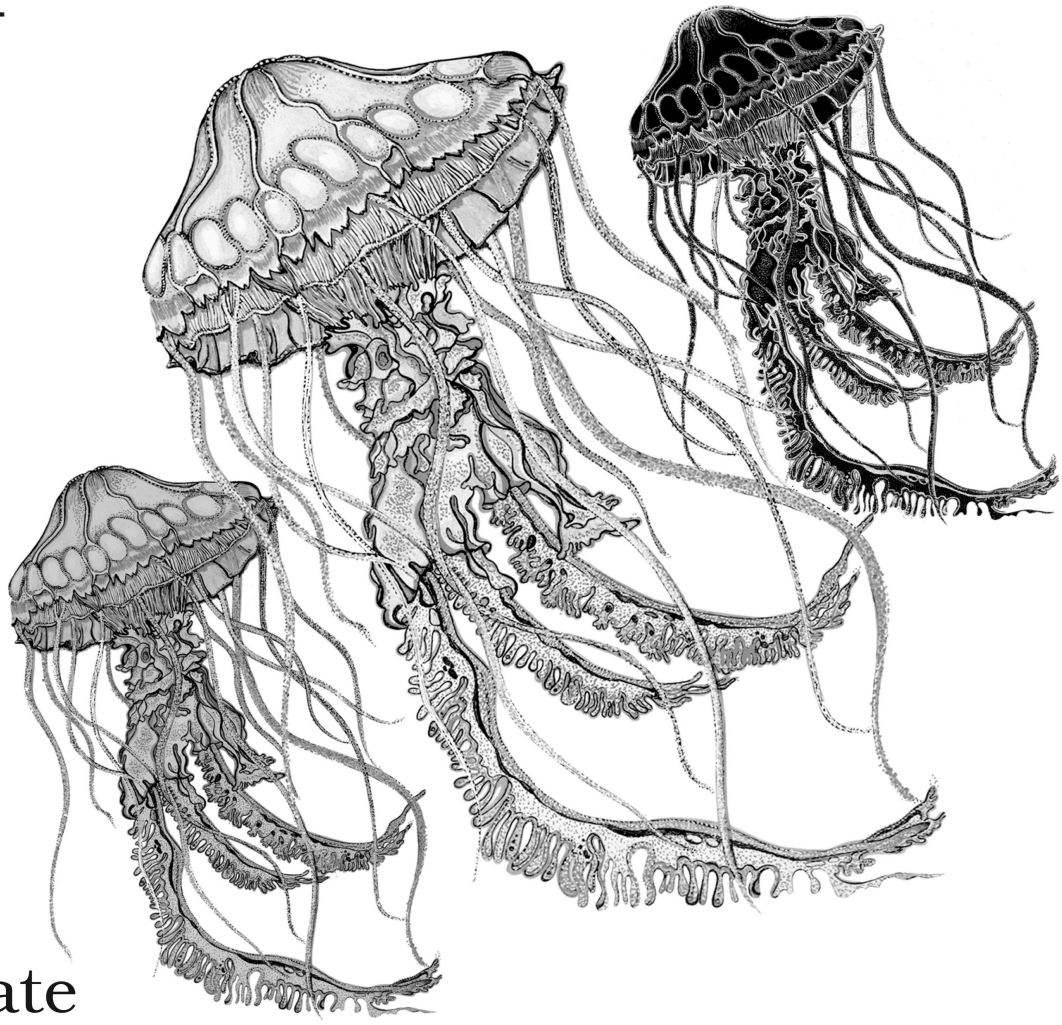
The Pragmatic Programmers

grox.io Learning

Programmer Passport

Nerves



Bruce A. Tate

Edited by Jacquelyn Carter

Programmer Passport: Nerves

Bruce Tate

The Pragmatic Bookshelf
Raleigh, North Carolina

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

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

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

ISBN-13: pending
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—Monthname, yyyy

Contents

Change History

The book you're reading is in beta. This means that we update it frequently. Here is the list of the major changes that have been made at each beta release of the book, with the most recent change first.

B1.0: Dec 1, 2020

- Initial beta release. We're releasing the first chapter that focuses on burning firmware with Nerves on a Raspberry Pi, and then building a circuit with an LED we'll control remotely from IEx.

CHAPTER 1

Nerves Tooling

Elixir is an excellent general purpose language. It has features that let you write programs that are more reliable, easier to understand, and highly concurrent. Many of those same capabilities make Elixir an ideal language for embedded systems.

Embedded systems are smaller, special purpose computers used to control hardware. They show up in cars, appliances, industrial controllers, and more. In traditional embedded projects, most developers use languages like C and Java. Until recently, lower level computers required speed and a compact footprint because embedded systems optimized power requirements, cost, and footprint over computing power. Most embedded chips were shockingly underpowered. That's no longer true.

Today, for ten bucks, a hobbyist can get a simple computer with multiple cores and gigabits of onboard memory. As embedded processors get more sophisticated, the demand for more complex systems grows, such as fridges that double as web browsers and doorbells that double as security cameras and networked intercoms. To support these new requirements, we need higher level languages with more powerful features:

Concurrency

Devices like simple light switches are now networked devices, and thermostats have built in web servers. New embedded systems have multiple cores. Concurrency is a critical component when you mix sophisticated networking with complex user interfaces.

