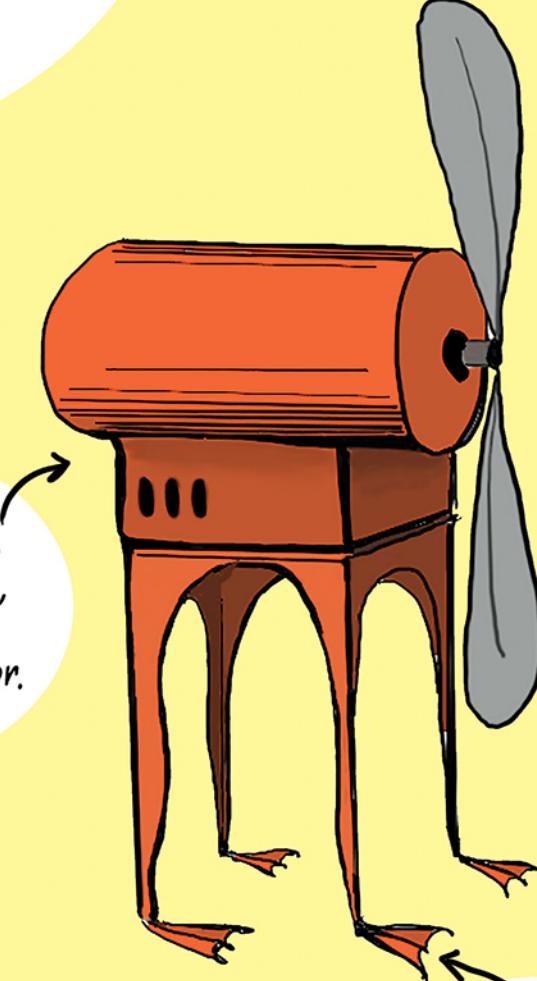


JAVASCRIPT ON THINGS

Hacking hardware for web developers

Lyza Danger Gardner



Circuitry and wires
are stashed inside,
with a vent for our
temperature sensor.

Ridiculous Feet:
because we can!



SAMPLE CHAPTER



JavaScript on Things

by Lyza Danger Gardner

Chapter 1

Copyright 2018 Manning Publications

brief contents

PART 1 A JAVASCRIPTER'S INTRODUCTION TO HARDWARE.....1

- 1 ■ Bringing JavaScript and hardware together 3
- 2 ■ Embarking on hardware with Arduino 26
- 3 ■ How to build circuits 48

PART 2 PROJECT BASICS: INPUT AND OUTPUT WITH JOHNNY-FIVE79

- 4 ■ Sensors and input 81
- 5 ■ Output: making things happen 107
- 6 ■ Output: making things move 144

PART 3 MORE SOPHISTICATED PROJECTS179

- 7 ■ Serial communication 181
- 8 ■ Projects without wires 214
- 9 ■ Building your own thing 253

PART 4 USING JAVASCRIPT WITH HARDWARE IN OTHER ENVIRONMENTS	295
10 ■ JavaScript and constrained hardware	297
11 ■ Building with Node.js and tiny computers	332
12 ■ In the cloud, in the browser, and beyond	375

Part 1

A JavaScripter's introduction to hardware

T

his part of the book will introduce you to the fundamentals of embedded systems and electronic circuits. In chapter 1, you'll learn what *embedded systems* are and how to analyze their constituent components. We'll spend some time looking at what it means for JavaScript to "control" hardware, and we'll examine the different ways that JavaScript and electronics can work together.

You'll meet the Arduino Uno R3 development board in chapter 2, which we'll use with all of the experiments through chapter 7. You'll learn what the main parts of development boards do and how they interact with other software and hardware components. You'll try out some basic LED experiments with the Uno using both the Arduino IDE and the Johnny-Five Node.js framework.

Chapter 3 will teach you the key fundamentals of electronic circuitry, diving into Ohm's law and the relationships between voltage, current, and resistance. You'll work on a breadboard, constructing series and parallel circuits that contain multiple LEDs.

When you're finished with this part of the book, you'll have grasp of the basic embedded-system underpinnings and core circuit concepts. You'll be ready to start building small, JavaScript-controlled projects with different kinds of inputs and outputs.

1

Bringing JavaScript and hardware together

This chapter covers

- Components and hardware involved in hobbyist projects and the “internet of things”
- Common components of embedded systems
- Different methods for using JavaScript with embedded systems
- Tools and supplies you’ll need to start building

As a JavaScript-savvy web developer, you make logical alchemy happen every day. But now it’s possible to wield your software-development skills in a new way, to program and control things in the real world. In this chapter, you’ll learn about the hardware involved in different kinds of projects and devices, and you’ll also see how JavaScript and hardware can work together.

We’re surrounded by little magical *things* that blend the physical world with the realm of the logical, connected, and virtual (figure 1.1). A keychain that broadcasts its location wirelessly so you can find it with an app on your smartphone. A plant pot that makes whining noises when it needs to be watered, or, better yet, sends you

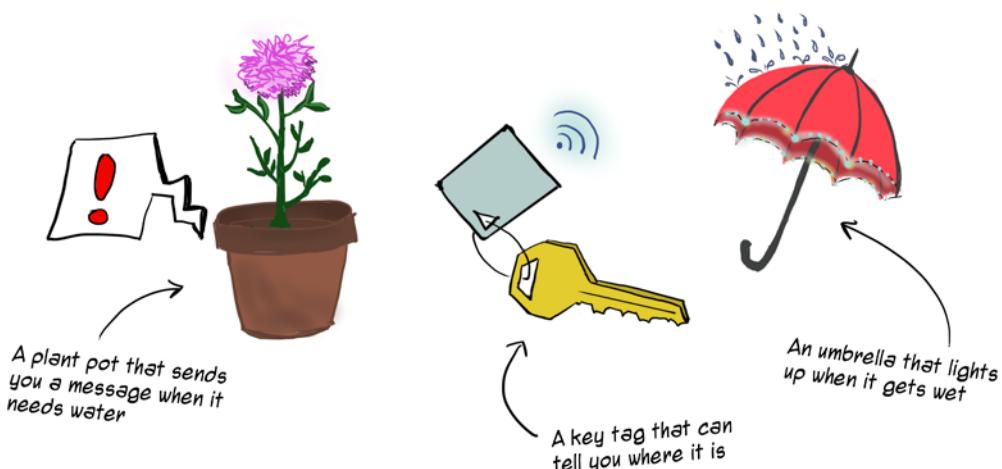


Figure 1.1 Oh, the magical things in our world!

a petulant text message. Billions of such objects blink, beep, tweet, automatically dim the lights, make customized pots of tea, and otherwise perform their specialized duties across the planet.

It's fun to build this stuff. The creativity involved when crafting with these kinds of physical gadgets, the grassroots charm of inventive homebrew projects—these are the kinds of things that hold appeal for web developers. We're cut out for prototyping, experimenting with new technologies, and blazing our own trails.

But getting started can be intimidating. When we see all the wires and components, hear the jargon, stand on the outside looking in at hardware-hacking communities, the kinds of skills involved can feel formidable, foreign. As a JavaScript developer, you may be faced with some hurdles—perceived complexity, overabundant and scattered information, conflation of hardware and software concepts—as you make your tentative first forays into the world of physical hardware.

We're going to use your JavaScript know-how as an advantage, an aid to learning how to design and build the kinds of *things* that make up the “internet of things” (IoT) and inspire hardware hackers. You'll be able to use your software-development skills to skip past some distractions and get focused, quickly, on the new skills you need to learn.

To get a feel for the journey we're taking, let's first take a look at the kinds of things you'll be learning to build. Let's explore what we mean, exactly, when we say *things* or *hardware*.

1.1 The anatomy of hardware projects

We could build a little gadget that would automatically turn a fan on when it gets warm. This miniature, independent climate-control device would continuously monitor the temperature of the surrounding environment. When it gets too hot, the fan comes on. When it's nice and cool again, the fan turns off.

While we wouldn't win any prestigious awards for the invention of this admittedly pedestrian contrivance, its basic ingredients are common to the other—more inspiring—things you'll learn to build.

1.1.1 Inputs and outputs

The most important thing—really the only thing—our temperature-triggered device needs to do is turn a fan on when it's too toasty and turn it back off again when the area around it has cooled off. The motor-driven fan is an example of an *output device*.

To get continuous information about the temperature of the immediate environment—so that the device can make decisions about when to turn the fan on or off—we need data from an *input*, in this case a temperature sensor (figure 1.2).

Inputs provide incoming data to the system, and *sensors* are a type of input that provides data about the physical environment. There are all kinds of sensors you can use in projects: sensors for light, heat, noise, vibration, vapors, humidity, smells, motion, flames—you name it. Some, like our fan's temperature sensor, provide simple data—just a single value representing temperature—whereas others, like GPS or accelerometers, produce more elaborate data.

A project's *outputs* represent its net functionality to someone using it. Blinking lights, irritating beeping sounds, status readouts on LCD screens, a robotic arm moving sideways—all these are kinds of outputs. For this project, the fan is the sole output.

Not all inputs and outputs necessarily manifest in the physical world. A customer encountering an error when trying to order a product online (virtual input) might cause a red light to go on (physical output) on a device sitting on a support technician's desk. Conversely, a change in soil humidity (physical sensor input) might cause a plant pot to send a demanding text message (virtual output).

