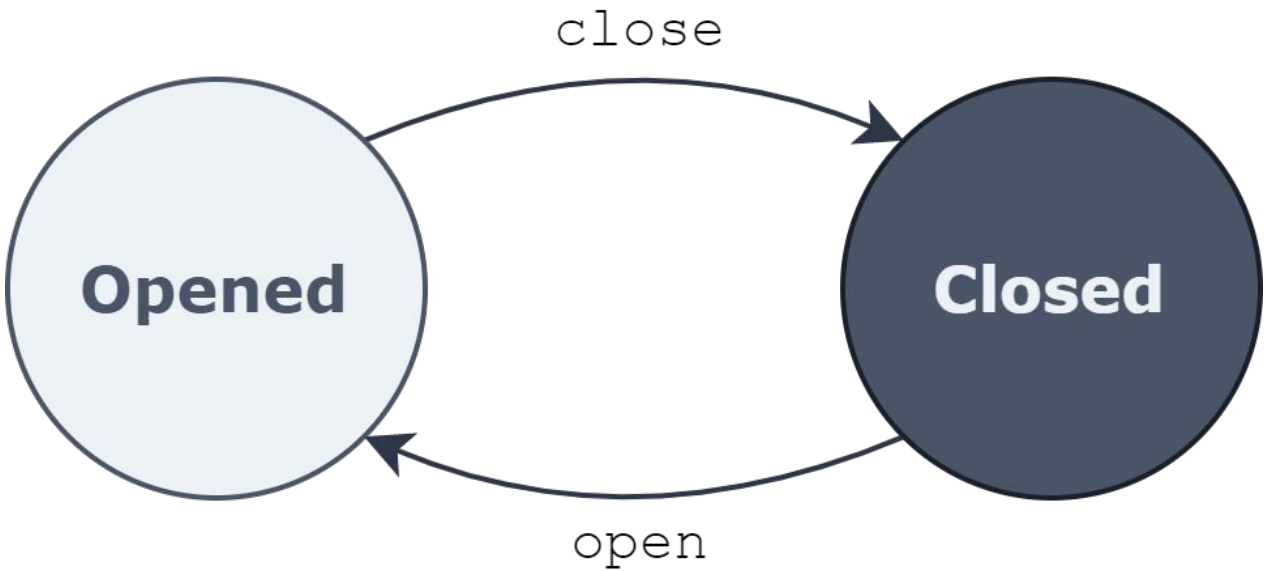
- External events
- Internal events
- Timeouts

You can **postpone** events.

Message **priority:**

1. Internal
2. Postponed
3. External

Basics


close
Opened
Closed
open

Base

```elixir
defmodule DoorFsm.StateFunction.Cast do
  @behaviour :gen_statem

  @type state :: :opened | :closed
  @initial_state :closed

  def start_link() do
    :gen_statem.start_link(__MODULE__, nil, [])
  end

  @impl :gen_statem
  def init(_) do
    {:ok, @initial_state, :no_real_data}
  end

  @impl :gen_statem
  def callback_mode() do
    :state_functions
  end
end
```

Public API

```elixir
defmodule DoorFsm.StateFunction.Cast do
  # public api

  def open(pid) do
    :gen_statem.cast(pid, :open)
  end
  def close(pid) do
    :gen_statem.cast(pid, :close)
  end
end
```

:state_function - Callbacks

```elixir
defmodule DoorFsm.StateFunction.Cast do
  # gen_statem callbacks

  # closed state
  def closed(:cast, :close, _data) do
    :keep_state_and_data
  end
  def closed(:cast, :open, data) do
    {:next_state, :opened, data}
  end

  # opened state
  def opened(:cast, :open, _data) do
    :keep_state_and_data
  end
  def opened(:cast, :close, data) do
    {:next_state, :closed, data}
  end
end
```

:handle_event_function - Callbacks

```elixir
defmodule DoorFsm.HandleEventFunction.Cast do
  # gen_statem callbacks

  # closed state
  def handle_event(:cast, :close, :closed, _data) do
    :keep_state_and_data
  end
  def handle_event(:cast, :open, :closed, data) do
    {:next_state, :opened, data}
  end

  # opened state
  def handle_event(:cast, :open, :opened, _data) do
    :keep_state_and_data
  end
  def handle_event(:cast, :close, :opened, data) do
    {:next_state, :closed, data}
  end
end
```

10 fsm.ex

```
...  @@ -15,16 +15,16 @@ defmodule DoorFsm do
15                                          15
16      # gen_statem callbacks              16      # gen_statem callbacks
17                                          17
18  -   def closed(:cast, :close, _data) do 18  +   def handle_event(:cast, :close, :closed, _data) do
19        :keep_state_and_data              19        :keep_state_and_data
20      end                                 20      end
21  -   def closed(:cast, :open, data) do   21  +   def handle_event(:cast, :open, :closed, data) do
22        {:next_state, :opened, data}      22        {:next_state, :opened, data}
23      end                                 23      end
24  -   def opened(:cast, :open, _data) do  24  +   def handle_event(:cast, :open, :opened, _data) do
25        :keep_state_and_data              25        :keep_state_and_data
26      end                                 26      end
27  -   def opened(:cast, :close, data) do  27  +   def handle_event(:cast, :close, :opened, data) do
28        {:next_state, :closed, data}      28        {:next_state, :closed, data}
29      end                                 29      end
30                                          30
...  @@ -39,6 +39,6 @@ defmodule DoorFsm do
39                                          39
40      @impl :gen_statem                   40      @impl :gen_statem
41      def callback_mode() do              41      def callback_mode() do
42  -     :state_functions                  42  +     :handle_event_function
43      end                                 43      end
44    end                                   44    end
```

Tests

:sys.get_state | :sys.trace - Tests

```elixir
defmodule Fsm.HandleEventFunction.CastTest do
  use ExUnit.Case

  alias DoorFsm.HandleEventFunction.Cast, as: Fsm

  setup %{} do
    {:ok, pid} = Fsm.start_link()
    # :sys.trace(pid, true)
    [pid: pid]
  end

  test "default state is closed", %{pid: pid}  do
    assert {:closed, _} = :sys.get_state(pid)
  end

  describe "Closed door" do
    setup [:closed_door]

    test "[closed] -(open)-> [opened]", %{pid: pid} do
      Fsm.open(pid)
      assert {:opened, _} = :sys.get_state(pid)
    end
    test "[closed] -(close)-> [closed]", %{pid: pid} do
      Fsm.close(pid)
      assert {:closed, _} = :sys.get_state(pid)
    end
  end
end
```

Demos

Light

Simple FSM to grasp the basic concepts and the syntax.

