O'REILLY®

Connecting Networked Devices

Prototyping and Scaling IoT in the Enterprise

Alasdair Allan
& Brian Jepson

ThingWorx is purpose-built for the Internet of Things, with tools, APIs, and marketplace extensions that lower costs, increase developer productivity, and speed time-to-market.

With the ThingWorx IoT Platform, you have access to a powerful development engine and a broad set of innovative technologies that extend the power of the IoT:

- CONNECT to any Thing
- CREATE Apps for all Users
- ANALYZE Machine Data
- EXPERIENCE all Things through Augmented Reality

Learn more about how the ThingWorx IoT Platform is the right choice to power your organization's digital transformation.

thingworx.com/go/ConnectedDevices

ptc

Connecting Networked Devices

Prototyping and Scaling IoT in the Enterprise

Alasdair Allan and Brian Jepson

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Connecting Networked Devices
by Alasdair Allan and Brian Jepson

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian Jepson
Production Editor: Shiny Kalapurakkel
Copyeditor: Jasmine Kwityn
Proofreader: Amanda Kersey

Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Panzer

November 2016: First Edition

Revision History for the First Edition
2016-11-10: First Release

978-1-491-96404-0

[LSI]

Table of Contents

Preface vii

1. Where Does Your Product Sit? 1
Project Requirements 1
Prioritizing Your Decisions 2
Designing the Minimum Viable Product 3
The Standards Problem 3

2. Starting with One 7
Know Your Device's Role 8
Don't Fall in Love with Your Parts Bin 9
Creating a Bill of Materials 10
Code and Hardware 11
What About the Network? 12

3. Your Developers' User Experience 13
The Two Platforms 13
You Won't Program to the Platform 16
Talking to the Cloud 17

4. The Physical Environment 19
Physical Environment 19
Enclosures 21
Power Requirements 22
Deep Sleep and Duty Cycle 22
Connectivity 23
Storage 24

5. Security Is Your Job. 25
A Unique Security Problem 25
Authentication and Authorization 26
The Internet of Things and the Industrial Internet 26
Broken Firmware 27
Reverse Engineering the Hardware 29

6. Time to Market Versus Common Sense. 33
Off-the-Shelf Components 34
What About the Prototype? 34
Managing Risk 35

7. Conclusions. 37

Preface

We're accustomed to yearly, or faster, upgrade cycles. A new phone each year, a new laptop every couple of years, and each year the slabs of aluminum, plastic, glass, and silicon get a little bit thinner and weigh a little bit less. And from that shift, even smaller devices have begun to appear, and everyday objects are becoming smarter.

Because of this, computing is slowly diffusing into our world. A decade from now, everything important will be measured and observed in real time. But right now almost all of the devices that will one day will be connected to the network aren't.

When most people think about big data, they think about it living out in the cloud. However, at the moment, the amount of data that lives on systems that aren't connected to the network, or are unreliably connected to the network, vastly outweighs the data that lives in the cloud. As more smart, connected devices come online that can push that data to the cloud, all this data which had previously been locked away will become available.

Right now, this infrastructure is being built, and chances are good you're one of the people building it. As you connect networked devices—to one another and to the cloud—you'll need to consider many factors, including your product's relationship with its environment, how your prototyping process considers its constraints, your developers' user experience, and the physical operating environment. And you'll do this all while balancing time to market, common sense, and security.

Products, Platforms, and Strategies

Over the last few years, many companies have introduced platforms and products designed for the Internet of Things (IoT). These products often are either just proprietary middleware and services, or are simply embedded electronics, with or without the addition of a network connection. Just adding a network connection to an existing device doesn't make it part of the Internet of Things.

In an environment that is rapidly changing, and will continue to be volatile for several more years before best practices, or even just accepted practices, start to emerge, designing your product will depend heavily on a number of factors: naturally, the requirements are important, but there's also the refresh cycle, time to market, physical environment, and how the device will connect to the network. One of the key problems in the current generation of Internet of Things devices is the refresh cycle problem. Enterprises reap the benefits of two- to three-year refresh cycles in computers and mobile phones; with new hardware and software come bug fixes for security. IoT devices are generally part of systems that are expected to run for many years, even decades, with minimal intervention (and minimal opportunity to perform software updates).

The design of your product will be heavily influenced by the typical refresh cycle in your area of business, and governed not just by the time to market, but also the amortization of nonrecurring engineering costs, and lifetime units sold.

The physical environment into which the product will be placed must also be taken into account. Computers, and network connected devices, no longer live in the server room—they're out in the world interacting with dirt, dust, water, and people.

That change in location also changes the way the device must be powered and connected to the network; in addition, it affects the options you have available for connecting a networked device and determines how you will exercise those options.

CHAPTER 1

Where Does Your Product Sit?

The dot-com revolution happened in part because, for a few thousand or even just a few hundred dollars, anyone could have an idea and build a software startup. Today, for the same amount of money, you can build a business selling goods or software, and both your product and the process by which you develop it are augmented because devices and people can communicate and connect across the globe.

Project Requirements

Everything begins with how users will interact with your device, and how it will interact with both them and the network. In other words, where will your product sit in the hierarchy of connected devices?

Local, Edge, or Cloud?

In general, connected devices can be roughly split into three broad categories: local devices, edge devices, and cloud-connected devices.

Local devices don't have the capability to reach the Internet, but they are still connected in a network. This network usually isn't TCP/IP based. For instance, both ZigBee and Bluetooth LE devices are good examples of network-connected things that operate locally rather than directly connected to the Internet, and illustrate the two types of local networking. In the case of ZigBee, the device operates in either mesh or peer-to-peer mode, with packets hopping between devices until they reach a gateway on the edge of the local network.

In the case of Bluetooth LE, the device operates in either paired or broadcast mode, with messages being picked up directly by a device on the edge of the network.