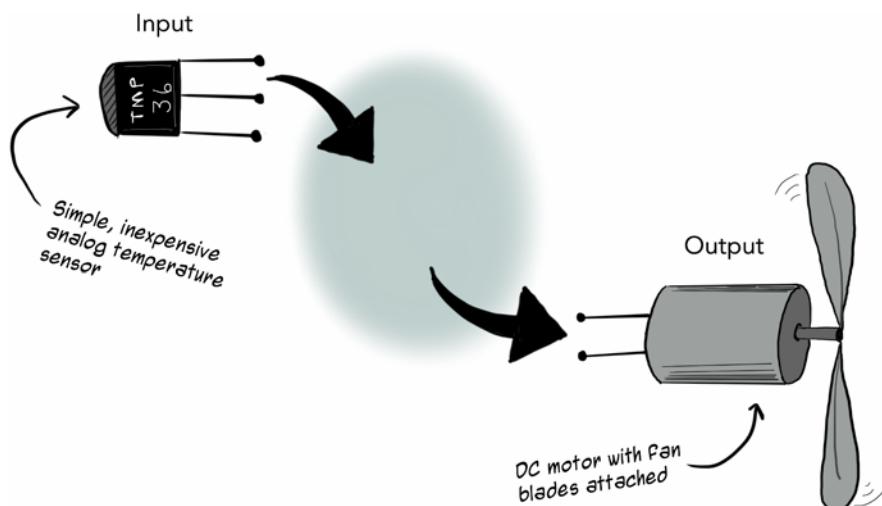


Figure 1.2 The automatic fan system needs to take *input* from a temperature sensor and manage the *output* of a motorized fan.

1.1.2 Processing

Our automatic fan also needs a brain, something that can pay attention to the temperature sensor's readings and turn the fan on when it gets too warm. The kind of brain it needs is in fact a tiny computer: a processor, some memory, and the ability to process inputs and control outputs. When processor, memory, and I/O functionality are contained in a single physical package, we call the resulting chip a *microcontroller* (figure 1.3).

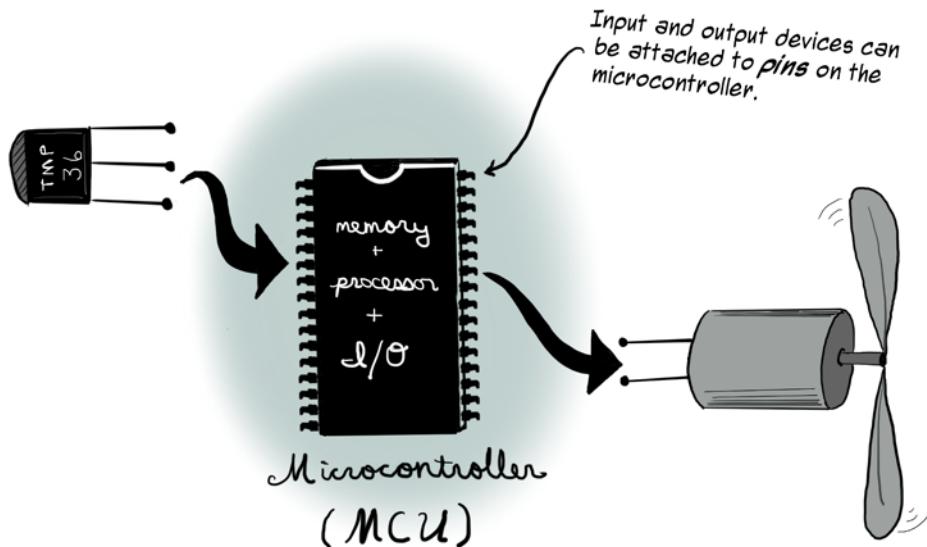


Figure 1.3 The automatic fan needs a brain. A popular option is a microcontroller, which combines a processor, memory, and I/O capabilities in a single package.

Microcontrollers (MCUs) aren't as powerful as the general-purpose processors in laptops. Most can't run a full operating system (most, not all, as you'll see), but they're cheap, reliable, small, and consume minimal power—that's why they're positively ubiquitous in hardware projects and products like our apocryphal automatic fan.

1.1.3 Power, circuits, and systems

We've now got input, output, and a brain—time to pull the bits together into a *system*. We'll need to connect the components using one or more electronic circuits and provide some power. Constructing a system involves both circuit design and the manipulation of components in physical space (figure 1.4).

Connecting wires directly to a microcontroller's tiny *pins* would require solder and a very steady hand. Not to mention that we'd end up with a lot of loose parts awkwardly floating around. To aid hardware developers, microcontrollers are often mounted onto physical *development boards* (figure 1.5). Among other things, boards make it easier to connect I/O devices to the microcontroller.

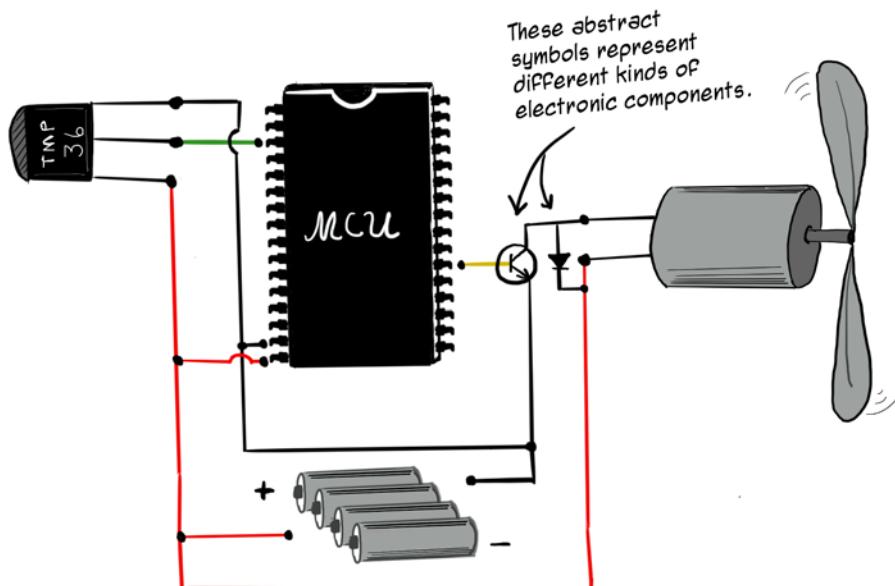


Figure 1.4 A rough schematic drawing showing how the fan's inputs, outputs, and microcontroller are connected in a system with power and circuitry. Don't stress out if the symbols are new to you—you'll be learning about circuitry as we continue our journey.

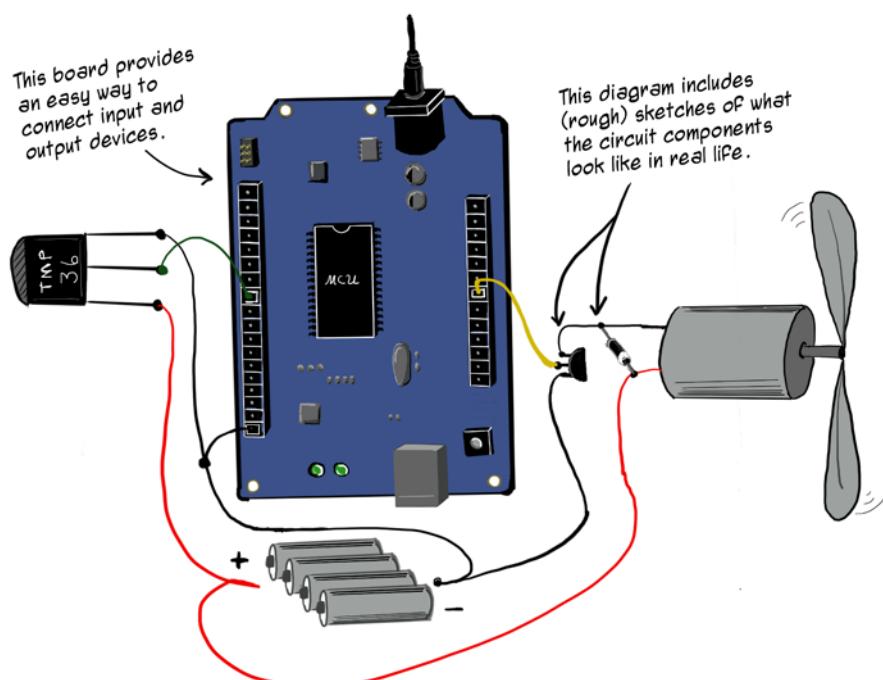


Figure 1.5 Microcontroller-based development boards make it more convenient to connect input and output devices.

A development board helps, but we’re still left with a number of loose wires and components. To help corral this, hardware developers use a prototyping tool called a *breadboard* (figure 1.6) to lay out circuits in physical space.