Reliability

A crash from a device in the field can be catastrophic. Reliable languages with stronger type systems than C and better failure management than Java are not just nice-to-have in embedded systems. They are critical.

Productivity

As software gets more complex, the software component of rolling out a consumer device like a video doorbell or a smart thermostat becomes a bigger part of the overall engineering project, and higher level languages are more productive.

Security

The internet of things provides conveniences that we could not have imagined a short time ago, but such automation comes at a cost with new attack possibilities. Beyond preventing someone outside from shouting "Hey, Google, open the front door," we need software that brings along the full weight of built in encryption keys.

That list of requirements no longer looks like a great fit for C. When all is said and done, embedded systems are no longer tiny snippets of disconnected code. They are full-featured distributed, concurrent systems, and *that* looks like a marketing list for functional programming and languages like Erlang or Elixir.

A few years after the creation of Elixir, Justin Schneck and Frank Hunleth recognized this opportunity and founded the Nerves Project. Nerves is the platform you'll use to build embedded systems with Elixir. On the project's website,[1] you'll find this description:

> Nerves is the open-source platform and infrastructure you need to build, deploy, and securely manage your fleet of IoT devices at speed and scale.

As the years passed, more people joined the community. Nerves became a staple at the yearly ElixirConf conferences and successful projects began to emerge. Nerves was a winner. In the chapters to follow, you're going to find out why.

Over the next few chapters, you're going to use Nerves to build a few tiny custom devices ranging from simple to complex. Let's plan our approach.

Our Plan

It's often hard to get started when working with hardware because there are so many small things that can go wrong. For that reason, it's important to establish several small quick wins instead of making one full project work end to end. So it is with Nerves.

1. https://www.nerves-project.org

In this book, you're going to be programming Nerves-compatible computers. The first few chapters will focus on using Nerves, establishing a flow with the platform, and learning to build code in layers. Then, in the spirit of Groxio's focus on data, we'll shift our focus to a series of sensors. You'll explore how to work with sensors and then build in networking to network your Nerves systems to other computer systems through wired and wireless connections.

Here's what our plan looks like in detail.

Burn Firmware

In early embedded computers, firmware was a program stored on read-only memory. Firmware had the programs that would run when you booted up an embedded computer. As the complexity of those firmware programs grew, it became important to be able to write updates to those programs and embed them without creating new hardware. Today's firmware is often installed on erasable storage, potentially on SSD cards. These cards have the same kind of memory card you'd use on a camera or drone.

Nerves works by combining the Elixir programs that you write with everything else that a specialized embedded computer needs to run. An increasing number of these tiny devices actually run the Unix operating system, and Nerves is built to run on those.

You'll start by installing a firmware program written by the Nerves team on a tiny embedded computer, called a *target*. This first step will verify that you can use your Nerves tool chain to change the firmware on your target. You'll install the firmware chip in your target, and connect to it using a USB cable. You'll then learn some of the same debugging techniques you'll use later on to build and install firmware.

Next, you'll build the simplest of circuits, a single LED that you'll control with one of the input-output features of your target. Once you've done that much, you'll connect to your target from your develoopment computer, called a *host*. You'll control your LED by making a remote connection from your host computer to your target. Then, you'll use IEx on your target to access the embedded program you burned. This step will demonstrate that you can build circuits, install them on a target, and control them with a host.

To finish off the chapter, you'll use your embedded Elixir firmware to blink your LED using IEx. When you're done, you'll know:

- You have a working Nerves tool chain for burning firmware

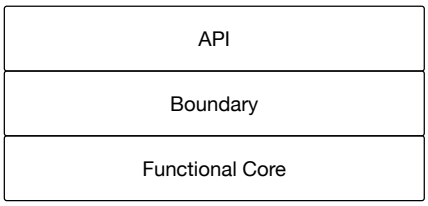
- You can remotely engage debugging tools to control your target from your development computer
- You have a working circuit
- You can exercise your circuit using IEx to talk to programs on firmware

This first step can be devastatingly complex, when you don't know about how Nerves works, but by taking one step at a time, you can limit potential problems.

Programming in Layers

After burning firmware containing an existing program using Nerves, you'll use Nerves to write your own. You'll add *compilation for a target* to your tool chain. You'll use Elixir's `mix` to compile software, and use the `MIX_TARGET` to define the hardware that will run your program. Nerves will do the rest. Nerves will build an image that has your program and everything else your embedded device needs.

You'll program in layers. The following figure tells the story.



Let's walk through each layer in the figure, starting with the functional core on the bottom, and working up. The *core* has the bulk of your logic, so you can spend most of your time reasoning about pure functions instead of complex processes or external interfaces that might fail. You'll wrap that core with a *boundary* layer. The boundary will help manage uncertainty, and deal with external interfaces or process machinery that's common when working with hardware. Finally, you'll establish an *API* layer, a friendly programming interface other programs can use to access your software in a predictable, user-friendly way.