Edge devices typically have multiple radios, and operate in both local and connected modes—for instance, utilizing ZigBee or Bluetooth LE to talk to a local non-TCP/IP network, but also fully supporting a TCP/IP connection to the outside world. They act as a bridge, or gateway, between a local network and the outside world, typically forwarding data received from a local network of sensor devices to the cloud, or sending commands to a network of actuators from the cloud.

Cloud devices connect directly to a TCP/IP network, in most cases using WiFi or cellular, and wired devices also count in this category. They're distinct from edge devices in that they typically don't communicate via a local network to other network-enabled devices. If they are part of an extended network of smart, connected devices, all the communication is normally funnelled via the cloud.

Prioritizing Your Decisions

Although it's tempting to imagine that a single protocol and network stack will come to dominate the IoT, it is in fact unlikely that this will happen. The history of the Internet suggests that the IoT will remain diverse in the protocols and architectures used.

If you want to deploy devices in massive quantities, cost management means it is likely that these devices will have the minimum viable specification that is required to do the task at hand. The purchase decisions you make may constrain future options as far as which protocols are available to you.

It's possible that some convergence will happen at the highest level of the protocol stack, with the data exchange formats tending toward a single standard, while all the underlying transport protocols remain diverse. This is almost the opposite of the existing digital Internet, where a diverse number of low-level networking protocols have been effectively replaced with TCP/IP, layered on top by a large number of other higher-level transport protocols and document standards.

Designing the Minimum Viable Product

A *minimum viable product* (MVP) is a product (rather than a prototype) that you deploy to customers, with the minimum set of features you need to gather information on how it is used so that you can refine it. Typically, you'd roll out the MVP to a small group of customers, and for each major set of refinements, expand that group. In the software world, this approach is popular and successful because of the relative ease of updating software in a cloud-based world.

However, the story of hardware development is somewhat different. While the concept of a minimum viable product still exists and is useful, building hardware tends to led by design rather than by feature. That results in two prototypes being built, a "looks like" and a "works like" prototype. The "looks like" model, true to its name, looks like the final product, but has little—or sometimes none—of the intended functionality. The "works like" prototype behaves like the final product, but generally bears no outward resemblance to the final product in terms of industrial design.

Confusing the "looks like" and "works like" prototypes, or mixing them, leads to what's commonly referred to as feature-driven design. For hardware products intended for the IoT, this leads to poor user interaction with the product. That's caused by design decisions you didn't make, instead offloading design decisions to the user. If you make these design decisions once, users won't have to go through that decision tree every time they use the product.

The Standards Problem

As we alluded to earlier, there is a brewing—if not all-out—standards war ongoing in the IoT. There is a great deal of confusion around protocols at different levels of the networking stack. For instance, there is a fundamental difference between low-level wireless standards like ZigBee or WiFi, and much higher-level concepts such as TCP/IP or above that HTTP and "the web" which sits on top of the TCP/IP networking stack. Above even that are document-level standards like XML and JSON, and even more conceptually wooly things such as patterns.

For instance, the concept of RESTful services is effectively a design pattern built on top of document- and high-level networking

protocol standards. It is not in itself fundamental, and is unrelated to the underlying hardware, at least if the hardware itself is capable of supporting an implementation of these higher-level protocols.

However, perhaps the greatest standards problem with the IoT is that, due to constraints in power or computing resources, it is a mess of competing and incompatible standards at the lowest level. Factors such as range, data throughput requirements, power demands, and battery life dictate the choice from a bewildering array of different wireless standards.

The Big Three

The big three, the most familiar to consumers and to developers, are Bluetooth, WiFi, and cellular.

The most obvious choice, perhaps even the default choice, for networking an IoT device is the WiFi standard. It offers good data rates, and reasonable ranges (of the order 150 feet or larger), and means your device can connect directly to the Internet if needed. However, WiFi devices with even moderate duty cycles are power hungry.

Bluetooth, especially the low-energy configurations intended for low data rates and limited duty cycles, is designed for personal (wearable) devices with limited ranges and accessories. While recent standards revisions include support for direct Internet access via 6LoWPAN (IPv6), there is still only limited support that effectively means that Bluetooth devices are restricted to local, and small, networks spanning (despite manufacturers claims) around 30 or 50 feet. In the shortest-range use cases (a few inches), you should also be looking at near-field communication (NFC).

Of the three, perhaps the most ubiquitous, with the widest deployment and market penetration, is cellular. If your cell phone can get signal in a location, so can an IoT device with a 2G or 3G module onboard. Data rates lie somewhere between WiFi and Bluetooth, with the main advantage being range. Cellular devices can be located up to 20 miles from a cell tower, and depending on intervening obstacles, still get reception. However, cellular is both power hungry and expensive to operate. While GSM may be a good choice for a gateway device, it's unlikely to be a good fit for most IoT devices.

Mesh Networks

Standards such as ZigBee and Z-Wave fill a niche in the local networking space. While they need a gateway device to talk to the Internet, both standards have mesh networking capability, so while individual devices can have between 30- to 300-foot range, the size of the network is actually only limited by the number of devices deployed. Both ZigBee and Z-Wave are targeted for low-power applications with lower data rates.

While ZigBee and Z-Wave have been around for a while, newer IPv6 protocols, such as Thread, which are based around a basket of standards including 6LowPAN, offer mesh networking and direct access to the digital Internet so long as IPv6-capable networking gear is in place. Designed to stand alongside WiFi, IPv6-based protocols such as Thread are attempting to address the lack of TCP/IP at the lowest levels of the IoT, accepting that the high-powered WiFi standard may be inappropriate for many (if not most) IoT devices.

Wide Area Networks

While cellular (typically GSM or UMTS) is the most popular standard used to provide wide area networks (WAN), there exist other newer standards that attempt to provide this functionality at much lower costs.

Sigfox uses the ISM radio bands, with typical operational ranges of up to 30 miles in rural environments and as low as 6 miles or less in urban environments. Sigfox networks are being rolled out in major cities across Europe and the United Kingdom. Making use of ultra-narrow band signaling, it is intended for very low data rates (just 10 to 1,000 bps) but also very low-power deployments (consuming a factor of 100 less power than cellular).

