Extracted from:

Programming WebAssembly with Rust

Unified Development for Web, Mobile, and Embedded Applications

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Programming WebAssembly with Rust

Unified Development for Web, Mobile, and Embedded Applications

Kevin Hoffman
edited by Andrea Stewart

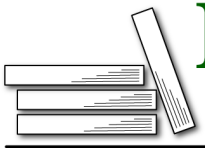
Programming WebAssembly with Rust

Unified Development for Web, Mobile, and Embedded Applications

Kevin Hoffman

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*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Andrea Stewart
Copy Editor: Jasmine Kwityn
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

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

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

ISBN-13: 978-1-68050-636-5
Book version: P1.0—March 2019

For my grandfather—Walter K. MacAdam—
inventor, tinkerer, and IEEE president. He
quietly supported my exposure to computers
and programming throughout my childhood,
often in ways I didn't know until after his death.
I always wanted to grow up to be like him, and
I only wish he could've seen this book.

CHAPTER 7

Exploring the Internet of WebAssembly Things

The *Internet of Things (IoT)* is now as ubiquitous as the Internet itself. Some people see this as a tremendous opportunity for growth and innovation while others are terrified of an impending future dominated by millions of woefully underprotected, overconnected devices.

Today we have smart watches, refrigerators, toasters, doorbells, clothing, and thousands of other things that attach the real world to the digital world of the Internet. Infrastucture companies want to sell us platforms to support our IoT applications, security companies want to help us secure our smart devices, and the maker community is constantly expanding and building open source, connected hardware. IoT represents a nearly infinite number of ways to spend and earn money, so it's no wonder it has inspired so much innovation.

As you've come to learn on your journey through this book, WebAssembly is about far more than just speeding up web applications. Its portable, compact format makes it ideal for systems under heavy disk, memory, and processing constraints. It's ideal for isolating business logic from presentation and, as you'll see in this chapter, from external, physical devices.

In this chapter, you'll take advantage of WebAssembly's portability and the Raspberry Pi's easy access to hardware systems to build a pluggable host that separates the logic of determining *what* to display on a hardware indicator from the *how* of displaying it. The LED and computer parts for this chapter's hardware are inexpensive, but even if you don't have a Raspberry Pi, you'll see how you can write and test code for hardware in isolation all from the comfort of your own workstation.

This chapter will operate on, and prove, the following two assumptions:

1. If a WebAssembly module can be hosted in a web browser or a console application, you can host it on a Raspberry Pi
2. If two WebAssembly modules adhere to the same contract, they can be interchanged like modular plugins

Before we start coding, let's take a tour of a use case illustrating the problem we want to solve.

Overview of the Generic Indicator Module

Let's assume that we've been tasked with designing and building part of an IoT project. This project is to build an autonomous wheeled robot that maneuvers its way through an obstacle course as they do in many robotics competitions.

Since we're working as part of a team and there are hundreds of individual pieces on this robot, we've been tasked with handling the *Generic Indicator Module System*. Since all hardware projects need acronyms, we'll call this one *GIMS*. Bonus points for a four-letter acronym, as that puts us just that tiny bit closer to feeling like NASA.

The robot will process multiple streams of sensor inputs from many different devices. GIMS's job is to allow the sensory input to be fed into a WebAssembly module that can then determine how the current state of some aspect of the robot should be visualized. We might have access to gauges, multi-colored LEDs, headlights that could turn on when it's dark—any number of amazing devices.

In some cases, the sensory input might already have been massaged a little bit by the more accurate timing of microcontrollers, while our GIMS will be running on a Raspberry Pi at the very heart of a robot. In this chapter, to keep from writing an entire new book on robotics and WebAssembly, we'll focus solely on the generic indicator system.

The robotics team leadership has built these types of competition robots before, and they know the pain and true price of the integration cost when they have to go back into their code and fuss with tiny details every time they change a piece of hardware. If you haven't played with microcontrollers, "maker kits," or Raspberry Pis, you might assume that LEDs are just LEDs —you control one the same way you control another. The truth is far more annoying.