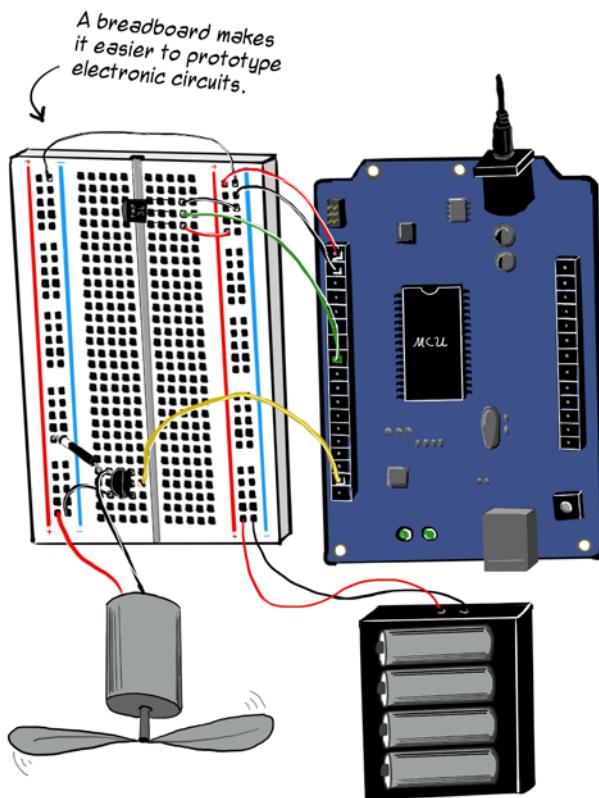


Figure 1.6 A breadboard provides an electrically connected grid on which to prototype electronic circuits.

1.1.4 Logic and firmware

Our hardware design is moving along, but you might be wondering how the microcontroller knows what to do. There’s logic involved here, as shown in the following listing: listening to the sensor, making decisions, sending instructions to turn the fan on or off.

Listing 1.1 Pseudo-code for temperature-triggered fan logic

```

initialize temperatureSensor
initialize outputFan
initialize fanThreshold to 30 (celsius temperature)

loop main
    read temperatureSensor value into currentTemp
    if currentTemp is greater than fanThreshold
        if outputFan is off
            turn outputFan on

```

```
else if currentTemp is less than or equal to fanThreshold  
    if outputFan is on  
        turn outputFan off
```

The dominant language for programming microcontrollers has long been C (or C-like derivatives). Writing C for microcontrollers tends to be platform-specific and can be quite low-level. References to specific memory addresses and bitwise operations are common.

The code is compiled to architecture-specific assembly code. To get the code into the project, it is physically uploaded, or *flashed*, to the microcontroller's *program memory*.

This program memory is usually non-volatile memory—ROM, the kind of memory that lets the microcontroller “remember” the program even if it's powered off (in contrast with RAM, which only retains its contents if it's powered). The space available for programs is constrained, often on the order of a few tens of kilobytes, meaning programs that run on microcontrollers need to be carefully optimized.

Once the program is flashed to the microcontroller, it functions as the microcontroller's *firmware*—when powered, the microcontroller runs the program continuously until it's programmed with something different (or otherwise reset).

For JavaScript developers accustomed to higher-level logic, this lower-level specificity may feel off-putting. Fret not. This is where JavaScript can help us, allowing us to write programs for microcontroller-based hardware without having to use C or tangling ourselves up in the nitty-gritty of hexadecimal register addresses right off the bat.

The process of getting program firmware onto microcontrollers has also become a lot easier thanks both to advances in chip technology and the wide availability of hobbyist-friendly development boards (figure 1.7).

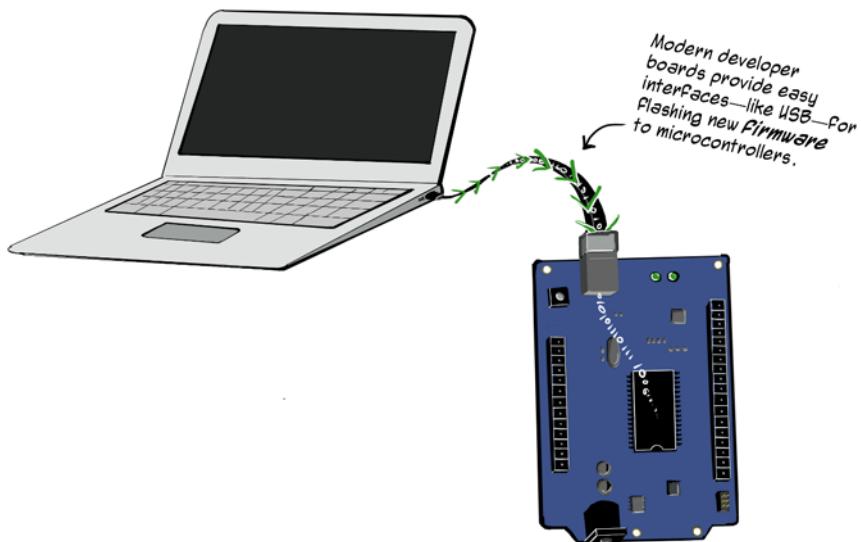


Figure 1.7 Non-volatile program memory (EEPROM and Flash) and user-friendly boards have made it easier to program microcontrollers with firmware.

EEPROM (electrically erasable programmable ROM), exemplified by the well-known *flash memory* medium, is commonly used in microcontrollers. This kind of rewritable memory makes it feasible to reprogram microcontrollers over and over again with different logic.

Development boards, in addition to making I/O connections easier, also aid hardware hackers by providing convenient interfaces for programming the board's microcontroller (USB is quite common). This alleviates the need for specialized hardware programming devices. These days, programming microcontrollers is often as easy as plugging in a USB cable and clicking a button in an IDE.

1.1.5 **Enclosures and packaging**

Our fan's design is almost done. But we can take it to the next level by packaging the auto-fan inside a nice enclosure—*embedding* our system inside of something, where its wires and circuits will be hidden from view (figure 1.8). Ta-da!

1.1.6 **Embedded systems**

Though the term *embedded system* can sound a bit formal or forbidding, it's not really too complicated. A tiny computer combining processor, memory, and I/O forms the brain. As you saw with our automatic fan, connecting the inputs, outputs, and microcomputer together and giving them power creates an independent *system*. We say it's *embedded* because it's often squirreled away inside of something—an enclosure, a teddy bear, a washing machine's control panel, an umbrella.

Though an automatic fan, an umbrella that lights up when it rains, and a tweeting teddy bear don't seem immediately similar, they have more in common than you might think. These examples, along with the majority of hardware projects and devices that form the IoT, can be described as *embedded systems*.

Now let's see how JavaScript fits into the picture.

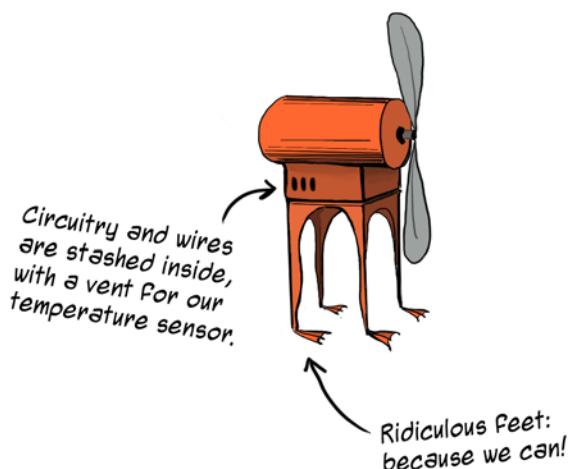


Figure 1.8 The completed, packaged, automatic fan is an example of an embedded system. Inputs and outputs are processed by a microcontroller-based microcomputer and supported by power and circuitry. And the whole thing is hidden inside a pretty fancy box, because, why not?

1.2 How JavaScript and hardware work together

When combining JavaScript with embedded systems, we still build electronic circuits in the same way as we would for other types of hardware projects. There are still inputs and outputs, wires and components. However, instead of using assembly code or C to define what the project's microcontroller or processor does, we use JavaScript.

There are several ways to do this, different *methods* for using JavaScript to provide the logic for hardware projects. These methods are categorized based on where the JavaScript logic itself executes: on a host computer separate from the embedded system, on the embedded system's microcontroller, or somewhere else entirely.

1.2.1 Host-client method

To get around the constraints of certain microcontrollers, the *host-client method* allows you to execute JavaScript on a more powerful *host* computer. As the host runs the code, it exchanges instructions and data with the embedded hardware, which behaves like a *client* (figure 1.9).

Many microcontrollers have limitations that impact their ability to run JavaScript. Program memory is constrained, meaning that complex programs either won't fit or

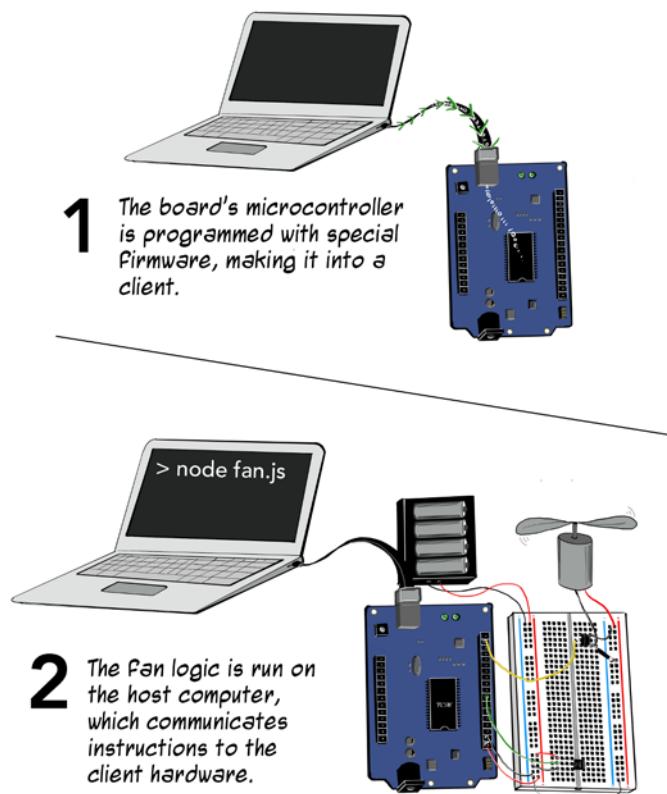


Figure 1.9 The host-client method of controlling hardware with JavaScript

have to be greatly optimized. Also, many inexpensive microcontrollers are built with 8- or 16-bit architectures running at clock speeds that are low relative to, say, desktop computers. Most wouldn't be up to the task of running an operating system, ruling out the ability to run a Node.js or other JavaScript runtime directly on the chip.