Other alternatives include Neul, which leverages the small slices of whitespace spectrum between the allocated TV bands and has a range of around 6 miles, and perhaps the most well known of the three, LoRaWAN.

LoRaWAN has a range of up to 3 miles in an urban environment and perhaps 9 or 10 miles in a suburban environment. Its rates range from 0.3 kbps up to as high as 50 kbps, and it makes use of various frequencies, depending on deployment. Much like Sigfox, it is also optimized for low-power deployments and large (millions of

devices) deployments. The first LoRa network with nationwide coverage was rolled out in June 2016 in the Netherlands by the Dutch telecoms group KPN.

CHAPTER 2
Starting with One

When you're building a connected device, you're building for a lot of different people. But no matter how someone uses your device and its associated stack of services, your users will fall into one or both of the following roles:

Users
: These are the people who interact directly with the device or its data. They need the device to be on and available when it's supposed to be, and they expect the same for its data.

Developers
: Every device needs some degree of integration and ongoing maintenance. The developers take your device (and its APIs) and connect it into their internal or customer-facing systems. They might be full stack hardware developers, or they might work at one of the levels in the stack: from firmware customization to adding new peripheral devices, on up to developing cloud analytics to work with the data, and anywhere in between.

A third role—hardware power users—emerges when someone falls into both of the preceding categories. Hardware power users might be users 80% of the time, but use scripting or plug-in tools to customize their experience.

Each role needs a different set of affordances and features, and will have different expectations about response time and access. A user might need smartphone access to a dashboard that's updated every second, but she doesn't need to access the device's serial port

remotely. A developer might need the ability to push over-the-air firmware updates to a device, but he doesn't need to receive notifications when it's time to replace a refrigeration unit that his device is monitoring.

However, your device has the weight of the world on its shoulders. It has to support all those scenarios, and look good doing it.

A prototype often starts with a breadboard, some components, a microcontroller board, and some ideas. It's tempting to throw every feature imaginable into the device. But just because something is *possible* it doesn't mean it is compulsory. Adding extra controls or displays to a product is a classic mistake at this stage of the product life cycle. It increases the cost of each unit and introduces cognitive overhead the users and developers must bear.

Over the last few years, we've finally reached a point in the software design world where we've figured out that offering choice to the user isn't always the best thing to do. More choice isn't a virtue in and of itself, and sometimes it can introduce confusion. Every control you add—to software or hardware—is a design decision *you* aren't making, where you're offloading the design of your interface onto the user. Make the decision once so your user doesn't have to make the decision several times a day.

Know Your Device's Role

Where does your device sit? What is its role? Edge device or gateway? More and more, the answer is both.

As connected devices are increasingly deployed into every corner of our environment, the need grows for them to be self-sufficient. More and more wireless modules, such as those from Espressif and Nordic, have a microcontroller that can run custom code and interface directly with sensors over GPIO, I2C, and SPI.

Sensors can also gather a tremendous amount of data. With a low-bandwidth or limited-data connection such as a cellular connection, it's ideal for the device to do as much processing on the data as possible before relaying it to the cloud.

As you develop your prototype, you'll understand its role better. How much will your device do on its own? Will it just relay raw sensor readings to a central gateway device? Will it read sensor

readings, perform some processing, and then relay it to a gateway? Or is the device itself the gateway, reading sensors, processing data, and sending it on to the cloud independently?

The answers to those questions will come from the physical (and software) environment you are deploying in as much as which hardware choices you make.

Don't Fall in Love with Your Parts Bin

For the prototype, where you're just throwing components onto a breadboard as fast as possible, it's sometimes convenient to use whatever you have on hand. That can be a good thing if you're an experienced product engineer, and your parts bin consists of things that you've used in other successful projects.

But new components, whether new generations of existing parts, or entirely new parts, become available quite regularly. When Espressif introduced the inexpensive ESP8266, it was a big deal for prototypers. Here was an inexpensive module ($2 in single-unit quantities) that not only could be used to add WiFi networking to a prototype or a device, but could itself be programmed. In many prototypes, the $2 ESP8266 has enough I/O and processing power to completely replace both an Arduino and WiFi module. This device came to attention in mid-2014, and by mid-2016, its use was widespread.

In addition to keeping on top of new and emerging components, you need to maintain reliable standbys in your parts bin. Because choices made at the start of engineering design have a tendency to become set in stone as the later prototypes evolve, you can't afford to prototype with a component that has reached end-of-life. Therefore it's important, especially if you're building a hardware product for the first time, to fill your parts bin with parts that are easy to source. Readily available parts are more affordable, resulting in a lower bill-of-materials cost, and can be sourced from multiple manufacturers if needed, resulting in a more reliable supply.

Several contract manufacturing companies have established parts catalogs to advise your parts choices. For example, SeeedStudio has compiled an Open Parts Library (OPL) catalog, which is a collection of the most commonly used components. If you're working with Seeed, using OPL components in your design means that Seeed will

have sufficient stock to fulfill on short lead times, at lower costs than other components.

Other sites, such as Parts.io, exist to simplify the component discovery process. Using these types of sites when considering your parts choice means that you can manage exposure to supply chain risk. You can get feedback not just on the availability of the part, but also the life-cycle stage of the component, and variation in cost over time and between competing suppliers.[1]

When assessing whether a component should be designed into a product in the first place, it's important to consider not just whether it's available now, but whether it'll be available over the entire life cycle of your product. While some components may only have a single source, you should always try to find a more common substitute. Supply volatility of components can leave you unable to find a critical component, which could lead to your product being retired from the market early or a costly mid-life-cycle redesign of your product.

Creating a Bill of Materials

You may start most products with a single model, but eventually, you'll need to make many. Whether a dozen, one hundred, or one million, you're going to need to make more than one.

The critical starting point for mass production is your bill of materials, which serves as the list of ingredients for building the product.