The reality is that you can go from the simplest LED (apply current, it lights up, *magic!*) to chains of multicolor LEDs that operate with simple timing sequences to systems that use very specific communications protocols like *I2C*.[1] Changing your peripherals mid-build can be a "stop the world" event, but we can engineer our way around that with a little help from WebAssembly.

With the GIMS design, we'll be putting the indicator logic—which translates a series of sensor inputs into a series of hardware manipulation commands—into WebAssembly. This way, the indicator logic remains isolated and loosely coupled from the physical indicator(s). If someone changes an LED from a simple light-and-resistor to a brick of 200 "LED pixels," they should be able to make a small change to an interface layer and leave our indicator relatively unbothered.

In short, we're taking the software engineering principles of *loose coupling* and *separation of concerns* and, with the power of WebAssembly, bringing them to the world of consumer-grade electronics. The first thing we're going to need to do in order to make that happen is design the contract between the host and the WebAssembly modules.

Designing the Module Contract

As you saw in the chapter on basic JavaScript integration, the contract between a WebAssembly module and its host is a very basic, low-level contract built from numeric primitives. That contract defines how linear memory is accessed, how parameter values can be passed to functions, how we can invoke functions exported from a module, and within the module, invoke functions imported from the host.

Above these low-level bindings, what we need is an API. We need an API that lets the host invoke functions whenever there are new sensor readings available. This API also needs to let the WebAssembly module control the indicator lights. We could even let WebAssembly modules control more hardware like motors and actuators, but that's outside the scope of our GIMS project (though it certainly could be a lot of fun to explore).

First let's think about sensor inputs. I'm sure in real-world circumstances, our sensors would have all different kinds of outputs, and some might have more than one value. But knowing that we're doing this for the Raspberry Pi, and that other microcontrollers closer to the data might be able to massage it for us, it's safe to assume that we'll be able to get a decimal value from each

1. i2c.info learn.sparkfun.com/tutorials/i2c

sensor whenever a value changes. So our host is going to want to call a function like the one below to inform our wasm module of a new data point:

```
fn sensor_update(sensor_id: i32, sensor_value: f64) -> f64;
```

Let's say the motor speed is sensor 1, the ambient light detector is sensor 2, the collision detector is sensor 3, the battery of our main laser cannon is 20, etc. We'll have to maintain the Rust-equivalent of a header file so that we can ensure all our modules are operating on the same list of sensors. If the team disagrees on sensor IDs, we're basically back at square 1 and haven't fixed any problems.

Another function we want the host to be able to call is apply(). If we need to animate or update our display over time, we could probably attempt some kind of intricate threading scheme to run each module, but it's far easier to use the "game loop" model and just invoke something like apply() *n* times per second. We might be able to do fancier things when threading becomes a part of a future version of the WebAssembly specification, but this is good enough for our needs today.

To let the modules know about the passage of time, we can, however, invoke the same function at fixed intervals and pass a *frame* value that increases for each call. We can either agree on a frame rate for updates or write our code so it doesn't really matter:

```
fn apply(frame: i64);
```

For example, an animated indicator might have apply called 20 times per second.

That's it for the input to our modules. Now we need to give the WebAssembly modules a way to control hardware without tightly coupling them to it. For this, we'll abstract over the notion of setting the color of an individual LED with a function that takes an LED index and 3 RGB values between 0 and 255, like so:

```
fn set_led(led_index: i32, r: i32, g: i32, b: i32);
```

If you think back to the fundamentals chapter, recall that while we're allowed to use plenty of data types privately within the module code, we can't import and export higher-level data types like structs. Let's recap and take a look at the three functions in the *GIMS* API contract (import and export are from the point of view of the WebAssembly module) as shown in the table on page 11.

Now that we've got a preliminary contract defined between our hardware host and the wasm modules, we can create a couple of different indicators.