#state_function

#state_timeout

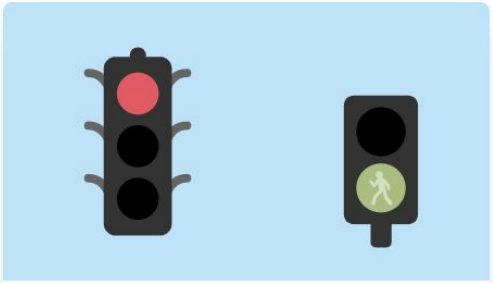
#state_enter

Payphone

Introduce the handle_function syntax and learn about generic_timeout using absolute time.

#handle_function

#generic_timeout



Traffic light

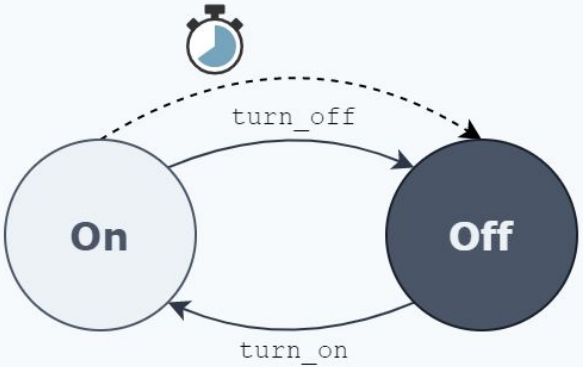
Synchronize two FSMs and discover the power of postpone and internal_event when used together.

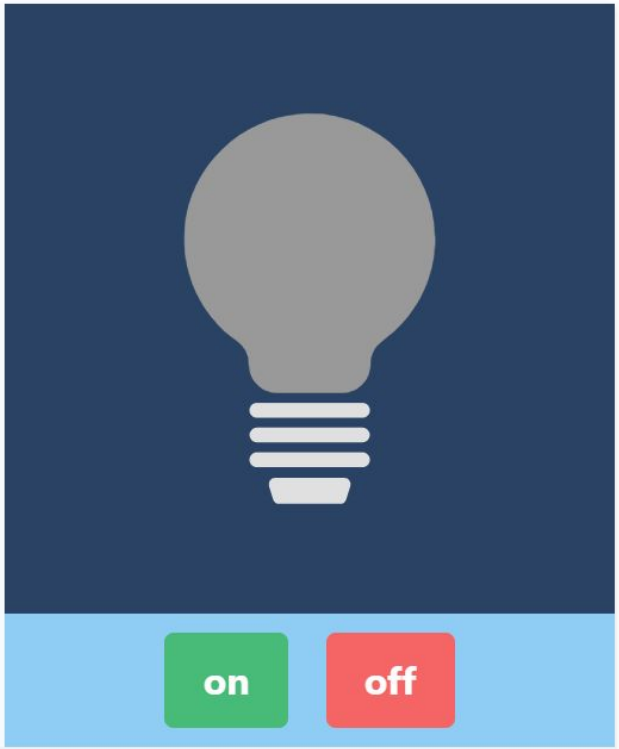
#postpone

#internal_event

state_timeout

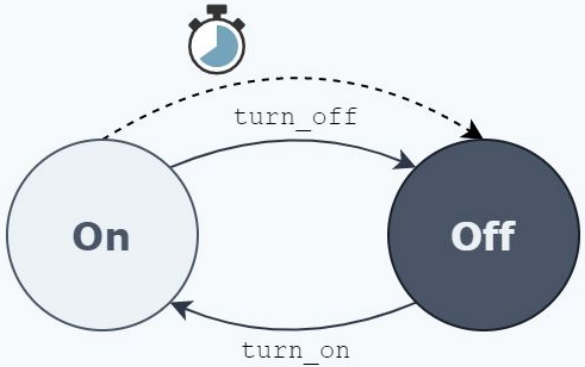
Light FSM





pid: #PID<0.1361.0>
state: off

Light FSM

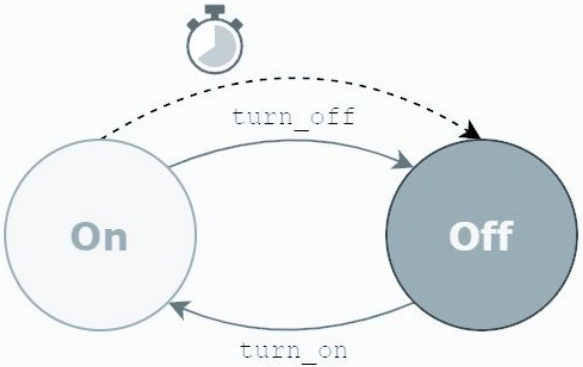



```
pid: #PID<0.1361.0>
state: off
```

Rules:

- Light is turned off by default
- When the “on” button is pressed, light is turned on
- When the “off” button is pressed, light is turned off
- After a given time, the light turns off itself

Light FSM




```
pid: #PID<0.1361.0>
state: off
```

Basic transitions

state_timeout

Rules:

- Light is turned off by default
- When the “on” button is pressed, light is turned on
- When the “off” button is pressed, light is turned off

- After a given time, the light turns off itself

:state_timeout

```elixir
defmodule DoorFsm.StateFunction.Cast do
  # gen_statem callbacks

  def closed(:cast, :open, data) do
    actions = [{:state_timeout, 200, :open_ttl}]
    {:next_state, :opened, data, actions}
  end

  def opened(:state_timeout, _, data) do
    {:next_state, :closed, data}
  end
end
```

6 state_timeout.ex

```
...  @@ -19,14 +19,18 @@ defmodule DoorFsm do
19       :keep_state_and_data                                  19       :keep_state_and_data
20     end                                                     20     end
21     def closed(:cast, :open, data) do                       21     def closed(:cast, :open, data) do
22 -     {:next_state, :opened, data}                          22 +     actions = [{:state_timeout, 200, :open_ttl}]
                                                               23 +     {:next_state, :opened, data, actions}
23     end                                                     24     end
24     def opened(:cast, :open, _data) do                      25     def opened(:cast, :open, _data) do
25       :keep_state_and_data                                  26       :keep_state_and_data
26     end                                                     27     end
27     def opened(:cast, :close, data) do                      28     def opened(:cast, :close, data) do
28       {:next_state, :closed, data}                          29       {:next_state, :closed, data}
29     end                                                     30     end
                                                               31 +   def opened(:state_timeout, _, data) do
                                                               32 +     {:next_state, :closed, data}
                                                               33 +   end
30                                                             34
31     def start_link() do                                     35     def start_link() do
32       :gen_statem.start_link(__MODULE__, nil, [])           36       :gen_statem.start_link(__MODULE__, nil, [])
...
```