Instead, the host-client method involves executing JavaScript logic on a host computer, such as your laptop, which does have the brawn necessary to run a full OS. The host machine is able to run Node.js and can make use of the worldwide JavaScript software ecosystem (including npm and the web).

The trick to getting this setup to work is to make the client hardware (such as the microcontroller) and host system (your laptop) communicate with each other using a mutually intelligible “language”—a common API (figure 1.10).

To configure our automatic fan system to use this method, we'd first need to prepare the embedded hardware by uploading special firmware to the microcontroller's program memory. Instead of a specific, single-purpose program for controlling the fan, this firmware program makes the microcontroller able to communicate back and forth with other sources that speak the same “language” (the API). That is, it turns the microcontroller-based hardware into a client, all ears and ready to do the bidding of the host computer (figure 1.11).

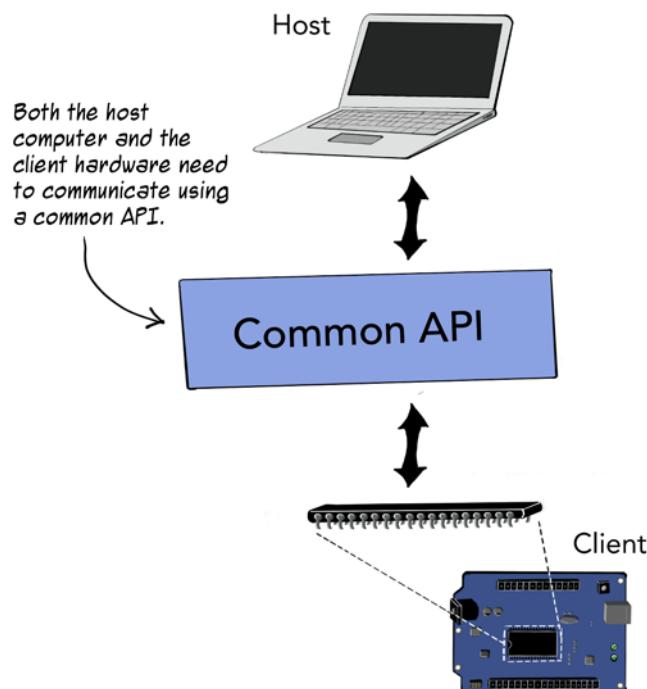


Figure 1.10 For host computer and client hardware to communicate in this method, they both need to use a common API.

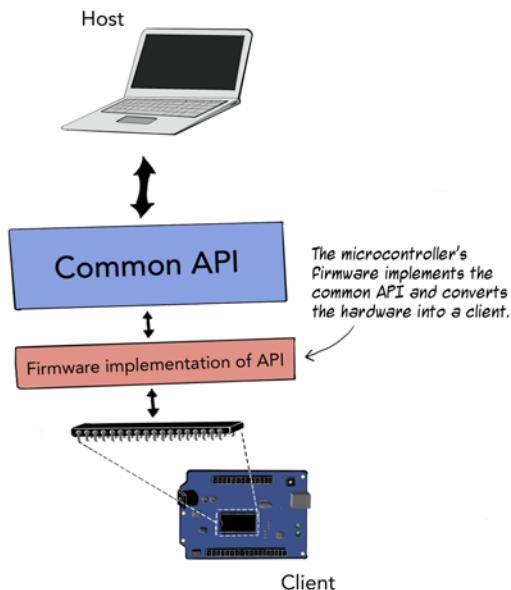


Figure 1.11 Specific firmware converts the microcontroller into a client.

The hardware is now ready to communicate—the next step is to write software for the fan, using the host computer. For the hardware and software to understand each other, the host computer needs to bark out instructions in a language the microcontroller can comprehend. To make this happen, we can write code using a library or framework that implements the common API (figure 1.12).

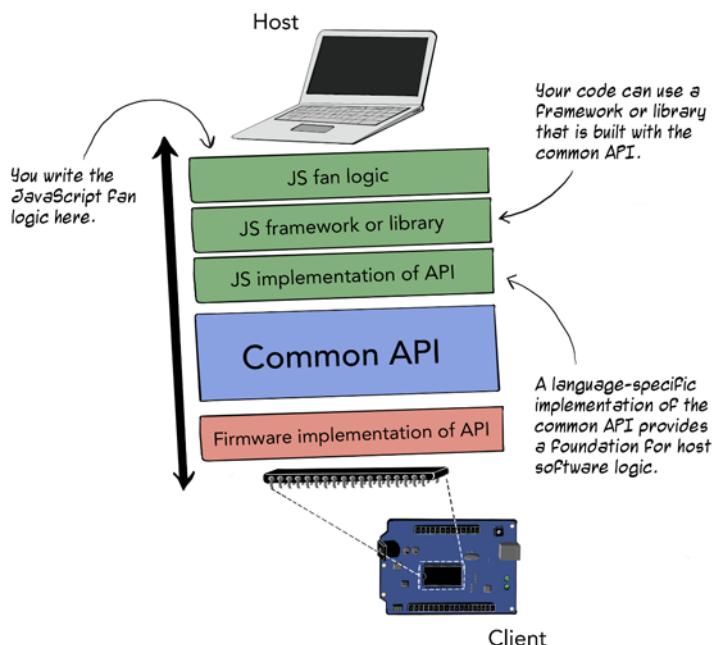


Figure 1.12 The host also needs to communicate using the common API.

The host is connected to the client hardware, either with a physical, cabled connection (often USB) or wirelessly (WiFi or Bluetooth).

Then we execute the fan-controlling JavaScript on the host computer. The host continuously communicates instructions for running the fan to the client. The client can also send messages back to the host, such as data from the temperature sensor (figure 1.13).

Don't panic, you won't have to write low-level firmware protocol API software! There are straightforward, open source options for firmware and Node.js frameworks that implement those firmware protocols, so you can write your host-side JavaScript logic with minimal fuss.

The benefits of the host-client approach are that it's easy to set up and it's supported on many platforms. What's more, it gives you access to the entire Node.js ecosystem, while avoiding the performance and memory constraints of inexpensive microcontrollers. The downside is that the client hardware is helpless without the host—it can only do its thing when the host computer is actively running the software.

We'll go wireless eventually, but we'll be starting out with the simplest of host-client options—USB tethering. That means that, for a while, your projects will be physically attached to your computer.

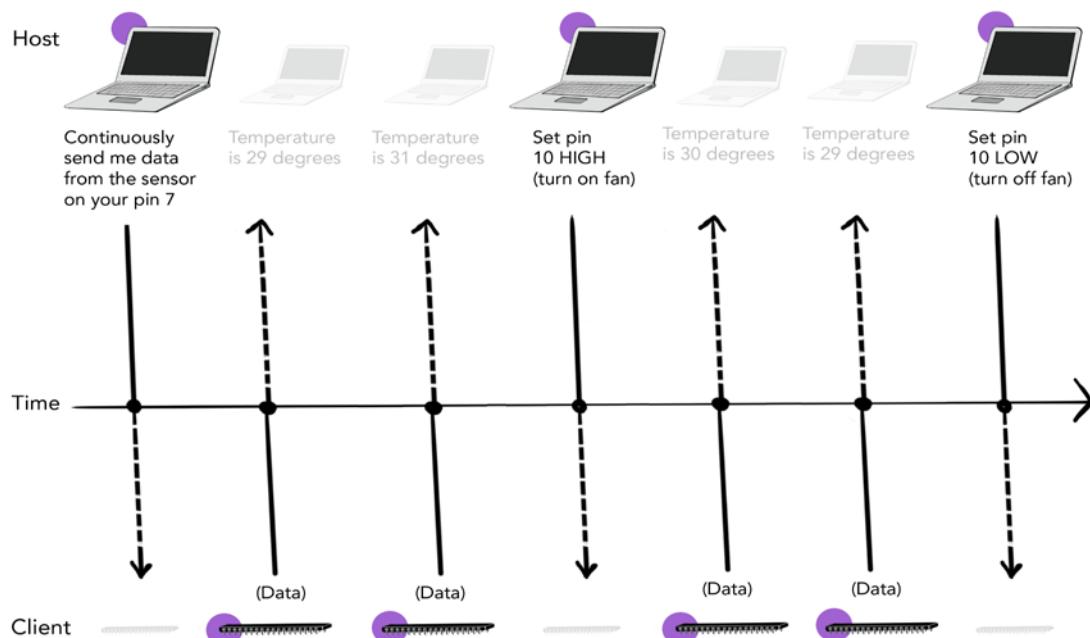


Figure 1.13 As the host executes the JavaScript logic, instructions and data are continuously exchanged between client and host, using a common API.