After you've built a working version of your LED-blinking software, you'll build in networking so you can share data and features with the outside world. You'll use a simple hand-written web service you can call from an external system. You'll use this program to control your blinking LED light.

When you're done, you'll know how to:

- Write your own programs, and then burn them onto firmware

- Build software in layers, with functional cores that handle logic and boundaries to handle external interfaces
- Connect to your embedded device from networked computers to burn firmware, collect data, or use circuits you build, like your LED circuit

When this step is done, you'll have a working Nerves skeleton. You'll use this function as the foundation of the services you build throughout the rest of the book.

Work With Sensors

In the final few chapters, you'll do a variety of jobs with sensors. Mainly, you'll make third-party sensor data available via your service. Together, we'll choose from sensors that check air quality, temperature, light, and the like. You'll fill out these details as you get closer to the release of these chapters around mid January.

In general, the plan for each of the sensors will be the same. You'll build a circuit with a given sensor, install a driver, write some code to read from the sensor, and wrap it in a service that you can access through a simple web app.

With a plan for the whole Nerves book, we can build a shopping list before you work on your first project. Everything else depends on installing firmware to control your embedded device, so let's start there.

Let's plan our first tiny project, installing firmware.

Project: Install Firmware

This project involves purchasing a target, and loading a working Nerves program. Then, you'll talk to the target from your host. You'll want to make one tiny step at a time, so in this initial project, you won't write your own program. Instead, you'll load up a known working piece of firmware onto your target.

Here's how you're going to proceed. This list of tasks will get you to the point where you've loaded firmware, and confirmed that your computer is working:

- Choose a target computer
- Get all of the hardware you'll need
- Install a tool to load your firmware
- Download and burn firmware for your target
- Connect to your host computer, and explore it in IEx

That's a long list, but aside from the shopping trip, the project is going to go quickly. This project tracks closely with the first few Groxio[2] videos. Let's pick an embedded computer.

Choose a Computer

For this initial project, your target will need to be able to do at least three things.

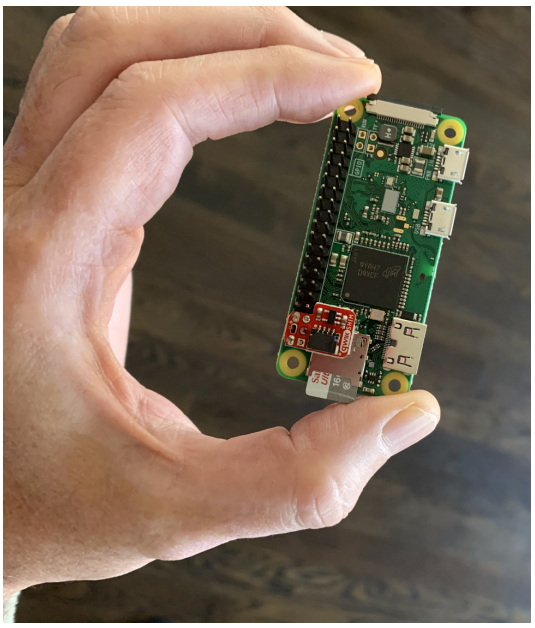
First, it must be on one of the two Nerves target lists. The first list of officially supported hardware[3] contains the list of targets that the Nerves team will help support, but if a computer you want to use isn't there, you can check the second list of community supported hardware[4] for a more exhaustive list. Unless you're experienced with Nerves, stay with computers on the first list.

The second requirement is connectivity. You need a computer with wireless networking and a USB connection port so you can interact with your computer through both wired and wireless connections using supported hardware.

The third requirement is sensor support so you can complete additional Groxio projects with supported sensors. You'll want to use sensors for light, air quality, and the like.

Luckily, in recent years, a new wave of tiny general purpose computers have entered production. These computers are small, cheap, and powerful. Among the most popular ones for hobbyists (called *makers*) is the Raspberry Pi. One version of that computer, the Raspberry Pi Zero, is available for around ten bucks, and is shown in the following figure.

2. https://grox.io/language/nerves/course
3. https://hexdocs.pm/nerves/targets.html
4. https://hex.pm/packages?search=depends:nerves_system_br



Slightly larger than a pack of gum, the Pi Zero has on board wireless and bluetooth, and you can see in the figure the two micro-USB ports on the right side near the top, and an onboard microSD card on the bottom with a loaded card.

On the left, you can see many general purpose attachment points called *pins* of various types that will let you interface with your own circuits. The Raspberry Pi Zero has more than enough power for our needs, though everything in our next few projects will work with other targets as well.

Now that you know your target embedded system, you can build a shopping list.

Project Shopping List

Our list specifies a target computer with *headers* pre-soldered on so you can connect your own wires to the various pins. You also need a microSD card and a reader to go with it. Finally, you'll need a micro-USB cable that will work with whatever USB connections your personal computer supports.

- Raspberry Pi Zero with headers[5]
- microSD card[6]
- microSD card reader (depends on your computer)

5. https://www.adafruit.com/product/3708
6. https://www.adafruit.com/product/2693