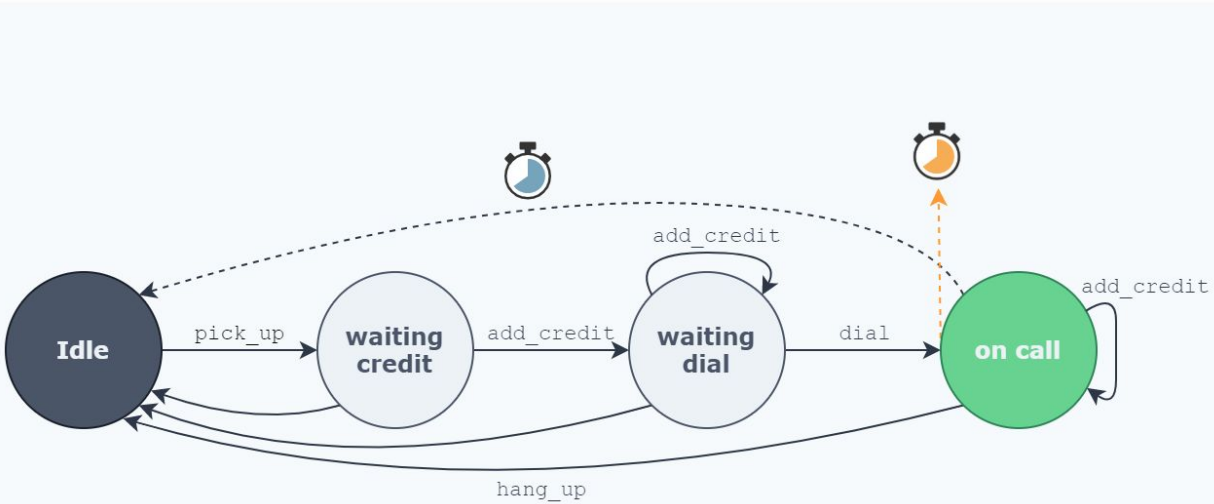
Tests

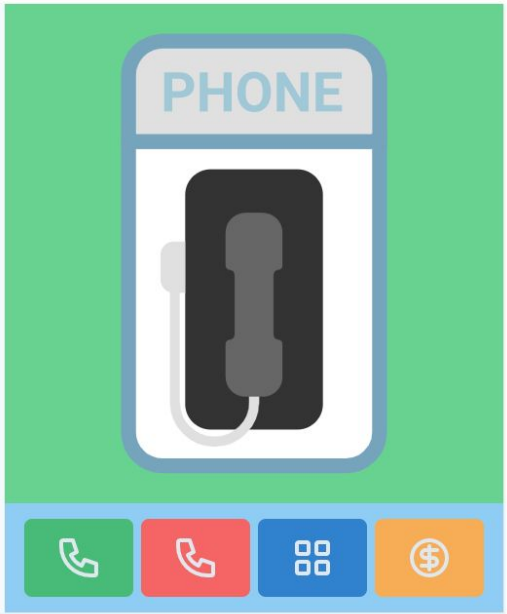
:timer.sleep()

```elixir
defmodule Light.FsmTest do
  use ExUnit.Case

  @on_ttl Application.get_env(:fsmlive, :light)[:durations][:on_ttl]

  describe "on state" do
    setup :on

    test "Auto off after a given time", %{pid: pid} do
      :timer.sleep(@on_ttl)
      assert {:off, _} = :sys.get_state(pid)
    end
  end
end
```