1.2.2 Embedded JavaScript

With *embedded JavaScript*, the JavaScript logic to control the project runs directly on the hardware's microcontroller.

Many microcontrollers aren't up to running JavaScript natively, but some are. As you'd expect with the march of technology, inexpensive microcontrollers are getting more advanced. It has become possible to run JavaScript, or an optimized variant of JavaScript, directly on certain embedded processors.

Each embedded-JavaScript platform is a combination of hardware and software ingredients working in tandem. On the hardware side, development boards up to the task of running code natively are based on more capable (but still cheap) chips.

Most platforms also provide a suite of software tools to complement their hardware. There may be a library or framework to use for writing compatible JavaScript code and a CLI (command-line interface) or other method for preparing the code and uploading it to the microcontroller.

Espruino (www.espruino.com) is an example of a JavaScript-based embedded platform. Espruino's flavor of JavaScript combines optimized core JavaScript with an API of hardware-relevant features. For example, you write code for the Espruino Pico board in a web-based IDE and upload it to the board via USB (figure 1.14). To adapt our automatic fan for an Espruino board, we'd need to write the logic using Espruino's API.

Another example of embedded JavaScript is the Tessel 2 (<https://tessel.io/>), a Node.js-based development platform. You can control and deploy code to your Tessel

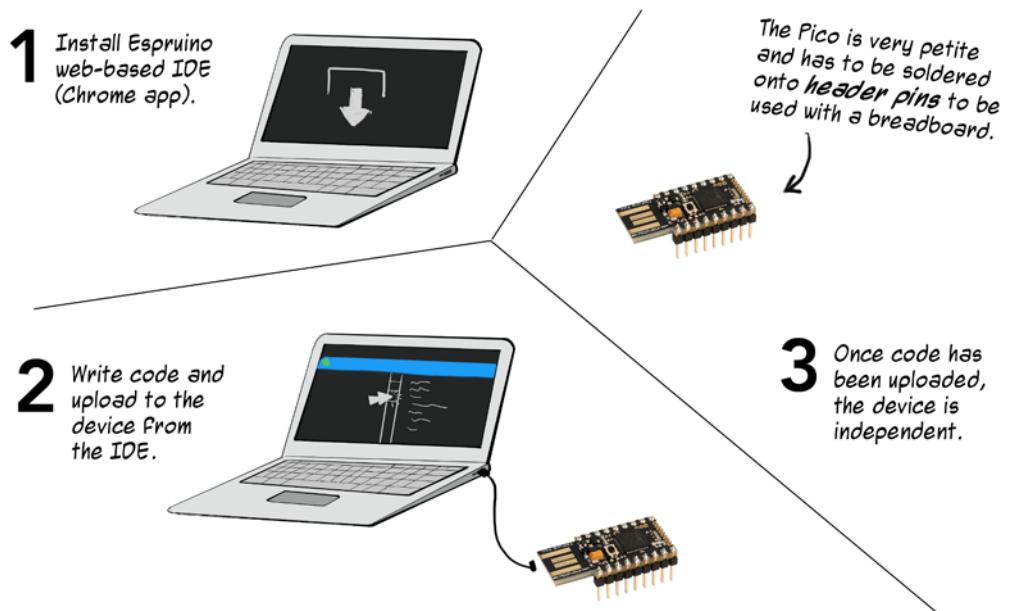


Figure 1.14 The Espruino platform combines small hardware boards with an IDE development environment.

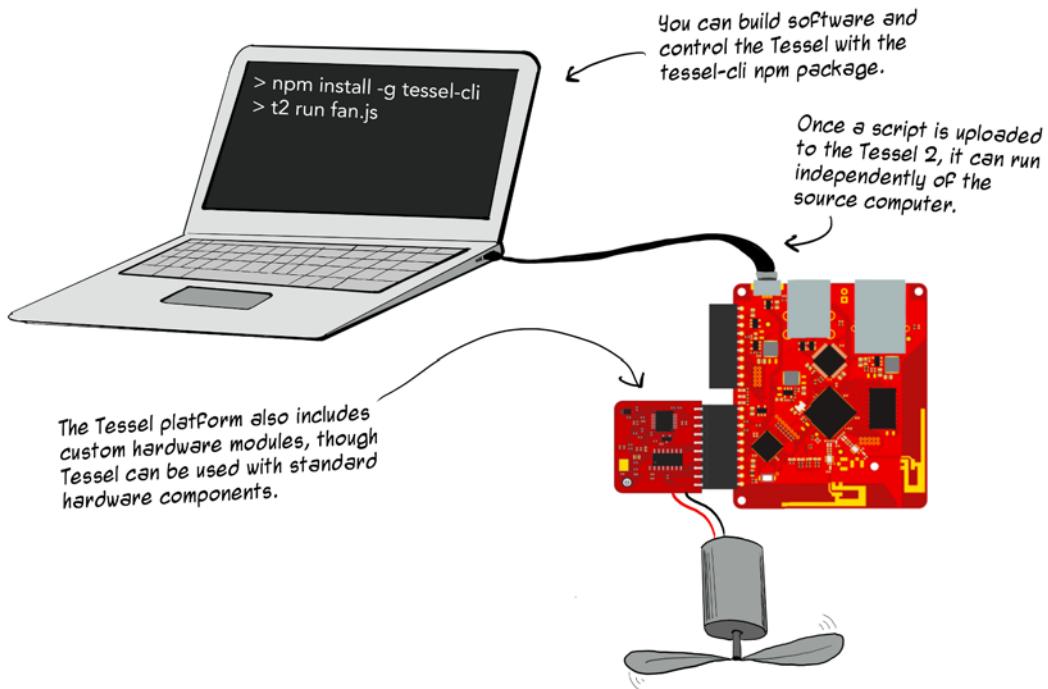


Figure 1.15 The Tessel 2 is an open source platform that runs Node.js natively.

using the `tessel-cli` npm module—wirelessly, if you like, because Tessel 2 has built-in WiFi (figure 1.15).

Being able to run JavaScript directly on embedded hardware can be power-efficient and self-contained. Projects are independent systems that can run on their own. Unlike the host-client setup, which requires firmware to translate from JavaScript to machine code, there are (usually) fewer layers of abstraction between your JavaScript and the hardware.

This sounds great, and you might wonder why we wouldn't use this approach exclusively. There are a few downsides. For one, there are fewer hardware options at the moment. Also, each platform has its own platform-specific techniques (software, tools, methodology), which can muddy the waters when learning hardware basics. Most also have certain limitations, either in JavaScript language feature support or in the types of inputs and outputs supported. But it's an inspiring method with a very bright future.

1.2.3 **Other hardware-JavaScript combinations**

Aside from the host-client method and running embedded JavaScript, there are a few other ways to combine JavaScript with hardware projects.

Tiny, *single-board computers* (SBCs) blend the host and the client into one unit. Cloud-based services make it possible to write JavaScript code online and deploy it wirelessly

to hardware. And emerging, new, and experimental features in web browsers themselves may offer a portal into the world of hardware for millions of web developers.

RUNNING JAVASCRIPT ON TINY COMPUTERS (SBCs)

Single-board computers (SBCs) like the Raspberry Pi family and BeagleBone Black can run full OS environments (typically Linux), and, by extension, Node.js. Instead of an 8- or 16-bit microcontroller, SBCs have higher-performance, general-purpose processors. But many SBCs also have I/O pins and capabilities built right into the same board (figure 1.16).

Using an SBC to control a hardware project blends aspects of both the host-client method and running embedded JavaScript. The processor has to continuously run the JavaScript logic for the project to work (as in the host-client model), but the whole package is contained on one board and feels more like an independent, embedded setup.

Unlike microcontrollers that run embedded JavaScript logic, though, the processor on an SBC doesn't run a single-purpose program—it can simultaneously run other processes.

These single-board computers are getting cheap. At this moment, there's the \$5 Raspberry Pi Zero (if you can get your hands on one—they're notoriously out of stock) and the WiFi-enabled Pi Zero W for just a tad more. There's no longer such a large cost differential between low-power microcontroller hardware and legitimate tiny computers with processors that rival tablets and smartphones.