From the information contained in the bill of materials, it should be possible to determine the lead time to procure the materials necessary for production, the manufacturing processes necessary to bring them together, and the time to manufacture and ship the product.

While most companies understand the importance of the bill of materials, there is little consistency in format, and that can add friction when dealing with contract manufacturers. To combat this, there have been several attempts[2] to produce standard bill-of-materials templates (e.g., the Dragon Standard BOM4) and, especially for first-time product builders, these can be invaluable.

1 For more information, see Parts.io's blog post "Improving RiskRank."

2 See Dragon Innovation Blog's post "Introducing the Dragon Standard BOM".

Code and Hardware

In recent years, hardware has come to be seen as "software wrapped in plastic." While it's not a popular view with hardware engineers, these days the code running on your hardware can be just as important if not more so than the hardware itself. Like design, in an age where all things that are buildable are rapidly copyable, the software running on your device may prove to be even more important than the hardware it runs on.

You should also consider the possibility that your code can be used again and write your code for reusability. If you're building one product, it's likely you may build a second version, or a third. As such, it's important to separate the code that talks directly to the hardware of your device—the dials, sliders, buttons, knobs, and other physical things—from the code that talks to the network.

With the drop in the cost of computing—both in terms of price and power consumption—many manufacturers are now basing their connected devices around relatively powerful processors. A common pattern emerging in this space is to have the code that talks to the hardware running as a separate process from that which implements the functionality of the networked device. This code, which may be written in a much lower-level language than the application which manages the rest of the connected device, often uses a REST or other network-level API to talk to the management code.

This means that the bulk of your device code can be written in a higher-level language, decreasing developer time and increasing your pool of development talent. But it also means that the management code can expose command and control functionality in just one place. The same API used by the device's own hardware can in turn be exposed by the device's network interface and used to control it from some other network device.

What About the Network?

As you develop your prototype, you will make decisions about how to connect to the network, and how to move data from device to cloud. If your device is both an edge device and a gateway, and is connected directly to a WiFi, wired Ethernet, or cellular network, you'll be in well-worn territory.

But if you're not directly connected to a network, and you need to use a gateway of some sort, you'll have to decide how that's handled:

- Are the devices clustered together? Maybe adding Bluetooth or ZigBee to your gateway and edge devices is the answer.
- Are the devices spread out all over the place? Maybe LoRa or cellular is the way to go here.
- What is the ratio of gateways to devices? If you have fewer highly mobile devices, you might be scaling up the number of gateways rather than the number of devices.

You may start with only one, but you will scale to many. And even when you're done with the first, it's not unreasonable to think that new versions and new products will soon follow. This chapter shared some guidance on the trickier parts of the process: making sure your product is approachable by the different developer and user roles, understanding your device's relationship to systems large and small, the parts that actually go into your device, and the software and network connectivity considerations you need. The next chapter looks in more detail at the relationship between your device and its users and developers.

CHAPTER 3

Your Developers' User Experience

Chapter 2 talked about reconciling the two main constituencies of your device: users and developers. This chapter digs down more into the developer experience and looks at how your prototyping decisions affect your future developers. A connected device is rarely just a black box. At a minimum, your customers will need someone to integrate that device into their system. But as more computational power comes to the devices you create, there's more opportunity to enable customers to customize well beyond the initial feature set you envisioned for your product.

The Two Platforms

There was a time when, if you wanted to build a connected device, you'd need to research all the chip vendors, spend hundreds of dollars on evaluation boards, sign nondisclosure agreements to get access to SDKs, and then swim upstream into the vendor's sales funnel when you needed any kind of guidance developing your prototype.

This old way of doing things was turned on its head with the arrival of the Arduino, an inexpensive microcontroller board for prototyping that allowed anyone with an idea, a modest understanding of electronics and programming, and motivation to create a connected networked device.

More than 10 years later, and Arduino will forever be remembered as the platform that launched the desktop 3D printing revolution.

Without Arduino, the creators of 3D printers such as MakerBot, RepRap Prusa, PrintBot, Ultimaker (and many more) would have had to roll their own controller boards for their motors. This early success has validated Arduino as a robust platform for creating connected devices.

And not too long ago, Arduino was joined by a board with very different capabilities: the Raspberry Pi. The Pi became the perfect complement to the Arduino. Where Arduino projects run on the bare-metal CPU, Raspberry Pi boards come with an operating system, Linux, Windows IoT Core, and others. Arduino and Raspberry Pi complete the platform level of the full hardware stack.

Unlike the Arduino, the Raspberry Pi was never really designed as a platform to be used by developers. However, at $35, it made single-board computing accessible, and it was months after its release before supply caught up with demand. The release of the Raspberry Pi Zero, at $5, was met with a similar rush. Supply still hasn't caught up with demand. While not optimized for development, the board was good enough and cheap enough to build a community around it.

With Arduino and Raspberry Pi, the choice of prototyping platform was simple. If you wanted to talk to arbitrary bits of electronics, your best bet was to buy an Arduino microcontroller board; if you needed the power of an ARM-based board and wanted to run Linux, Raspberry Pi was the best option. If you needed both, you could run a USB cable between them.

The story could have ended there, but it doesn't.

The New Platforms

Prototyping platforms are becoming more expansive. It's cheap to add radios to boards, and some platform builders have taken this very far with "kitchen sink" boards. These platforms take the attitude that if one radio is good, another is better. If you can cram another sensor onto the board with only a minor increase in the bill of materials, then another one again seems like a good idea.

On the other hand, we're seeing the emergence of new "use everywhere" boards, which have minimal features but are cheap, have onboard networking of some sort or another—usually Bluetooth LE

or WiFi—and are usually optimized for low-power environments. Typically they also come in small form factors.

The "kitchen sink" board

The great thing about using Raspberry Pi or Arduino is that lots of developers have them, and there is a great deal of community support around them. But specialized boards, like MediaTek's LinkIt One, have so many features, they can be hard to resist. The LinkIt One for instance supports GSM/GPRS, WiFi, Bluetooth BR/EDR and LE and has an onboard GPS, as well as GPIO, I2C, SPI, UART and Grove connector support.