generic_timeout

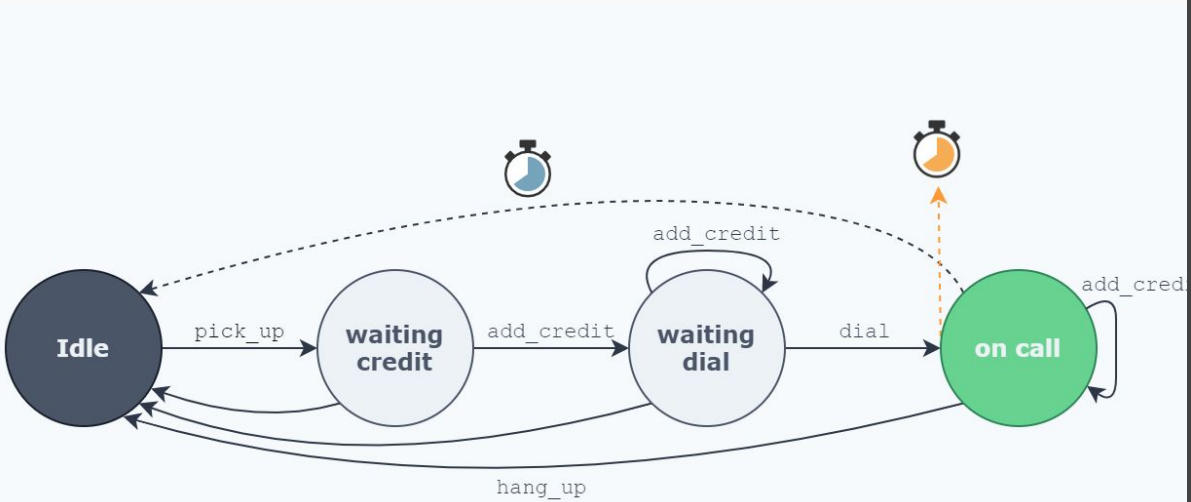
Payphone FSM





pid: #PID<0.675.0>
state: idle

Payphone FSM



pid: #PID<0.675.0>
state: idle

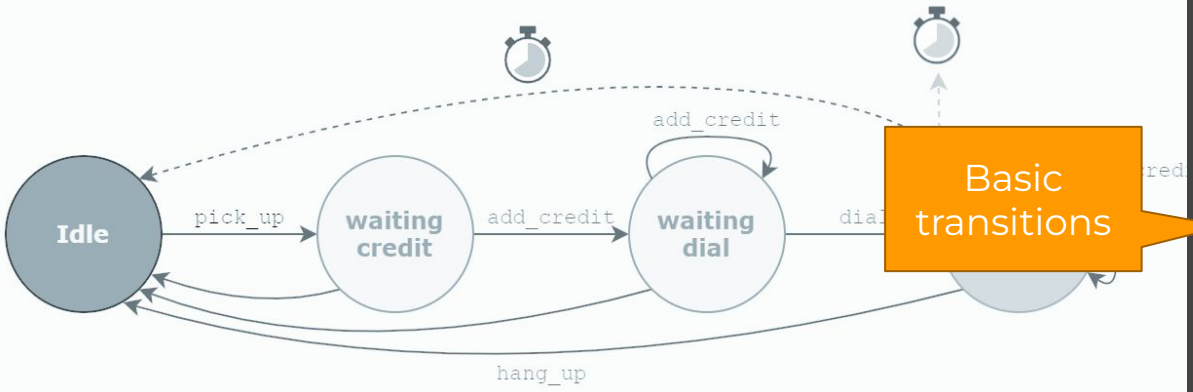
Sequence:

- **pick up** the phone
- add some **credit**
- **dial** a number
- speak
- **hang up** the phone

Rules:

- User can add credit when both dialing and calling.
- User can hang up at any time.
- When user runs out of credit, the phone hangs up itself.
- When the credit is low, the user gets a voice notification

Payphone FSM




```
pid: #PID<0.675.0>
state: idle
```

Basic transitions

state_timeout

?

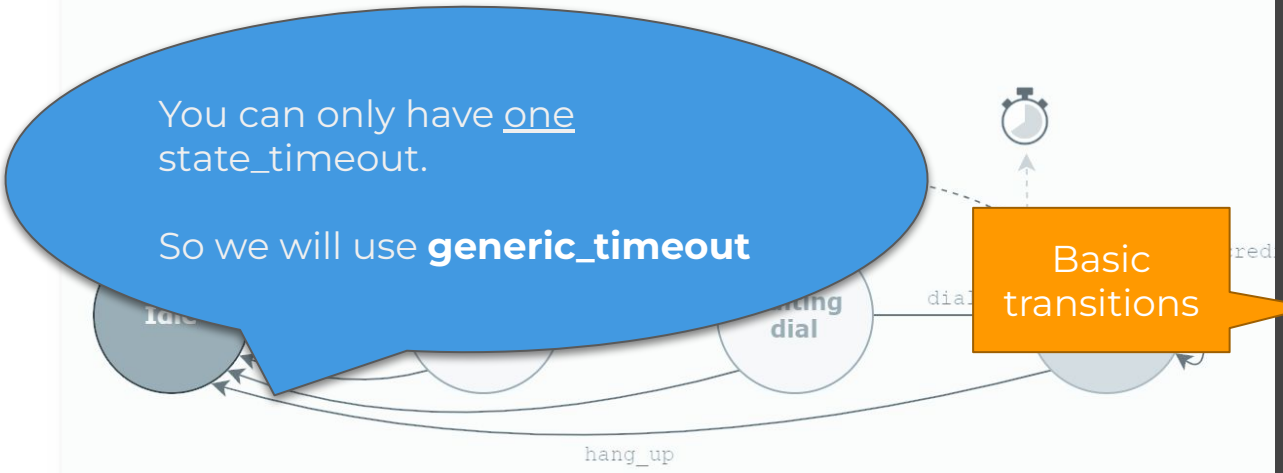
Sequence:

- **pick up** the phone
- add some **credit**
- **dial** a number
- speak
- **hang up** the phone

Rules:

- User can add credit when both dialing and calling.
- User can hang up at any time.
- When user runs out of credit, the phone hangs up itself.
- When the credit is low, the user gets a voice notification

Payphone FSM




```
pid: #PID<0.675.0>
state: idle
```

Basic transitions

state_timeout

?

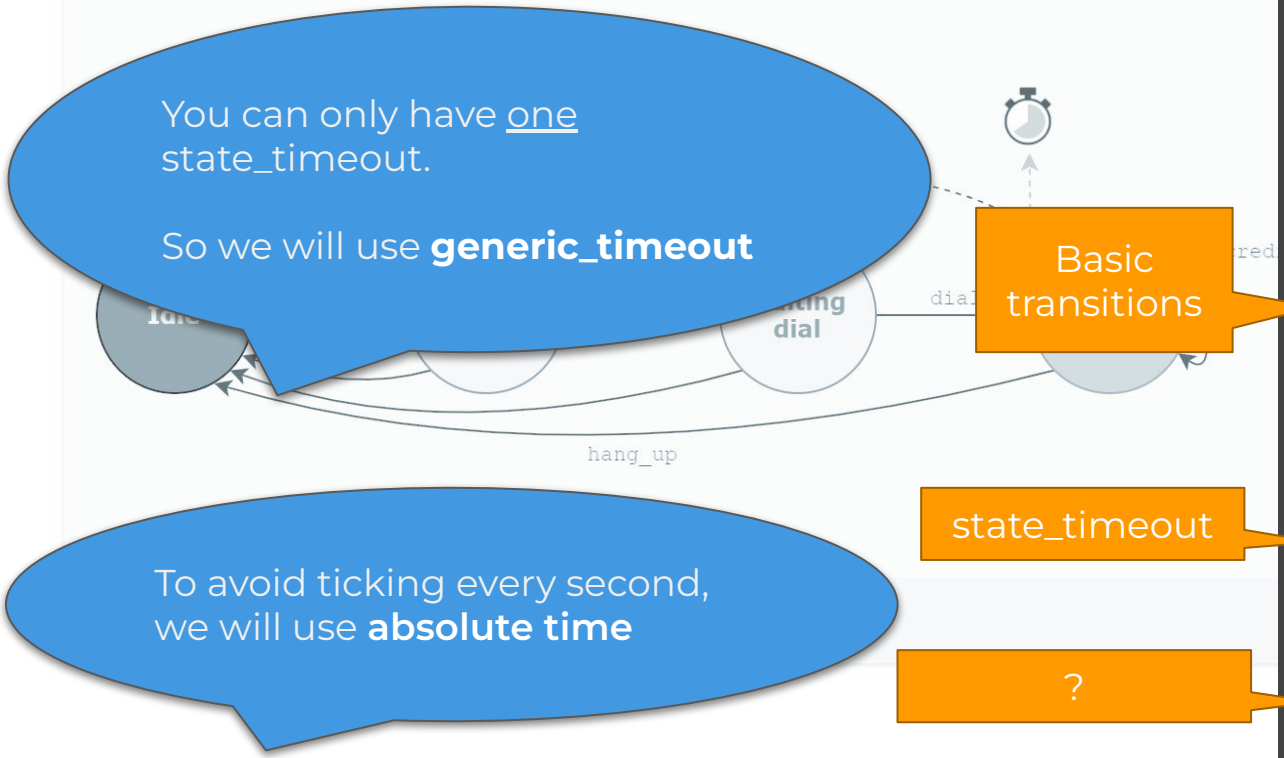
Sequence:

- **pick up** the phone
- add some **credit**
- **dial** a number
- speak
- **hang up** the phone

Rules:

- User can add credit when both dialing and calling.
- User can hang up at any time.
- When user runs out of credit, the phone hangs up itself.
- When the credit is low, the user gets a voice notification

Payphone FSM



Sequence:

- **pick up** the phone
- add some **credit**
- **dial** a number
- speak
- **hang up** the phone

Rules:

- User can add credit when both dialing and calling.
- User can hang up at any time.
- When user runs out of credit, the phone hangs up itself.
- When the credit is low, the user gets a voice notification

```elixir
defmodule Payphone.Fsm do
  def handle_event({:call, form}, :dial, :waiting_dial, data) do
    new_data = %{data | call_started_at: :erlang.monotonic_time(:millisecond)}

    actions = [
      {:reply, form, :ok},
      no_more_credit_timeout(new_data),
      warning_low_credit_timeout(new_data)
    ]

    {:next_state, :on_call, new_data, actions}
  end

  def handle_event({:timeout, :warning_low_credit}, nil, :on_call, data) do
    @hardware.exec(data.hardware_pid, :screen, :warning_low_credit)
    :keep_state_and_data
  end

  defp warning_low_credit_timeout(data) do
    end_call = data.call_started_at + credit_to_time(data.credit)
    ts = end_call - data.warning_threshold
    {{:timeout, :warning_low_credit}, ts, nil, {:abs, true}}
  end

  defp reset_warning_low_credit_timeout() do
    {{:timeout, :warning_low_credit}, :infinity, nil, {:abs, true}}
  end
end
```

Tests

assert_receive / refute_receive

```elixir
defmodule Payphone.FsmTest do
  use ExUnit.Case

  alias Payphone.Fsm, as: Fsm

  @credit_duration Application.get_env(:fsmlive, :payphone)[:credit_duration]
  @warning_threshold Application.get_env(:fsmlive, :payphone)[:warning_threshold]

  describe "on_call state" do
    test "add_credit delays the low credit warning signal" do
      {:ok, pid} = Fsm.start_link(self())
      Fsm.pick_up(pid)

      Fsm.add_credit(pid, 2)
      Fsm.add_credit(pid, trunc(@warning_threshold / @credit_duration))
      # now we have a total of 2 + warning_threshold credits

      # After one credit duration, I add one more credit
      spawn(fn ->
        :timer.sleep(@credit_duration)
        Fsm.add_credit(pid, 1)
      end)

      Fsm.dial(pid)

      # At the end I had been able to call for 3 credit time without any warning signal
      refute_receive {:hardware_mock, ^pid, :screen, :warning_low_credit}, 3 * @credit_duration
      assert_receive {:hardware_mock, ^pid, :screen, :warning_low_credit}, 50
    end
  end
```

state_enter

:handle_event_function - State enter

```elixir
defmodule DoorFsm do
  # ...
  @impl :gen_statem
  def handle_event(:enter, _oldState, state, data) do
     # ...
    :keep_state_and_data
  end

  @impl :gen_statem
  def callback_mode() do
    [:handle_event_function, :state_enter]
  end
end
```

:state_function - State enter

```elixir
defmodule Light.Fsm do
  # ...
  # state off
  # ...
  def off(:enter, _oldState, data) do
    # ...
    :keep_state_and_data
  end

  # state on
  # ...
  def on(:enter, _oldState, data) do
    # ...
    :keep_state_and_data
  end

  @impl :gen_statem
  def callback_mode() do
    [:state_functions, :state_enter]
  end
end
```

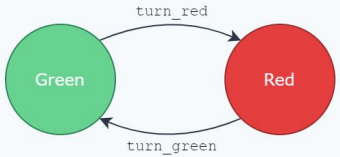
Internal_event

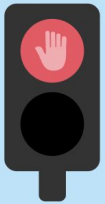
&

postpone

Synchronize

TrafficLight.**People**Fsm

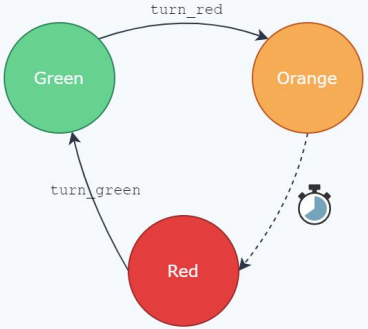


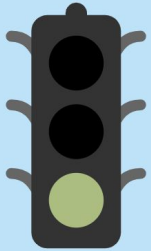


turn_green turn_red

pid: #PID<0.1437.0>
state: red

TrafficLight.**Car**Fsm

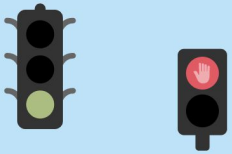




turn_green turn_red

pid: #PID<0.1436.0>
state: green

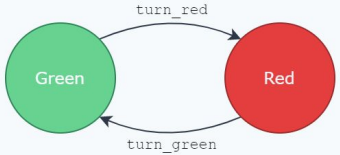
TrafficLight.**Manager**Fsm

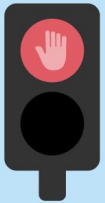


cross

```
car_pid: #PID<0.1438.0>
people_pid: #PID<0.1439.0>
manager_pid: #PID<0.1440.0>
```