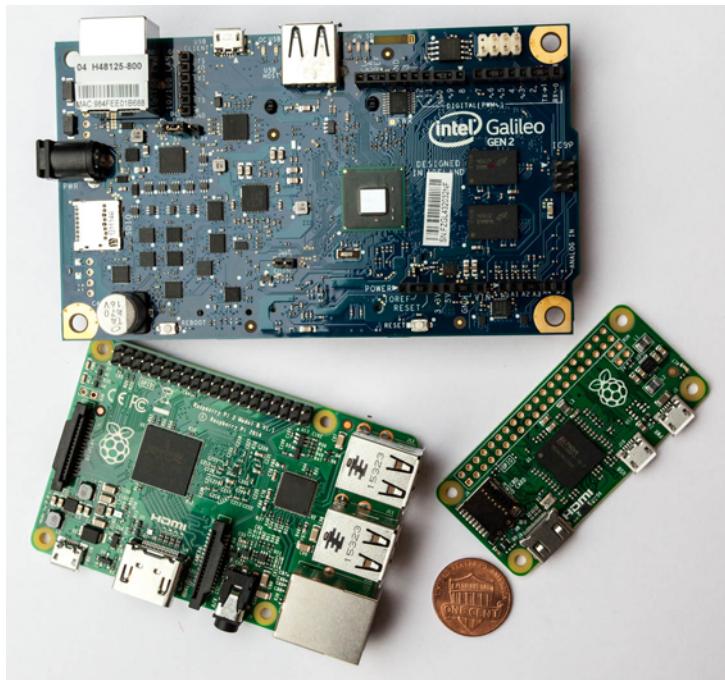


Figure 1.16 Several single-board computers (SBCs): Intel Galileo, Gen 2 (top), Raspberry Pi 2 Model B (bottom left), and Raspberry Pi Zero (bottom right)

Although running JavaScript on single-board computers with GPIO (general-purpose I/O) support gives you lots of options on one piece of packaged hardware, it has a few drawbacks. SBCs aren't as low-power as many microcontroller-based boards—the Raspberry Pi 2 Model B draws 4 watts. The SBCs we'll look at do have GPIO support, but the pin mappings and usage can be confusing and documentation sketchy or technical, which can be challenging if you're just learning about hardware hacking. You'll also need to be ready to face system administration hurdles, as the Linux distributions for SBCs, especially when combined with Node.js, can require some debugging and patience.

CLOUD-BASED SERVICES AND THE BROWSER

This last catch-all category for hardware-JavaScript combinations is admittedly blurry. Stuff's changing. *Fast*. The current growth of commercial, cloud-based services for the IoT has taken on the proverbial hockey-stick shape, and we're just seeing the very vanguard of advances that will let us directly interface with hardware from the browser itself.

Cloud-based services try to ease the complexity of managing fleets of IoT devices at scale. Many of these are targeted at the enterprise. Resin.io (figure 1.17), for example, builds, packages, and deploys containerized application code to provisioned devices, taking care of some of the security and automation headaches for you.

And then there's the browser itself, where many of the most cutting-edge hardware-JavaScript combinations are just starting to emerge. A few browsers already allow you to

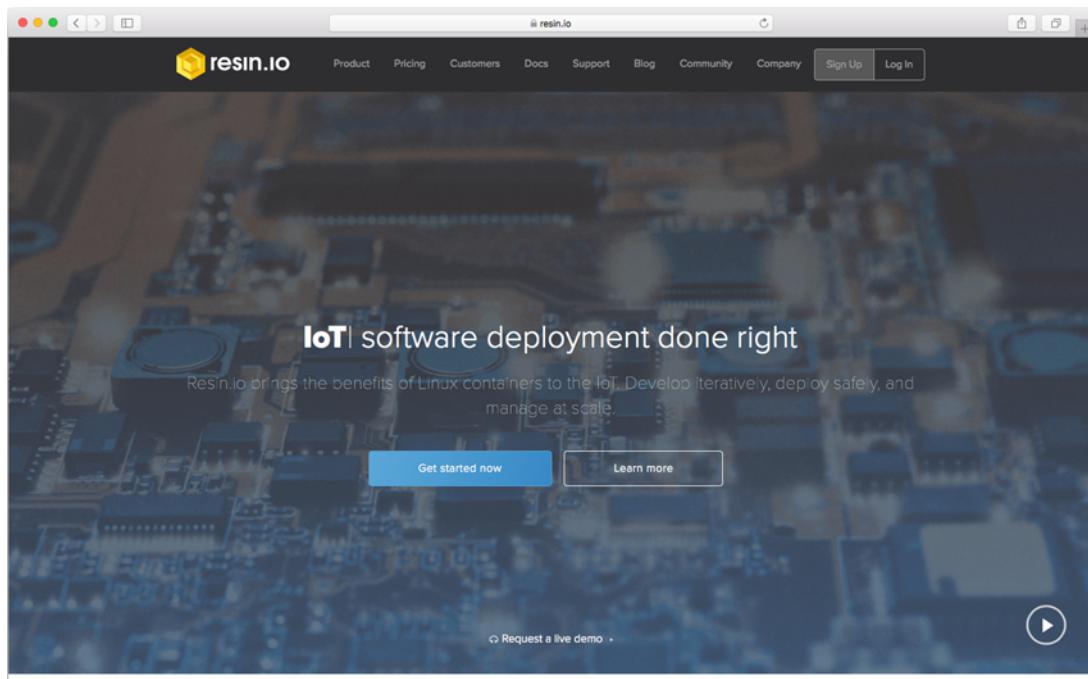


Figure 1.17 The Resin.io service helps to streamline application deployment to and management of Linux-capable SBCs.

experiment with Web Bluetooth, an API that, while not currently on the standards track, may be a harbinger of webby things to come. Web Bluetooth, as its name suggests, lets you connect to and control Bluetooth Low Energy (BLE) hardware, using JavaScript, from within the browser.

Another open project coming out of Google, the Physical Web, proposes an uncomplicated idea: give a small device the ability to broadcast a URL with Bluetooth Low Energy (BLE). A beacon used like this could transform a bus stop sign into a real-time arrivals tracker by broadcasting the URL to a web app with that information (figure 1.18). A simple concept, but flexible.

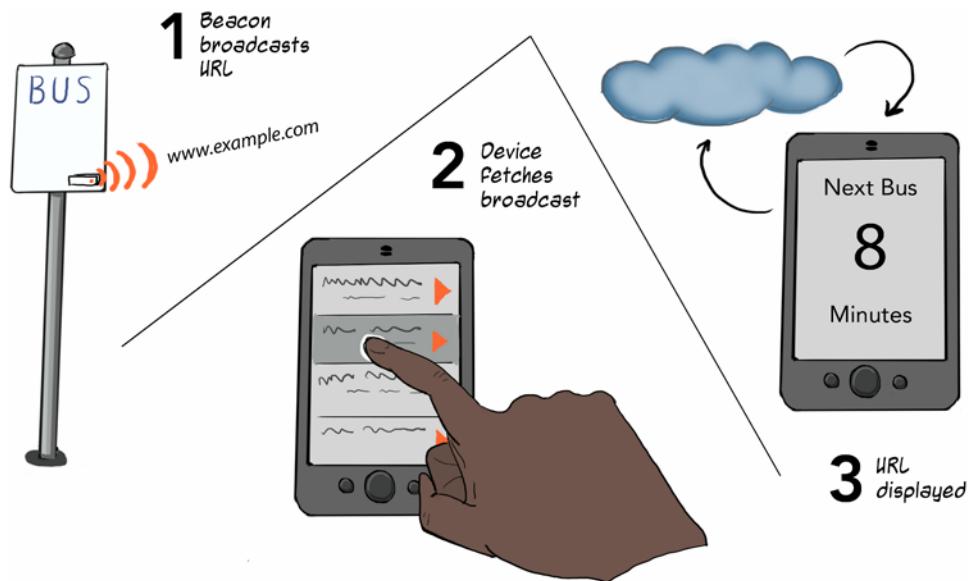


Figure 1.18 In this example application of the Physical Web, a bus stop sign uses a BLE beacon to broadcast a URL once per second (1); a human in the vicinity can scan for available beacons on their device and select the one corresponding to the bus stop (2); the bus-rider's device can now fetch the URL broadcast by the beacon and display it in the browser (3).

Of all the marriages between JavaScript and hardware, this variant—the deeper integration of the web with hardware—is the most volatile. It’s simultaneously intriguing and unpredictable. It’s likely that the demand for more ways to build IoT products with JavaScript will lead to head-spinning acceleration in this space.

1.3 Is JavaScript a good fit for hardware?

So maybe we can use JavaScript to hack on hardware in various ways, but should we? Is there utility here or is it just a self-indulgent parlor trick?

When the idea of using JavaScript with hardware first started surfacing a few years ago, it wasn’t met with universal enthusiasm. It was seen by some as arbitrary and misplaced cleverness—a *do we really have to use JavaScript everywhere?* weariness. Others

argued that the performance of JavaScript on constrained hardware would never be acceptable for anything but hobby use. A certain amount of old-guard crustiness surfaced, comment threads bogged down with passionate excoriations against anything but C/C++, and naysayers warned that a higher-level language would obscure essential low-level hardware nuances from newcomers.

And yet, there were many who remained open-minded. *Why use JavaScript when C/C++ is good enough?* had a curious echo of an earlier paradigm shift in hardware: *Why use C when assembly language is good enough?*

Whether it's awesome or it sucks—and we're not going to have that argument now—JavaScript is the de facto programming language of the internet. People know it, people use it, and it's everywhere. JavaScript's ubiquity gives it a unique potential to serve as a gateway for millions of web developers who sure would love to get going on the IoT.

Certain aspects of JavaScript programming lend themselves well to hardware, especially its proficiency at event handling and asynchronous processes. JavaScript is also a good tool for prototyping, a boon for fast iteration.

It's going to be fascinating to see where we end up. The JavaScript train is pulling out of the hardware station, and a lot of folks are jumping on for the ride.

1.4 **Putting together a hardware toolkit**

You've had a whirlwind tour of the ingredients that make up embedded systems and the methods of combining hardware with JavaScript. Let's now get more specific about the types of physical hardware, accessories, and tools needed to concoct these types of projects. Then we'll be ready to stock up a basic toolkit to get you started.

Our projects will combine a development board with input and output hardware. To build circuits and connect the systems together, you'll need supporting electronic components, as well as wires, power, and accessories. Throw in a few basic tools and you're ready to go.