Boards like these are designed to be a platform for prototyping IoT devices, but the boards themselves are not designed to be deployed as IoT devices. Right now a lot of the work being done inside the IoT community is exactly this, to build platforms, developing something that other developers will build on.

The "use everywhere" board

The poster child for affordable, easy-to-deploy IoT boards is the ESP8266. The Espressif ESP8266 SoC was originally released as a serial-to-WiFi bridge for another microcontroller, like an Arduino.

But it cost less than $2, and as a result a somewhat bewildering selection of module and breakout boards have been produced. It was so cheap people started looking at it rather closely and discovered that it is a full microcontroller in its own right.

In fact, it's a very capable 80 MHz microcontroller based around a Tensilica Xtensa LX3 core with WiFi support, both as client and as an access point, supporting 802.11 b/g/n protocols at 2.4 GHz and WPA/WPA2 with a full onboard TCP/IP stack with DNS support. It has GPIO, I2C, I2S, SPI, and PWM support. It also has a 10-bit ADC.

Most of these "use everywhere" boards are designed with small form factors and low power requirements. In almost all cases, they also come with advanced power management capabilities and the ability of sleep (and wake) based on external interrupts.

These boards exist in a very different niche than the "kitchen sink" boards being developed as platforms for experimentation. On the one hand, "kitchen sink" boards provide infrastructure for already-

identified IoT niches, such as remote distributed sensor networks. On the other hand, they are generally too expensive in single-unit quantities to be viable deployment platforms. However, "kitchen sink" boards are often based on reference designs that chip vendors provide and can be invaluable prototyping tools.

You Won't Program to the Platform

The Arduino software has been expanded by its makers and their competitors to go beyond the initial 8-bit microcontroller it supported to ARM, x86, and other boards. Large chip makers, like Texas Instruments with its LaunchPad boards based around its MSP430 processor, have gone out of their way to provide Arduino-compatible development environments in an attempt to capture the Arduino developer community.

So even if you're not using an Arduino, you can prototype on another platform with the same programming language and libraries; the ESP8266 and LinkIt ONE are both examples of boards that you can program with the Arduino IDE.

The upshot is that connected device developers can reach for a Raspberry Pi, Arduino, or any of dozens of boards when prototyping, and won't have any shortage of developers or knowledge to draw upon. And when it comes time to bring a device to mass production, the path from one prototyping board to many manufactured units is well understood, even if you're a different chip than what powers the Arduino or Raspberry Pi.

But at the end of the day, your Arduino code is just C++ running on top of some libraries and macros that simplify your code and make it possible for the same source code to compile to multiple chipsets. With Raspberry Pi, you're using the same Linux software development tools that you use on desktop machines or servers.

If you stick with Linux on single-board computing platforms like the Pi, and the Arduino IDE on all the bare-metal boards, it will be hard to go wrong. You'll end up with a prototype that has portable code that can be deployed to many different platforms.

Talking to the Cloud

An IoT device does not necessarily need to connect directly to the Internet. However, while a smart, connected device can operate in a standalone fashion, or form part of a local (possibly mesh) network, most connect outward to the cloud in some fashion.

Notwithstanding our discussion in "Local, Edge, or Cloud?" on page 1, the architecture of most connected devices is similar: there is the thing itself, an application running on a computer or gateway device that controls the thing, and a cloud service that backs both.

To the Cloud

There are three main models of cloud computing: public, private, and hybrid. A public cloud is one in which services and infrastructure is provided by a specialist cloud company. A private cloud is one where services and infrastructure, the machines the services are running on, are owned and maintained by your own company. A hybrid cloud is a mix of public and private cloud infrastructure.

There's no reason the cloud components of a device need to be tied to the company that built it. There's nothing to stop companies from creating devices that will self-deploy the cloud capability the device needs directly to the customer's cloud provider of choice. This way, customers can pay for their own cloud usage but also migrate the device's cloud components between different providers as needed.

CHAPTER 4

The Physical Environment

At the prototyping stage, the physical environment your device operates in is fairly benign: an air-conditioned office or a lab. However, once out in the world, your final product may face an unforgiving environment: dust, vibration, exposure to elements, and even wear and tear from users.

Physical Environment

When designing the prototype, it's important to consider where it will eventually be installed. Will it be inside or outside? If inside, what sort of environment will it face? While an office environment may be temperature regulated, a factory floor may suffer extremes of heat or cold. If installed outside, there will be a large variation in temperature over the course of a year, or even over the course of each day. The device needs to be able to cope with the expected maximum and minimum temperature extremes and rapid variations between extremes.

Beyond the temperature, your device may have to face other factors. Perhaps your device's operating environment is subject to moisture and dust. If so, the device enclosure needs to be sealed against water ingress, with external physical ports covered for when they are not in use.

When your prototype has progressed beyond the breadboard stage, it's important to take it to the physical environment in which it will normally operate.

Consider the Enclosure

Thanks to advanced manufacturing technology, it's become easy, and scalable to a certain point, to manufacture one, one hundred, or even one thousand of something. The tools that now make prototyping easier—tools like 3D printers, CNC mills, and laser cutters—scale poorly beyond thousands of units. But most contract manufacturers regard orders of thousands, or even tens of thousands, of units as "low volume" or "short runs" and charge a premium.

For production runs of consumer products parts, 3D printing or CNC milling is usually too slow. But for industrial scenarios, where you might only build dozens of devices at a time, and where there is a lot of customization, these advanced manufacturing solutions might be just right.

For runs of hundreds or thousands, there are companies, like Protomold, that do provide service to companies needing small orders. They typically use molds made from aluminum, rather than steel, so your tooling cost is typically less expensive. However, these sorts of molds will have much shorter lifespans, which won't be a problem if you need to customize each run anyhow.