TrafficLight.**People**Fsm

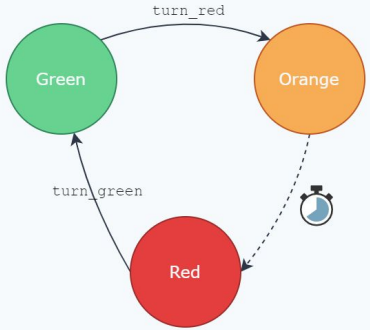


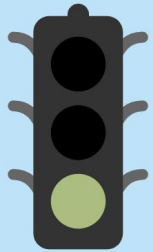


turn_green turn_red

```
pid: #PID<0.1437.0>
state: red
```

TrafficLight.**Car**Fsm

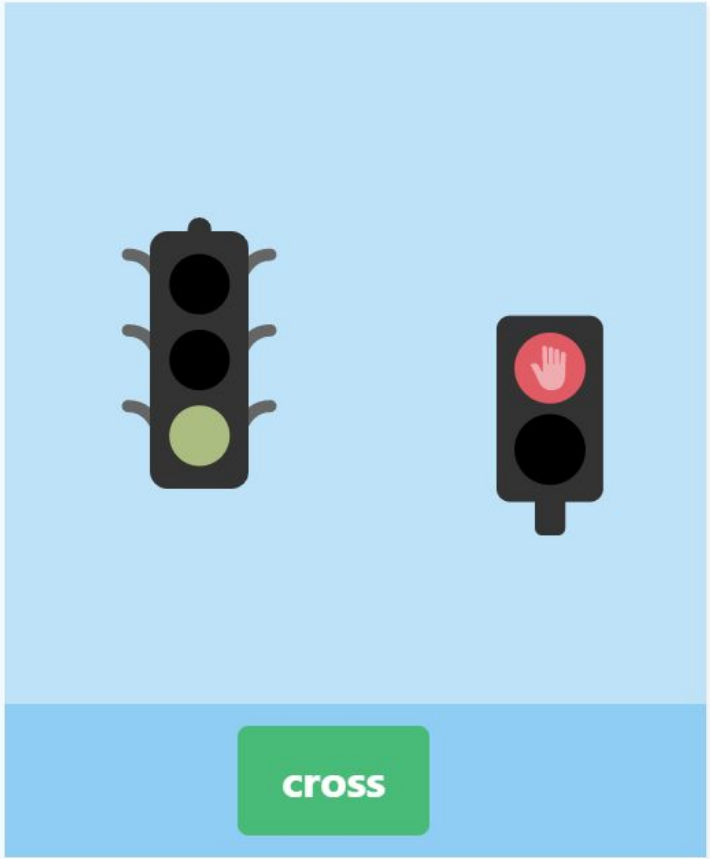




turn_green turn_red

```
pid: #PID<0.1436.0>
state: green
```

?



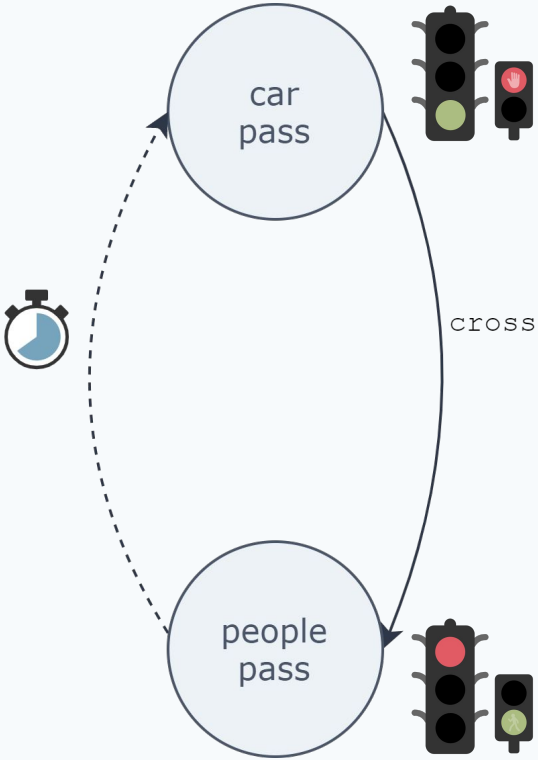
car_pid: #PID<0.1438.0>
people_pid: #PID<0.1439.0>
manager_pid: #PID<0.1440.0>

```
car_pid: #PID<0.1438.0>
people_pid: #PID<0.1439.0>
manager_pid: #PID<0.1440.0>
```

Goal:
Crossing the road.

Sequence:

- By default, the traffic lights let the cars pass.
- Occasionally, people ask to cross the road using a button.
- Traffic light color changes to let the people pass.
- After a given time, traffic lights let the cars pass again

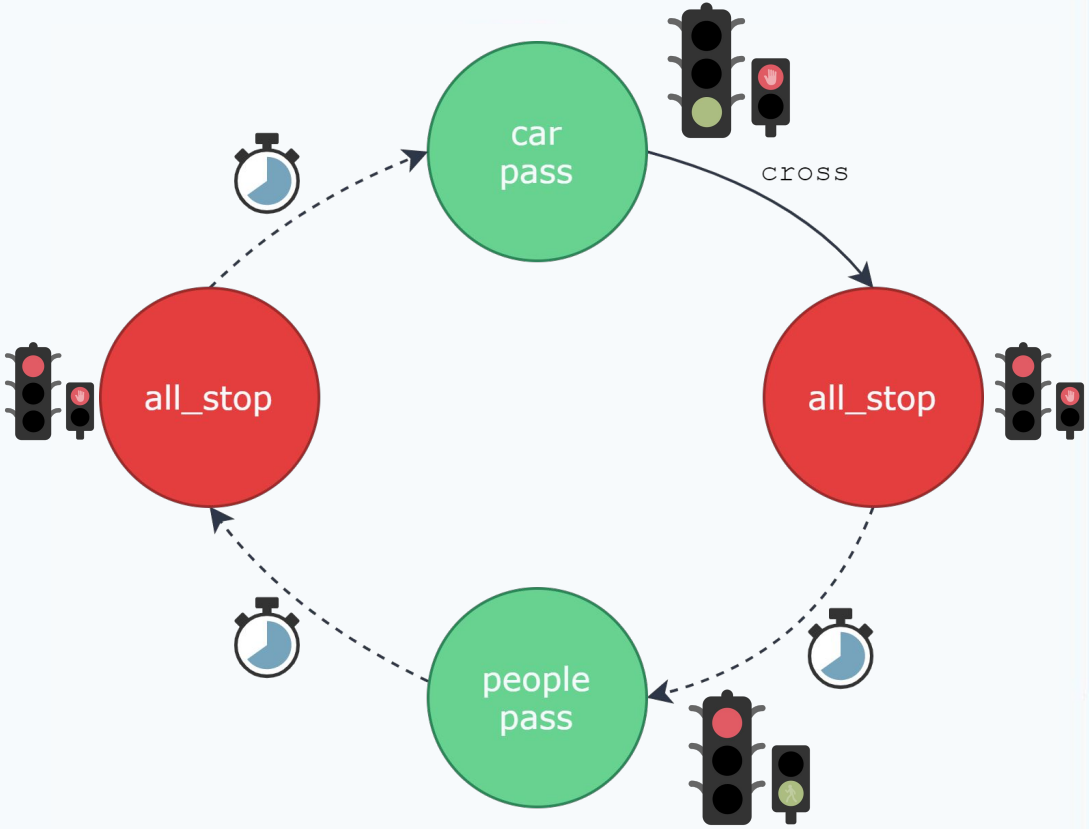



```
car_pid: #PID<0.1438.0>
people_pid: #PID<0.1439.0>
manager_pid: #PID<0.1440.0>
```