Name	Direction	Params	Returns
apply()	export	• frame	None
sensor_update()	export	• sensor_id • sensor_value	Value
set_led()	import	• led_index • red • green • blue	None

Creating Indicator Modules

Creating an indicator module is really just a matter of creating a regular Rust-based WebAssembly module that adheres to the contract we've defined. You've seen how to create wasm modules using Rust a number of times throughout this book, so it should be easy to get started.

To start, create a root directory that will hold a battery indicator, an animated indicator, and the host application. I chose to call my directory gims, but you can choose whatever you like. As a convenience, to allow you to run builds and tests on all subdirectories at once, you can create a new Cargo.toml in the gims directory with the following contents:

iot_gims/Cargo.toml

```
[workspace]

members = [
  "animatedindicator",
  "batteryindicator",
  "pihost"
]
```

Use cargo new --lib to create the batteryindicator and animatedindicator projects, and cargo new --bin to create the pihost project.

Creating the Battery Indicator

The first indicator module we're going to build is a battery indicator. Its operation is fairly simple: one of the sensor inputs represents the amount of battery remaining as a percentage. In response to that percentage, we're going to control the color of a group of eight LEDs.

These LED indicators are each capable of lighting up with colors comprised of RGB components ranging from 0 through 255. The actual hardware used

will be a Blinkt! module from Pimoroni, and I'll include all the details later in case you want to go shopping for your own kits.

Its core logic will be to convert a number from 0-100 into an eight-element array with each element containing an RGB color value—think of it like an LED-based progress bar. For this indicator, we'll divide the percentages among the LEDs and only light them up if the value is >= the base value for that LED. Figuring out the base value for each LED is simple—divide the eight LEDs by 100 and we get 12.5% per LED.

Update your lib.rs with the following code:

iot_gims/batteryindicator/src/lib.rs

```
#[derive(PartialEq, Debug, Clone)]
(1) struct LedColor(i32, i32, i32);

const SENSOR_BATTERY: i32 = 20;

const OFF:LedColor =  LedColor(0, 0, 0);
const YELLOW: LedColor = LedColor(255, 255, 0);
const GREEN: LedColor = LedColor(0, 255, 0);
const RED: LedColor = LedColor(255, 0, 0);
const PCT_PER_PIXEL: f64 = 12.5_f64;

extern "C" {
(2)     fn set_led(led_index: i32, r: i32, g: i32, b: i32);
}

#[no_mangle]
(3) pub extern "C" fn sensor_update(sensor_id: i32, sensor_value: f64) -> f64 {
    if sensor_id == SENSOR_BATTERY {
        set_leds(get_led_values(sensor_value));
    }
    sensor_value
}

#[no_mangle]
pub extern "C" fn apply(_frame: u32) {
    // NO OP, not an animated indicator
}

(4) fn get_led_values(battery_remaining: f64) -> [LedColor; 8] {
    let mut arr: [LedColor; 8] = [OFF,OFF,OFF,OFF,OFF,OFF,OFF,OFF,];
    let lit = (battery_remaining / PCT_PER_PIXEL).ceil();

    // 0 - 20 : Red
    // 21 - <50 : Yellow
    // 51 - 100 : Green

    let color = if 0.0 <= battery_remaining &&
        battery_remaining <= 20.0 {
        RED
```

```
    } else if battery_remaining > 20.0 && battery_remaining < 50.0 {
        YELLOW
    } else {
        GREEN
    };

    for idx in 0..lit as usize {
        arr[idx] = color.clone();
    }

    arr
}
```

```
❺ fn set_leds(values: [LedColor; 8]) {
    for x in 0..8 {
        let LedColor(r, g, b) = values[x];
        unsafe {
            set_led(x as i32, r,g,b);
        }
    }
}
```

❶ Create a tuple-struct to hold the three-color codes

❷ Import the set_led() function from our host

❸ Expose the sensor_update() and apply() functions to the host

❹ Core logic to convert a percentage into a set of eight color codes

❺ Invoke the unsafe import in a loop to set all the LED colors on the host

With this code in place, we're going to want to test our module before we plug it into real hardware.