Good contract manufacturers are hard to find. If you are manufacturing in larger quantities, ask your potential manufacturers to do a detailed design for manufacturability (DFM) analysis of your product. They should be able to take your Gerber, CAD, and other design files and give you detailed feedback about how each part will be made, and any potential issues around building them, as well as ways to change the design to make it easier to manufacture. This can potentially save a large amount of money by cutting manufacturing times.

Heating and Cooling

Electronics generate heat. When overheating, some processors will throttle performance to stay within their ideal temperature range. Some processors will operate so far below their normal performance level that you'll notice the speed difference.

In some situations, electronics used in embedded devices can make do with passive cooling using small heat sinks. But in other situations, active cooling may be necessary. While fan cooling is traditionally thought of as noisy—laptops and microwave ovens

spring to mind—operating a modern, low-speed (under 2,000 RPM) DC-powered ball-bearing brushless fan at the low end of its voltage range will result in sound levels less that 16 to 18 dB. At these levels, the fan noise is inaudible from more than a meter away.

Even if operated at low speeds, small fans also may lead to dust accumulation inside the device, which can reduce cooling efficiency.

Alternatives to fans do exist. If you need to move large amounts of heat away from a component, you could use a Peltier cooler. Conversely, in those circumstances where the electronics need to be heated, rather than cooled, a Peltier heater can be used. The need to heat the electronics can occur if the device is meant to operate unprotected at high altitudes, or in colder climates. However, care must be taken when designing heating systems, as the electronics will self-heat in many circumstances. Left "on," many systems will generate enough heat from operation to stay in operation if protected from other factors by a good enclosure.

Enclosures

While human interface requirements will often drive the look and feel of an enclosure, the underlying requirements of the environment in which the device operates have to be taken into account.

A good place to start when thinking about designing enclosures, especially for industrial use, are companies like Polycase, which stocks a range of pre-built plastic enclosures and may be able to customize them with cutouts for little additional cost.

If the device is to operate for extended periods in an isolated location, you may wish to think about adding additional environmental sensors inside your enclosure to provide a self-monitoring capability to your device. You may even need to enable it to "call for help" if it determines that conditions are starting to get outside of its operating range before a failure occurs. A machine learning platform such as ThingWorx Analytics can apply machine learning to anomaly detection.

Power Requirements

The main consideration regarding power requirements is whether the device will operate on or off the power grid. If the device is designed to operate on electrical grid power, then considerations must be given to power efficiency (an inefficient power supply will waste money and generate excess heat).

If the device is to operate off grid, the required battery life will be the driving factor behind many design decisions. Large batteries will not only heavily influence the size and weight of the device, but they are also expensive and will have a dramatic influence on the bill-of-material cost of the device. Shipping costs may also be affected, as some shipping methods cannot be used with some commonly used batteries such as Lithium-ion. If you will be using solar panels with rechargeable batteries, you need to factor those into the design.

The primary power drain for most connected devices will be its radio. Therefore, anything you can do to reduce this can in turn lead directly back to a smaller battery, and lower bill-of-material costs. The choice of radio may well be determined by the power requirements. For instance, typically Bluetooth LE radios operate with power draws of less than 15 mA, while WiFi radios more typically require from 80 to 200 mA to operate. That can make a dramatic difference to the power consumption of a device.

Deep Sleep and Duty Cycle

In practice, many connected devices do not need to be connected to the Internet at all times. Therefore one common practice when working with "use everywhere" boards (discussed on page 15), which have often been designed to be used off-grid, is to use the deep sleep capabilities of the microprocessor.

For instance, the power consumption of the ESP8266 SoC drops from a typical 80 mA in operation, down to well under 10 mA while in sleep mode. This figure can be brought (much) lower by eliminating power-hungry additions like power LEDs attached to many of the breakout boards using the chip, down into the µA range. Using a 10 percent duty cycle and the sleep capabilities of the ESP8266 means that a 200 mA battery, which would normally be exhausted in just over 2 hours of normal operation, can be stretched to last over a

day. If deployed outside, even a small solar panel can be used to keep the battery topped off, allowing for long duration operation of the device.

Some chips, like the ESP8266, include wake-from-sleep on interrupt capability. With this feature, battery life can be extended even longer as remote devices can be woken up only if "something interesting" happens.

Connectivity

One issue that usually won't be encountered by your prototypes is a hostile radio environment. WiFi, Bluetooth, ZigBee radios, and several other standards all operate at or around the 2.4 GHz bands. This means that in deployment where there are lots of radios in operation, your device might suffer from poor throughput or other connectivity issues. To some extent this can be alleviated by intelligently choosing bands that are uncongested, but in the end there is only so much that can be done.

You therefore may have to consider communications protocols that are robust against data loss and disruption. The Space Communications and Networking team at NASA has done a lot of informative work on disruption tolerant networking (DTN) standards. In addition to the basic store-and-forward internetworking service, DTN also provides efficient reliability, security, in-order delivery, duplicate suppression, class of service (prioritization), remote management, rate buffering, and data accounting, all over possibly asymmetric and time-disjoint paths. Applications including file transfer, messaging, and streaming audio and video can all be implemented on top of DTN.

Another issue which may affect connectivity of your device includes accounting for motion. Is the device statically installed or in a movable enclosure? Or is it installed in a location that will *itself* move (e.g., under the hood of a truck)? If the device is intended to move, then DTN may prove to be a viable option, or alternatively more than one radio may be required to give multiple routes out to the Internet. For example, primary WiFi networking may be shut down when the connected device detects that it has left the area of WiFi coverage and a backup WAN connection enabled (see "Wide Area Networks" on page 5 for options). If available, a low-powered secondary WAN radio is preferable. (The Netherlands has just become the

first country to roll out a low data rate [LoRa] mobile communications network throughout the country.)