Goal:
Crossing the road.

Sequence:

- By default, the traffic lights let the cars pass.
- Occasionally, people ask to cross the road using a button.
- Traffic light color changes to let the people pass.
- After a given time, traffic lights let the cars pass again



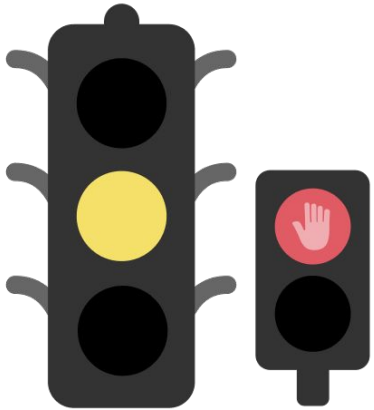

```
car_pid: #PID<0.1438.0>
people_pid: #PID<0.1439.0>
manager_pid: #PID<0.1440.0>
```

Goal:
Crossing the road.

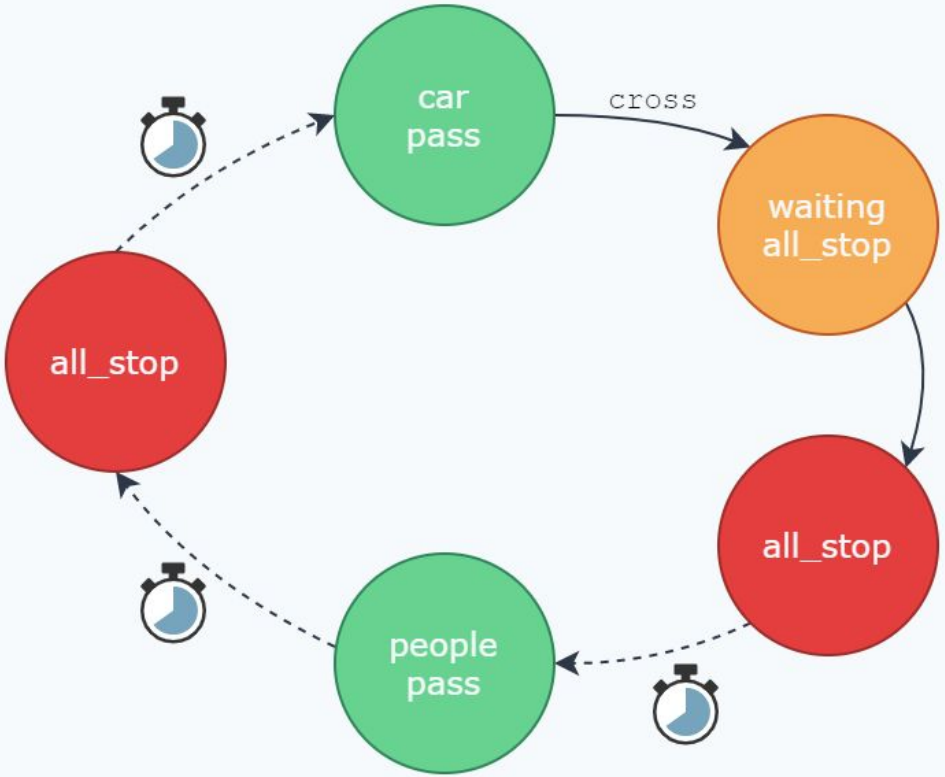
Sequence:

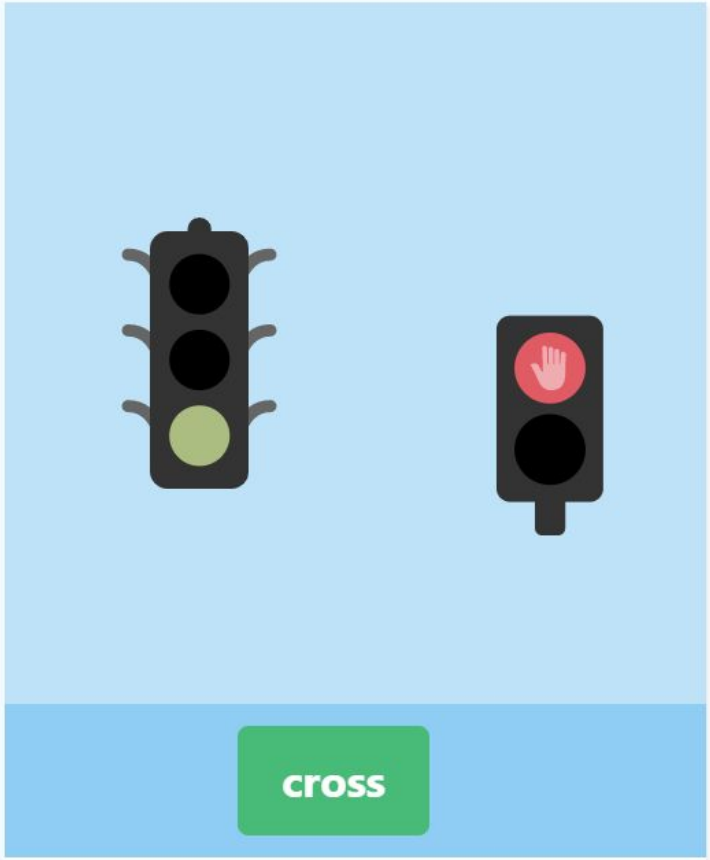
- By default, the traffic lights let the cars pass.
- Occasionally, people ask to cross the road using a button.
- Traffic light color changes to let the people pass.
- After a given time, traffic lights let the cars pass again

