Extracted from:

Build a Binary Clock with Elixir and Nerves

Use Layering to Produce Better Embedded Systems

The Pragmatic Bookshelf

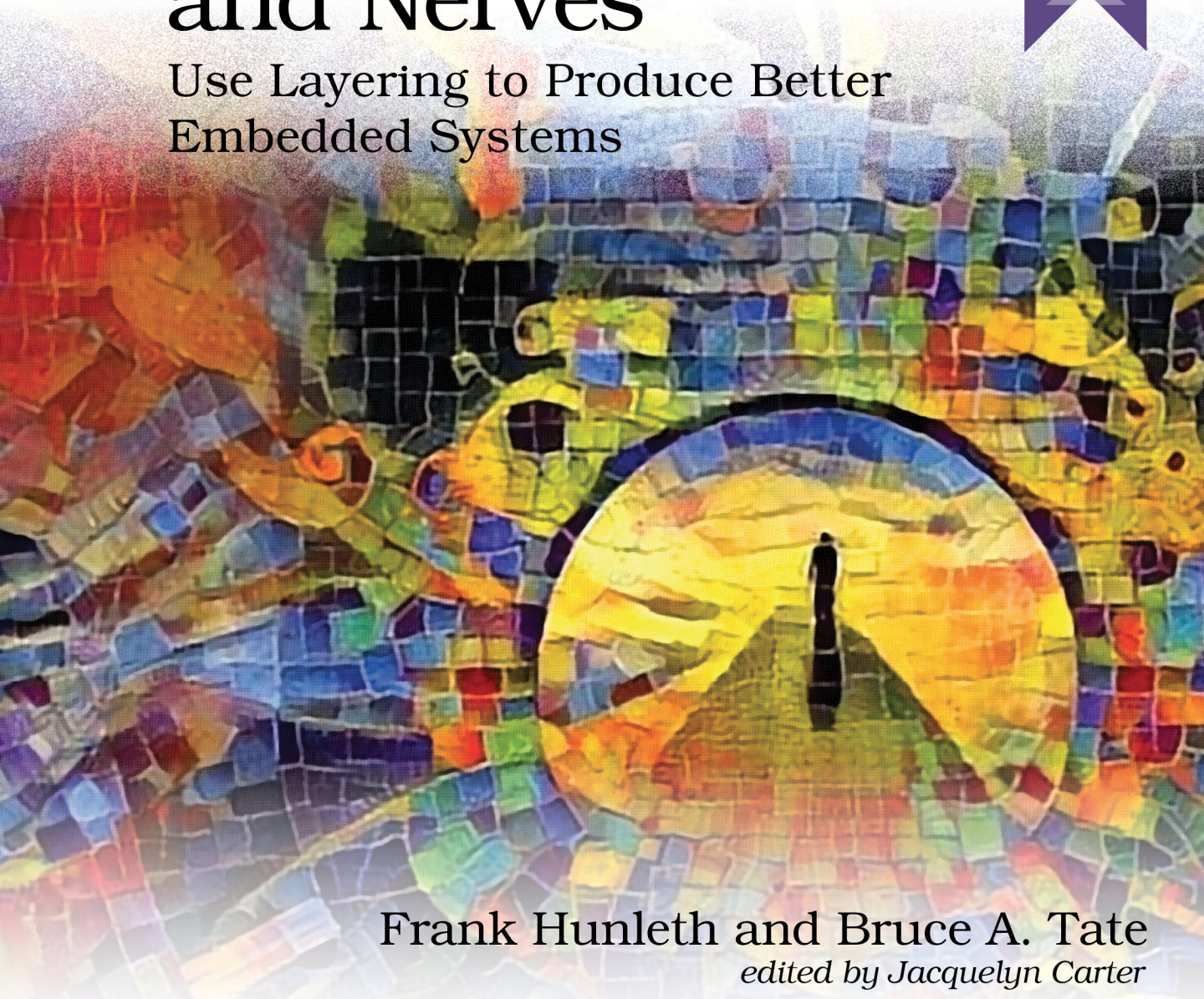
Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

Build a Binary Clock with Elixir and Nerves

Use Layering to Produce Better Embedded Systems



Frank Hunleth and Bruce A. Tate

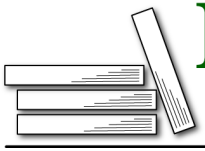
edited by Jacquelyn Carter

Build a Binary Clock with Elixir and Nerves

Use Layering to Produce Better Embedded Systems

Frank Hunleth
Bruce A. Tate

The Pragmatic Bookshelf
Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Jacquelyn Carter
Copy Editor: L. Sakhi MacMillan
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-923-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2022

Build a Blinker Boundary

This section with an awesomely alliterative title will build a boundary on top of our LED layer to blink an LED one time, and then multiple times. Rather than put all of the functionality of our program in one place, we're going to separate the parts that know how to communicate with hardware from the parts that know how to blink. You probably won't be surprised to learn that in this section we're going to use a GenServer.