1.4.1 **Development boards**

Development boards, also called *prototyping boards* or just *boards*, are physical development platforms that combine a microcontroller or other processing component with useful supporting features (figure 1.19). They're the bread and butter of the hardware-hacking lifestyle. Boards range in cost from just a few bucks to over \$100 for high-end SBCs.

Boards are centered around their brain, a combination of processor, memory, and I/O. 8- or 16-bit microcontrollers are at the center of straightforward, entry-level prototyping boards like (most) Arduinos (figure 1.20). Boards with more sophisticated 32-bit microcontrollers may be able to run embedded JavaScript.

Not all boards are microcontroller-based. More powerful SBCs are powered by components you'd normally find on a computer's motherboard. The architecture of these boards is accordingly more complex, involving one or more miniaturized systems on a chip (SoCs) and additional interconnects like HDMI, audio, or Ethernet.

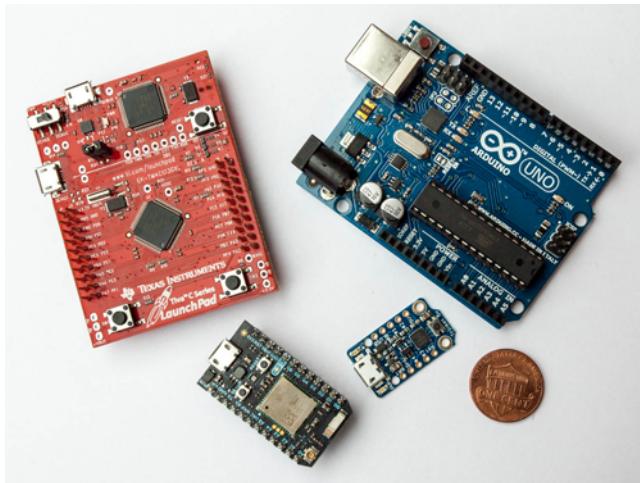


Figure 1.19 Some typical microcontroller-based development boards, clockwise from top left: a Tiva C-Series LaunchPad from Texas Instruments, an Arduino Uno R3, an Adafruit Trinket (5V model), and a Particle Photon

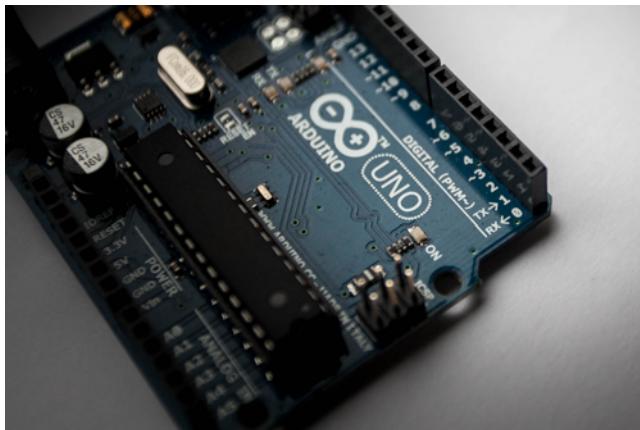


Figure 1.20 This Arduino Uno board is powered by the AVR ATmega 328-P, a 8-bit microcontroller.

Although SBCs may have physical I/O interfaces on-board—Raspberry Pis do, for instance—their general-purpose processors can as easily be put to use to power non-hardware-centric projects.

1.4.2 Input and output components

Oh, my, there are so many sensors and gizmos you can connect to your boards to enhance your projects! This is all sorts of fun, but it can also feel overwhelming at first. Lots of technical terms get thrown around, and there are lots of numbers, values, and specifications to absorb. You'll learn to find your bearings as you go through this book.

Most of the input and output components we'll work with are simple in design and ready to be plugged into a breadboard (that is, they are *breadboard-friendly*). Some are packaged as *breakout boards*. In the same way that development boards make I/O easier

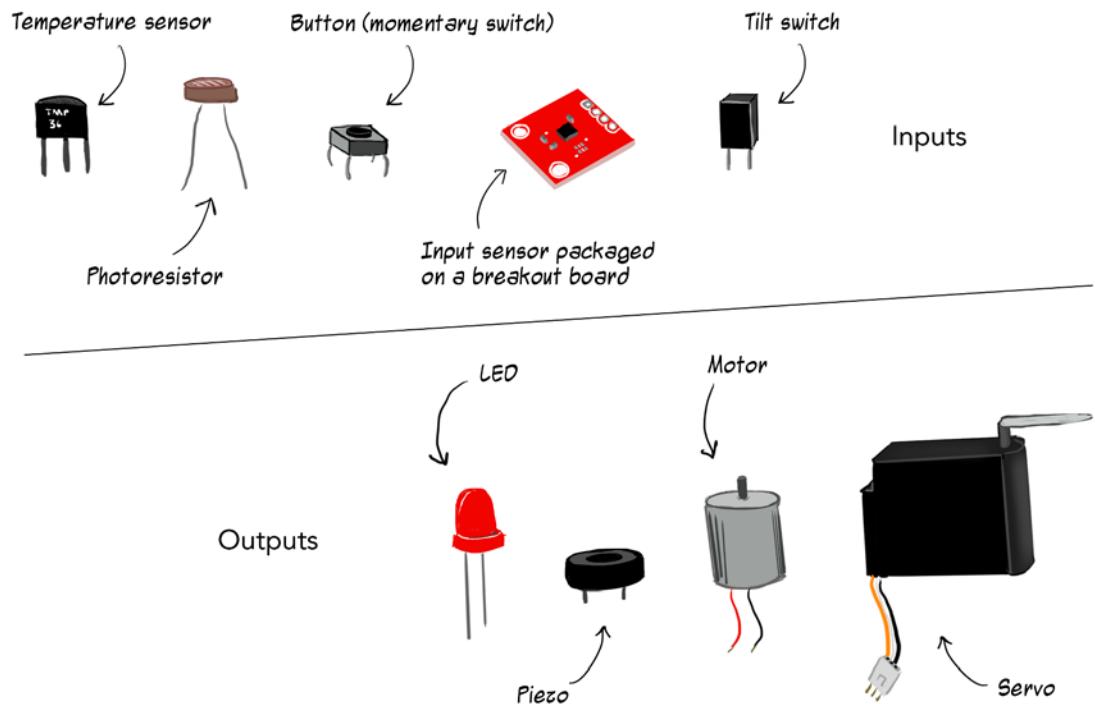


Figure 1.21 An assortment of common input and output components

by wiring a microcontroller's tiny pins to connections that are more convenient, breakout boards make it easier to work with single-purpose sensors or output devices by wiring their pins to more convenient connections (figure 1.21).

1.4.3 Other electronic components

Cobbling together electronic circuits requires a collection of supporting electronic components.

Although it can feel like there are a lot of little pieces, the basic components like resistors, capacitors, diodes, and transistors are inexpensive and can be bought in convenient starter kits (figure 1.22). We'll take our time to get to know these parts—soon they'll feel like old pals.

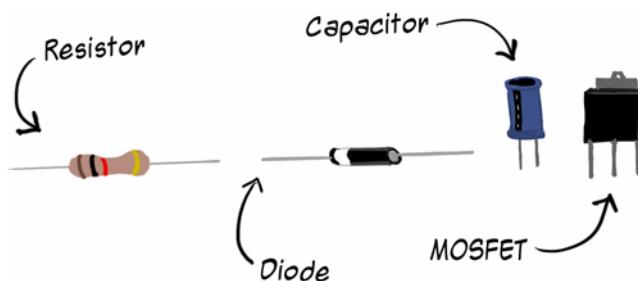


Figure 1.22 Common components like these will help you build functional electronic circuits.

1.4.4 Power, wires, and accessories

One thing you'll soon realize is that there are a whole lot of ways to power a project!

Development boards can be powered over USB or by plugging them into a DC adapter (wall wart). In many cases, other project components can take advantage of that same power source (figure 1.23).

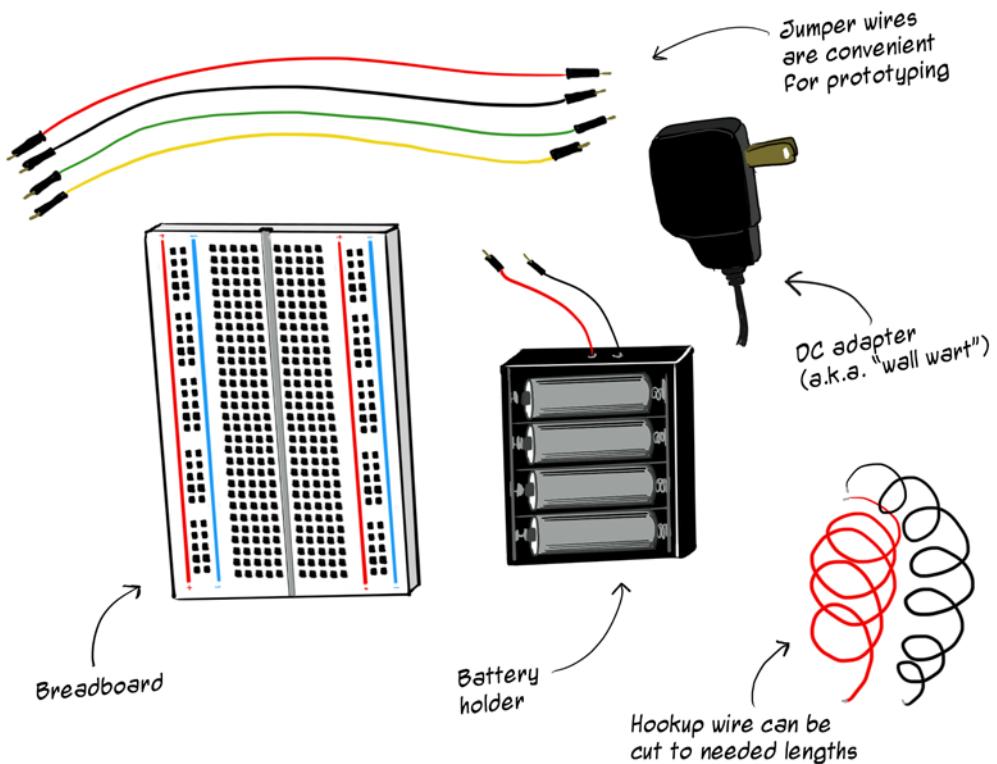


Figure 1.23 A sampling of wires and accessories for power and circuitry

Batteries are useful for making projects wire-free, as well as for providing additional power at different voltages. There are many kinds of battery *snaps* and holders for connecting batteries to projects.