Oops! we forgot this guy



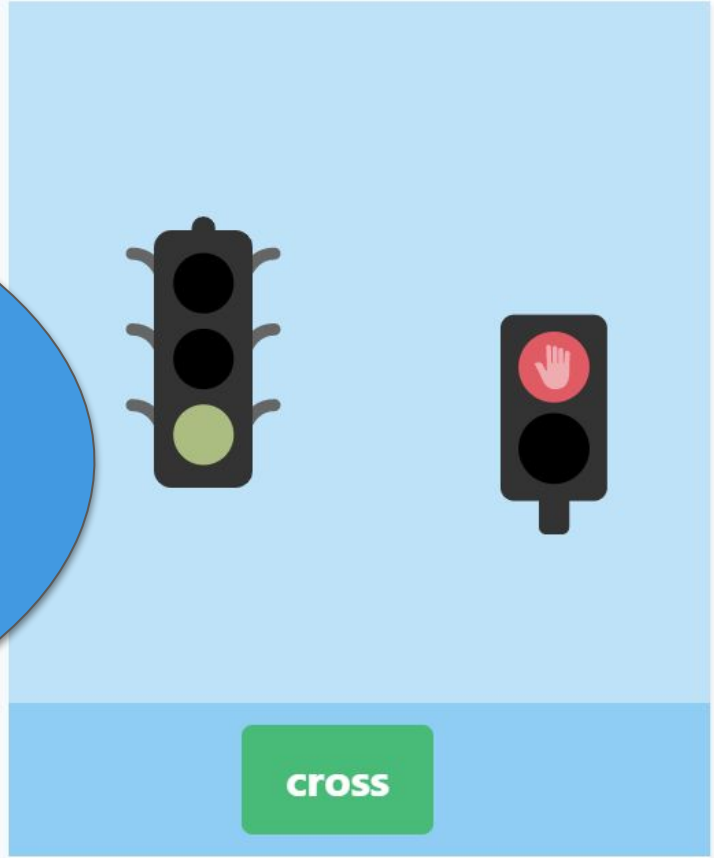
car_pid: #PID<0.1438.0>
people_pid: #PID<0.1439.0>
manager_pid: #PID<0.1440.0>





car_pid: #PID<0.1438.0>
people_pid: #PID<0.1439.0>
manager_pid: #PID<0.1440.0>

car_state: green
people_state: red
manager_state: car_pass



car_pid: #PID<0.1438.0>
people_pid: #PID<0.1439.0>
manager_pid: #PID<0.1440.0>

car_state: green
people_state: red
manager_state: car_pass

ManagerFsm call FsmCar.turn_red

```elixir
defmodule TrafficLight.ManagerFsm do
  # public api

  def cross(pid) do
    :gen_statem.cast(pid, :cross)
  end

  # state car_pass
  @impl :gen_statem
  def handle_event(:cast, :cross, :car_pass, data) do
    :ok = CarFsm.turn_red(data.car_pid)
    {:next_state, :waiting_all_stop, data}
  end

  # state waiting_all_stop
  def handle_event(:info, {:callback, _car_fsm_pid, :red}, :waiting_all_stop, data) do
    actions = [{:state_timeout, @durations[:all_stop], :before_people_pass}]
    {:next_state, :all_stop, data, actions}
  end
end
```

CarFsm setup an internal_action

```elixir
defmodule TrafficLight.CarFsm do
  def handle_event({:call, {caller_pid, _ref} = from}, :turn_red, :green, data) do
    duration = Application.get_env(:fsmlive, :traffic_light)[:durations][:orange]

    actions = [
      {:reply, from, :ok},
      {:state_timeout, duration, :orange_ttl},
      {:next_event, :internal, {:callback, caller_pid, in_state: :red}}
    ]

    {:next_state, :orange, data, actions}
  end
end
```

CarFsm postpone the previous internal_action (callback) until it's in the red state

```elixir
defmodule TrafficLight.CarFsm do
  # internal events: callback
  def handle_event(:internal, {:callback, caller_pid, in_state: state}, state, _data) do
    send(caller_pid, {:callback, self(), state})
    :keep_state_and_data
  end

  def handle_event(:internal, {:callback, _caller_pid, _}, _state, _data) do
    {:keep_state_and_data, :postpone}
  end
end
```

```elixir
defmodule TrafficLight.CarFsm do
  def handle_event({:call, {caller_pid, _ref} = from}, :turn_red, :green, data) do
    duration = Application.get_env(:fsmlive, :traffic_light)[:durations][:orange]

    actions = [
      {:reply, from, :ok},
      {:state_timeout, duration, :orange_ttl},
      {:next_event, :internal, {:callback, caller_pid, in_state: :red}}
    ]

    {:next_state, :orange, data, actions}
  end

  # internal events: callback
  def handle_event(:internal, {:callback, caller_pid, in_state: state}, state, _data) do
    send(caller_pid, {:callback, self(), state})
    :keep_state_and_data
  end

  def handle_event(:internal, {:callback, _caller_pid, _}, _state, _data) do
    {:keep_state_and_data, :postpone}
  end
end
```

ManagerFsm receive the callback from CarFsm

```elixir
defmodule TrafficLight.ManagerFsm do
  # public api

  def cross(pid) do
    :gen_statem.cast(pid, :cross)
  end

  # state car_pass
  @impl :gen_statem
  def handle_event(:cast, :cross, :car_pass, data) do
    :ok = CarFsm.turn_red(data.car_pid)
    {:next_state, :waiting_all_stop, data}
  end

  # state waiting_all_stop
  def handle_event(:info, {:callback, _car_fsm_pid, :red}, :waiting_all_stop, data) do
    actions = [{:state_timeout, @durations[:all_stop], :before_people_pass}]
    {:next_state, :all_stop, data, actions}
  end
end
```

Links

- https://potatosalad.io/2017/10/13/time-out-elixir-state-machines-versus-servers
- https://andrealeopardi.com/posts/connection-managers-with-gen_statem/
- https://hex.pm/packages/gen_state_machine
- https://ericent.in/elixir/erlang/gen_statem/gen_state_machine/2016/07/27/stately-machines.html

source code

https://github.com/antoinereyt/gen_statem_meetup

Questions ?