Bruce says:

Did You Try Turning It Off and On Again?

Customer support representatives are famous for asking users to turn appliances or devices off and on again. Whether you're troubleshooting a cable device or a new electric car, you have likely encountered these instructions.

Elixir's OTP is a library for running generic services in a way that's concurent, distributed, and resilient. Elixir is famous for reliability because of OTP. When services experience problems, we let them crash and start them in a fresh starting state. OTP is Elixir's way of asking, "Did you try turning it off and on again?"

The lib/blinker/server.ex file will have the service boundary, and it will look like this:

```
defmodule Blinker.Server do
  alias Blinker.LED
  defstruct [:led, :on, :ticker]
  use GenServer
  @pin 26
```

At the top of each file is a bit of ceremony, but this code is doing a lot of work. We alias our LED and define the structure that will make up the state of the GenServer. The led is the representation for a hardware GPIO pin, the :on is the current state, and the :ticker is a function to send the next blink. You could imagine this struct having a count integer to track the number of blinks, but we'll keep this program simple.

Let's build the startup machinery, including a constructor to make dealing with options simple:

```
def new(opts) do
  %__MODULE__{
    on: false,
    led: LED.open(opts[:pin] || @pin),
    ticker: opts[:ticker] || &wait/0
  }
```

```
end

def start_link(opts \\ []) do
  GenServer.start_link(__MODULE__, opts, name: __MODULE__)
end

def init(opts \\ []) do
  send(self(), :blink)
  {:ok, new(opts)}
end

def wait, do: Process.send_after(self(), :blink, 1000)
```

We have a new constructor that creates the state for the GenServer. Notice we have convenient defaults for every argument but preserve flexibility by making each option configurable. The start_link/3 function starts the process, naming it __MODULE__ so we'll be able to use the Server name instead of the pid. We also provide the init function to send the initial :blink message and return the initial state of the GenServer.

The wait/0 function will wait a bit of time before triggering the next :blink. Notice that we make this function configurable in the :ticker argument because our tests will be more useful if they don't always have to send messages or sleep.

Now, let's provide the API and implementation of the :blink message.

```
  def handle_info(:blink, blinker) do
    blinker.ticker.()
    {:noreply, blink(blinker)}
  end

  defp blink(%{on: true}=blinker) do
    LED.on(blinker.led)
    %{blinker| on: false}
  end
  defp blink(%{on: false}=blinker) do
    LED.off(blinker.led)
    %{blinker| on: true}
  end
end
```

The handle_info/2 function processes a single :blink by sending the next blink message and calling the blink/1 function to do the bulk of the work. The individual blink/1 functions match on whether the light is on or off. Then they call the appropriate LED functions to turn the light off or on and return a new blinker with a toggled blinker.on field.

It's a short program with a complex flow, but since we manage the complexity one layer at a time, the code is remarkably easy to follow.

Test Drive the Simple Blinker

Let's fire it up. Start iex -S mix without a target to run on the host, or recompile if it's already open. Then exercise the blinker, like this:

```
iex(2)> alias Blinker.Server
iex(3)> Server.start_link
Opening 26
Off: #Reference<0.4070557734.2003697698.10599>
{:ok, #PID<0.273.0>}
On: #Reference<0.4070557734.2003697698.10599>
Off: #Reference<0.4070557734.2003697698.10599>
On: #Reference<0.4070557734.2003697698.10599>
Off: #Reference<0.4070557734.2003697698.10599>
```

It works! It's pretty nice that we can test things without burning firmware because we're already confident that our LEDs work. The development time messages give us reasonable confidence that our blinker is working because they are triggering the right message at the right time.

Now, you already know how to burn firmware. It's almost going to be anticlimactic because you've already verified that you can blink your LED module from the host and that the LED module can control the physical circuit. With a MIX_TARGET of rpi0, run mix firmware, run mix upload, and then shell into your device with mix nerves.local, like this:

```
Blinker.Server
iex(3)> Server.start_link
```

And, as shown in the figure on page 8, the light mercifully blinks!

The blinking is simple, but the time honored computer-controlled flashing is almost enough to make a budding maker weep for joy. It's time to wrap up.