Storage

Beyond connectivity there is also the issue of local storage. In a unique (for modern times) situation, hardware resources are scarce on embedded platforms. How much application memory (RAM) and persistent storage (e.g., flash memory) is needed by your use case? If you make use of DTN, will longer term storage increase due to caching? It is difficult to discuss storage in general terms, as the need for onboard storage will vary dramatically with usage, but it should be factored into your design from the start, using estimates of your projected data throughput.

Where Does the Data Go?

When thinking about storage you should also consider how and where the data is going to be processed. While it may seem initially convenient to send all the data off to a remote cloud for analysis and storage, as the number of devices scales, that can become on onerous burden. An additional consideration is that you never have better context for data than at the moment of collection. Reconstructing that context later (or elsewhere) tends to be expensive, if not outright impossible. Instead of streaming every bit of information up to the cloud, make as many decisions as you can locally (on the device), process the data as much as possible without losing important details, and send only the data you need up to the cloud. This will not only save energy, but will save any costs associated with data transfer (which is particularly important when dealing with a cellular connection).

CHAPTER 5

Security Is Your Job

Security has to be one of the first things you consider when you design a connected device. Customers are far more sensitive about data generated from things they can touch and handle than they ever have been about data created on the traditional Web. Big data is all very well when it is harvested quietly, silently, and stealthily, behind the scenes on the Web. Because, to a lot of people, the digital Internet still isn't as real as the outside world. But given the IoT's connection between the digital and physical, the stakes are high.

Ignoring security for a connected device, or even leaving it until later in the development process, is a mistake. It needs to be engineered into your device from the start. These seemingly smart devices are attractive to hackers because for a lot of manufacturers security is still viewed as an afterthought.

A Unique Security Problem

Even for devices with good security, the IoT presents a unique security problem. In the past, a great deal of computer security has relied on attackers not having physical access to the computer, but with an IoT that's the point—with small devices spread all over the office, factory, and more, it opens up a whole new can of security worms. This physical vulnerability of IoT devices means that attackers can leverage their access to a smart device to gain further access to a corporate network, and potentially compromise much more than just a single device.

Authentication and Authorization

When you log on to a device with a user name and password, you are authenticating. However, this is different than authorization. Authorization is the process of verifying that you should have access to something. One of the ongoing problems in computer security is that often these two very different concepts are pushed together into a single scheme. This is exacerbated in the case of smart devices, as many of the schemes we're accustomed to—the ubiquitous username and password of the digital Internet—no longer work for devices without a screen. The visual feedback to users of the lock icon in their web browser's location bar, reassuring them of a secure connection between them and the cloud, is also absent.

However, the features that make smart devices powerful also make them a new vector for verifying our identity and authenticating us, both to itself and the network of devices around it.

We need to consider how systems of devices, rather than a single device, should be authenticated.

The Internet of Things and the Industrial Internet

While the Industrial Internet has its roots in the SCADA systems of the early 1960s, the IoT has its roots in the web architectures of the dot-com boom. The clash of those cultures and architectures may well contribute to dangerous security problems.

The Industrial Internet isn't necessarily about connecting big machines to the public Internet; rather, it refers to machines becoming nodes on pervasive networks that use open protocols—therefore, Internet-like behavior follows. These behaviors occur because a lot of things become possible if the network can just be assumed… if connectivity can be assumed.

Earlier systems that tie together decentralized facilities were designed to be robust, easily operated, and repaired, but not necessarily secure. To be fair, such systems were never intended to be connected to public network. Unfortunately, this hasn't stopped people taking legacy systems and connecting them to Internet, often by employing a serial-to-WiFi bridge that plugs right into a legacy RS-232 port, exposing devices that were never meant to be exposed.

There's a big temptation to do so: it makes things a lot easier, and it looks powerful. Unfortunately, by virtualizing access to serial ports, and exposing them as IoT edge devices, many large systems that drive large-scale machinery are directly exposed to attack.

Stuxnet

Stuxnet was the first of a new breed of malicious code. It attacked in three phases. First, it targeted Microsoft Windows machines and networks, replicating itself. Then it sought out Siemens Step 7 software, which is a bit of Windows-based software used for industrial control systems, before finally compromising the programmable logic controllers (PLCs) attached to those boxes. But crucially, this would only happen if they were operating a very small range of variable-frequency drives: centrifuges, in other words. The worm's authors could thus spy on the industrial systems and then cause these fast-spinning centrifuges to tear themselves apart.

Now most speculation identifies Stuxnet's target as Iranian nuclear plants carrying out uranium enrichment: as many as 60 percent of the identified infected machines were in Iran, and the complexity of the worm implies that a nation-state was behind it.

However, Stuxnet was not a one-off or an aberration. It was a high-profile flag for what's coming as more and more sensors and actuators are put on public-facing networks. Most of these are going to be much softer targets than going after a IR-1 centrifuge operating with uranium hexafluoride. The next big attack will almost inevitably trade sophistication for scale.

Broken Firmware

One potentially serious problem with many of today's smart devices is that the high-level "smarts" often sit on top of the same silicon as other devices.

Because consumer device exploits are generally publicized more than privately deployed enterprise exploits, we'll look at examples from the consumer space.

Both the Fitbit Aria WiFi bathroom scales and the Ring smart doorbell make use of a WiFi module produced by GainSpan.

Pen Test Partners discovered a vulnerability in the firmware of the GainSpan module used by the Aria scales. It allowed attackers to retrieve the SSID and WPA PSK of the owner's network by placing the scales into setup mode, which can be simply done by pressing the reset button on the bottom of the scales, connecting to the access point (AP) the scales create when in this mode, and retrieving the information from a standard GainSpan firmware–provided endpoint.

The same firm discovered a similar vulnerability in the Ring Doorbell. Because the Ring Doorbell is mounted outside a facility, rather than in a bathroom, the vulnerability in this case was much more serious. Ring patched the vulnerability immediately when notified, but the device still exposes a number of pages left from the GainSpan SDK.