To connect stuff together, you'll need wires. *Jumper wires* are precut wires. One particularly handy variety has pins on each end that slide easily into breadboards and the I/O pins on many boards. Jumper wires are great for quick prototyping. Alternately, *hookup wire* usually comes on a spool and can be cut to specific lengths as needed.

1.4.5 Tools

A pair of needle-nose pliers and a precision screwdriver or two are useful companions when building projects. You'll want a pair of wire strippers—which usually have built-in

wire cutters—if you’re cutting or stripping hookup wire (precut jumper wires don’t need to be cut or stripped). As you progress, you might want to get your hands on a *multimeter*, a tool for measuring voltage, current, and resistance.

STORING YOUR ELECTRONIC COMPONENTS As you start building projects, you’ll end up with a lot of small parts. You can find compartmentalized storage boxes or drawer units at hardware and hobby stores. Boxes and cases designed for fishing lures can make especially handy containers for electronic parts because their compartments are small and their dividers fit snugly (figure 1.24).

It’s time to start our journey. Hacking with low-power embedded hardware can be fun, creative, and exciting—and it’s increasingly useful in the commercial world. Web

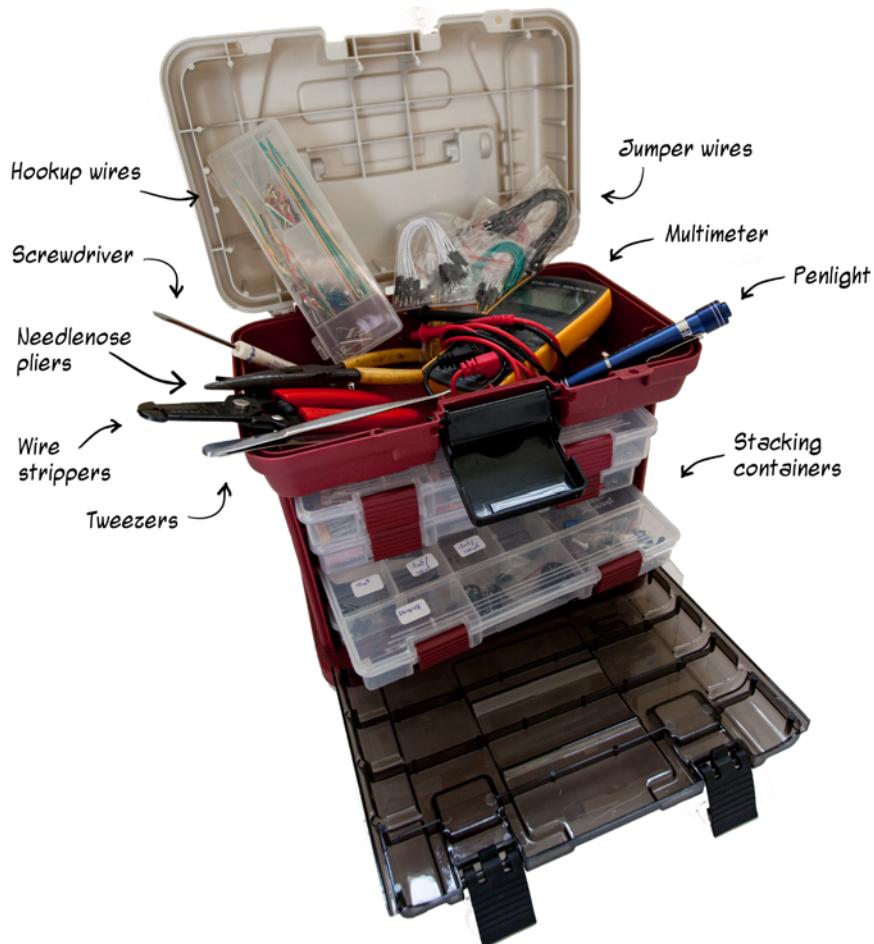


Figure 1.24 This compact tackle box has space in the top for tools, and storage for components in stacking, removable containers.

developers (like you) already have skills that can be great stepping stones on your path. You can use the ubiquitous language of the web, JavaScript, to get you going and reduce roadside distractions.

On our adventure, you'll get a fundamental understanding of the few basic relationships that make electronic circuits work. Not mathy? Don't fret, neither am I. You'll meet some helpful characters on the way: components and modules, different kinds of boards and software. We'll try out different combinations of things and learn how to dust ourselves off and try again when we blow up an LED.

The road goes on forever, the horizons are infinite. We won't be able to visit it all, but by the end of this book you'll be prepared to assess and use future technologies that haven't yet dawned. By the time you're midway through your travels, it's likely the road you set out on will have changed remarkably. But by relying on some constants as your compass—hardware basics, the application of JavaScript, web technologies—you'll be able to find your way.

Summary

- Starting from scratch on an embedded-electronics hobby can feel intimidating, but your existing JavaScript skills can give you a boost.
- Embedded systems combine a brain—a microcontroller or power-efficient processor—with inputs and outputs in a small package.
- A microcontroller combines a processor, memory, and I/O in a single chip. Logic defining the behavior of a microcontroller—the firmware—is typically flashed to the MCU's program memory.
- There are several ways JavaScript can control hardware: host-client, embedded JavaScript, Node.js on SBCs, and even from within a browser.
- In a host-client setup, Node.js executes on a host computer, and instructions and data are exchanged with the microcontroller using a messaging protocol (API). The project can't function without the host computer.
- Some constrained microcontrollers are optimized to run JavaScript (or a subset of JavaScript) directly on the chip (embedded JavaScript).
- Single-board computers (SBCs) have more sophisticated processors and additional features, like USB ports or audio connections. These devices can usually run full-fledged OSs and often behave like tiny computers. Many give you the option of controlling I/O and behavior with higher-level languages like python, C++, or JavaScript.
- Development boards are platforms combining a microcontroller (or other processing component) with handy supporting features. They provide convenient connections to I/O pins, allowing for quick prototyping of projects.
- Building projects involves a certain amount of electronic gear: development boards, input and output components, basic electronic components like resistors and diodes, power connections, and basic tools.

JAVASCRIPT ON THINGS

Hacking hardware for web developers

Lyza Danger Gardner

Are you ready to make things move? If you can build a web app, you can create robots, weather stations, and other funky gadgets! In this incredibly fun, project-based guide, JavaScript hardware hacker Lyza Danger Gardner takes you on an incredible journey from your first flashing LED through atmospheric sensors, motorized rovers, Bluetooth doorbells, and more. With JavaScript, some easy-to-get hardware, and a bit of creativity, you'll be beeping, spinning, and glowing in no time.

JavaScript on Things introduces the exciting world of programming small electronics! You'll start building things immediately, beginning with basic blinking on Arduino. This fully illustrated, hands-on book surveys JavaScript toolkits like Johnny-Five along with platforms including Raspberry Pi, Tessel, and BeagleBone. As you build project after interesting project, you'll learn to wire in sensors, hook up motors, transmit data, and handle user input. So be warned: once you start, you won't want to stop.

What's inside

- Controlling hardware with JavaScript
- Designing and assembling robots and gadgets
- A crash course in electronics
- Over a dozen hands-on projects!

Written for readers with intermediate JavaScript and Node.js skills. No experience with electronics required.

Lyza Danger Gardner has been a web developer for over 20 years. She's part of the NodeBots community and a contributor to the Johnny-Five Node.js library.

To download their free eBook in PDF, ePUB, and Kindle formats, owners of this book should visit www.manning.com/books/javascript-on-things



“JavaScript breaks free of the browser with this book!”

—Amit Lamba, Tech Overture

“The most accessible and enjoyable book on IoT tech I have ever read.”

—Andrew Meredith
Quantum Metric

“Calling all developers!
Unlock your inner hardware hacker with this exciting book.”

—Kevin Liao, Sotheby's

“Makes developing low-level circuits fun. A pleasure to read!”

—Earl Bingham
You Technology

ISBN-13: 978-1-61729-386-3
ISBN-10: 1-61729-386-5

53999



9 781617 293863

 **manning** US \$39.99 | Can \$52.99 [Including eBOOK]