Extracted from:

# Build a Binary Clock with Elixir and Nerves

## Use Layering to Produce Better Embedded Systems

This PDF file contains pages extracted from *Build a Binary Clock with Elixir and Nerves*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.



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

## Adapters Run One System, Three Ways

The first boundary that interacts with hardware is an *abstraction layer* called an *adapter*. These layers let programmers present one interface and multiple implementations. The goal is to have one program that runs in three places with as little disruption as possible.

If you wanted to, you could add a bit of compiler safety with a behaviour.[1] We're going to leave the behaviour implementation to you. Because Elixir is a dynamically typed language, all you technically need to do is provide adapter modules that present functions with the same names and arities.

Each of the adapters will handle a different use case. The test layer needs individual bits, the hardware layer needs binaries that work with `Circuits.SPI`, and the development layer needs to show pretty strings that represent the clock face.

**Bruce says:**

**My Nerves Breakthrough**

I took an initial pass at Nerves four years before I wrote this book but had a difficult time. Teaching OTP and applying the software layering techniques I taught opened up a whole new world for me. The main lesson was that interfaces allow back ends for the same system. Establishing interfaces for test, development, and production made everything click.

The `Circuits.SPI` interface we'll use for the target is based on a concept called a *bus*. Busses potentially have multiple devices, and a software layer must open one to interact with it, much like a file in an operating system. That means each adapter will need a *constructor* function to open the adapter. Then, each function will need a converter to present the LED pattern to the user. Let's start with the target.

### The Target Adapter

We'll make the adapters structs so they'll have the actual module built in as the `__struct__` key.[2] The target adapter must physically open the bus and send the bytes representing the clock face. In `lib/clock/adapter/target.ex`, build the constructor first:

1. https://embedded-elixir.com/post/2018-09-25-mocks-and-explicit-contracts-expansion/
2. https://elixir-lang.org/getting-started/structs.html

```
defmodule Clock.Adapter.Target do
  defstruct [:time, :spi]
  alias Clock.Core
  alias Circuits.SPI

  def open(bus, time) do
    :timer.send_interval(1_000, :tick)

    bus = bus || hd(SPI.bus_names())
    {:ok, spi} = SPI.open(bus)
    %__MODULE__{time: time, spi: spi}
  end
```

The constructor will need the `spi` reference and the time. The `open/2` function opens the `bus` and returns the adapter with the `time` and `spi` keys. Next, present the bytes to the user, like this:

```
  def show(adapter, time) do
    adapter
    |> Map.put(:time, time)
    |> transfer()
  end

  defp transfer(adapter) do
    bytes = adapter.time |> Core.new |> Core.to_leds(:bytes)
    SPI.transfer(adapter.spi, bytes)
    adapter
  end
end
```

We add the time to the adapter, and then send the adapter to a private function to transfer the bytes via `Circuits.SPI` using the data we build from the core. We return the adapter so the server will have the last time presented for debugging purposes.

Pausing quickly to test this function makes sense:

```
iex> a = Target.open "bus", Time.utc_now |> Target.show(Time.utc_now)
%Clock.Adapter.Target{
  spi: #Reference<0.862938587.3806461979.14092>,
  time: ~T[20:10:31.306738]
}
```

It appears to be working. Take the time to revel in your work. Build and push firmware to the target, and you'll be able to shell out to the Pi and display the time with LEDs. Do a brief happy dance, and then we'll build a test layer.

## The Test Adapter

The testing adapter will look much like the target one, with a couple of exceptions. First, there's no need to open an adapter. Second, rather than translating bytes, it makes more sense to add the bits to the adapter, so a test case could conceivably collect a few ticks and check the values using a strategy called *mocking*.

The lib/clock/adapter/test.ex file tells the story:

```
defmodule Clock.Adapter.Test do
  defstruct [:time, bits: []]
  alias Clock.Core

  def open(_bus \\ nil, time \\ Time.utc_now) do
    %__MODULE__{time: time}
  end
```

The defstruct across the adapters does not have to match. This one has a bits part to accumulate consecutive clock readings. There's no need for a spi key because we're not connected to hardware, so open/3 simply returns the time with the default values and moves on.

Now, let's show the results reducer:

```
  def show(adapter, time) do
    adapter
    |> Map.put(:time, time)
    |> concat
  end

  defp concat(adapter) do
    bits = adapter.time |> Core.new |> Core.to_leds(:none)
    %{adapter| bits: [bits| adapter.bits]}
  end
end
```

The only difference is the concat/1 function that tracks bits from the Core in the adapter accumulator. When you write test cases as an exercise, you'll use this bit to click your clock through a couple of cycles and make sure that show is computing bits correctly.

Testing this adapter means writing a test. Put it in test/adapter_test.exs:

```
defmodule AdapterTest do
  use ExUnit.Case
  import Clock.Adapter.Test

  test "Tracks time" do
    adapter =
      open(:unused, ~T[20:13:17.304475])
      |> show(~T[01:02:04.0])
      |> show(~T[01:02:05.0])

    [second, first] = adapter.bits

    assert [0, 0, 1|_rest] = first
    assert [1, 0, 1, 0, 0, 0, 1|_rest] = second
  end
end
```

This is a test of only the Adapter.Test module, but a test of the GenServer would work the same way. Neither of these adapters is convenient for IEx. A development adapter should make it easy to run our project in the console. We'd like to see messages printed or logged when important things happen. We don't care about the hardware because the development mode will run on the host. Let's build a development adapter next.

## The Dev Adapter

The dev adapter in lib/clock/adapter/dev.ex will be much like the test adapter but will send a log message rather than adding bits to the console. The ring logger will allow this adapter to work on the Pi for debugging as well. Let's see how it works:

```
defmodule Clock.Adapter.Dev do
  defstruct [:time]
  require Logger
  alias Clock.Core

  def open(_bus \\ nil, time \\ Time.utc_now) do
    :timer.send_interval(1_000, :tick)
    %__MODULE__{time: time}
  end
```

The struct needs a time, but not the spi key. The spi interface is meaningless on the host; the hardware is elsewhere. Still, this adapter is a great place to establish the ticks that will make our GenServer run later. This design will allow for the target and development environments to have a running

GenServer, and the test environment can test the features of the GenServer by explicitly sending tick messages instead of waiting on automated ticks. That way, the tests can be faster but still ensure the integrity of the software layer.

Now, let's see the show/2 reducer.

```
  def show(adapter, time) do
    adapter
    |> Map.put(:time, time)
    |> log
  end

  defp log(adapter) do
    face = adapter.time |> Core.new |> Core.to_leds(:pretty)
    Logger.debug("Clock face: #{face}")
    adapter
  end
end
```

This reducer works like the others. It has a custom function to show the clock face. The face is primitive, but it can easily be extended later based on the isolated :pretty formatter in the core. Now try it out:

```
iex> RingLogger.attach
:ok
iex> Clock.Adapter.Dev.open |> Clock.Adapter.Dev.show(Time.utc_now)

07:26:20.889 [debug] Clock face: --*-*------------*-**-*-
%Clock.Adapter.Dev{time: ~T[12:26:20.889926]}
```

It works, showing a friendly clock representation while the logger is attached. You can come back and improve the representation later. The important thing is that we don't need to unpack the binaries to see whether the bits are off or on.

Now let's build the GenServer in the services layer.