These cases show the potential security problems when dealing with off-the-shelf modular hardware. Very quickly, a vendor's error can become your worst nightmare. Most connected devices will be built from standard modules, as developing proprietary silicon is far beyond the capabilities of almost any company considering building a device. However, these modules do come with their own vulnerabilities. In the case of GainSpan WiFi, the original manufacturer regarded this as normal operation of its SDK and advises all manufacturers to remove these endpoints before production.

Fixing these sorts of problems once a significant number of units are in the wild can be problematic: most users can't or won't perform firmware updates—few will be aware when such updates may be necessary. You will find bugs in your connected device after shipping begins. Sometimes these bugs can lead to large exposure surfaces for attacks and will need to be fixed. While there are companies like Resin.io that are working to simplify automated firmware update deployment to distributed devices, right now the burden on doing so lies squarely with the manufacturer of the device.

Fixing the Firmware?

The time to fix firmware in a product that is massively distributed can be protracted, even if the manufacturer is proactive in fixing the problems. Depending how the device interacts with the cloud, or how thoroughly the security scheme is integrated into the SDK, making the required changes can take a great deal of time.

At the start of 2014, using a combined approach investigation of the Estimote Bluetooth LE beacon SDK proved that the beacons were easily reconfigurable in the field by unauthorized third-party attackers. The implications of that were fairly far reaching. If someone maliciously changes the iBeacon Major or Minor characteristic of a beacon, any consumer application configured to use that particular beacon will stop working. The beacons must be configured with a pre-defined identity to trigger the correct behavior inside the customer's own application when a device comes into proximity of the beacon.

Beyond that, you could potentially configure a "fake" beacon to act as an impostor of another beacon belonging to a retail chain, potentially gaining access to promotions, gift cards, and other location-dependent goodies tied to the beacon you're impersonating.[1]

A year and a half later, in the wake of Google's announcement of its new beacon standard Eddystone, the Estimote beacons were updated. However, despite changes to its SDK, the vulnerabilities discovered were still present in the beacon SDK. The added capabilities of the Eddystone support made the presence of this vulnerability much more critical. With a URL it is much easier to trick users into visiting a malicious web page, which could then automatically download and install a root kit onto their device.

Once publicized it took Estimote a month to fix the vulnerability in its firmware.[2] Do you think you could develop, test, and roll out a fix in one of your devices that quickly?

Reverse Engineering the Hardware

Dropping below the exposed software, and even below the firmware level, physical access to the hardware means that it, too, is vulnerable. Many connected devices are put in production with a serial port still on board. The pads on the PCB may no longer be connected to a socket that is exposed on the outside of the case, but the traces still exist on the board itself.

1 In fact, working with Sandeep Mistry, Alasdair did this twice with the CES Scavenger Hunt. See "Hacking the CES Scavenger Hunt" and "Hacking the CES Scavenger Hunt for a Second Time" for details.

2 See "Once an Altoids Tin, Now a Pinhole Camera…".

While not trivial, it's perfectly possible to use these vestigial serial ports—left by the engineers that designed the board for debug and (possibly) technical support purposes—to reverse engineer the device. Bypassing any high-level security, these ports often give attackers direct access into the heart of the device, its firmware, and even the data flows (SPI traffic, for instance) between pieces of hardware.

Beware What You Put in Production

You should be very mindful of the hardware that you put into production. While it's tempting to leave debugging ports on your board when it goes into production, and there may be good reasons why it should go into production that way, you must be careful about how and what it can access.

As you progress in the prototyping process, you need to carefully distinguish throwaway prototypes from the intermediate prototypes that bring you closer to your final product, making adjustments as necessary. Prototyping using off-the-shelf hardware, like the Raspberry Pi, can often lead to small-scale production runs using the same hardware as your prototype. Unfortunately, people rarely remember to update the software on these off-the-shelf devices, and the device accumulates well-known vulnerabilities over time.

If you deploy a device using the Raspberry Pi or a platform similar to the Raspberry Pi that runs a full Linux distribution, you need a plan for pushing updates to the device, and you need a way to distribute emergency updates with great haste. A botnet attack can make short work of exploiting thousands of devices shortly after a new vulnerability is disclosed, as the Carna botnet proved.

Built by an anonymous researcher to measure the extent of the Internet, the Carna botnet was designed to attack small embedded systems—the precursors to today's IoT devices—rather than desktop computers. The botnet made use of almost trivially exploitable security vulnerabilities, such as routers using the default password, to build a large-scale distributed port scanner. While a solid security strategy is necessary when building a connected device, the success of the Carna botnet is telling: simple *default* passwords gave its author access to hundreds of thousands of consumer devices, as well as tens of thousands of industrial devices.

The author documented the botnet in an online paper. As the author of the botnet concluded, "A lot of devices and services we have seen during our research should never be connected to the public Internet at all. As a rule of thumb, if you believe that 'nobody would connect that to the Internet, really nobody,' there are at least 1,000 people who did."

CHAPTER 6

Time to Market Versus Common Sense

As technology matures, it becomes cheaper. The ubiquity of the ARM processor, used in pretty much every smartphone, has dramatically dropped the price of computing. This rapid drop in the price of computing platforms at the low end has made prototyping much easier and has enabled a generation of new prototypes to be built.

Right now the proliferation of the "kitchen sink" developer boards (discussed on page 15) means it is easier than ever to prototype a product—however prototypes are not products. They're relatively expensive, often have large form factors, and can't be integrated into products. They're intended to aid development, not the core of your product, no matter what some manufacturers claim.

NOTE

There are some exceptions, such as companies that offer wireless modules that you can integrate into your own products. These include the Particle P0 and P1, which provide the core functionality that drives their larger development boards (in Particle's case, their Photon board). These modules offer a way to take your prototype and create a custom PCB and reuse the code from your prototype without changes.[1]

1 Particle offer a series of purchase options depending on the scale you intend to use their product; see "Particle Wholesale for Businesses".

The "use everywhere" boards (discussed on page 15) are cheap, low powered, and can indeed be used, if not everywhere, then many places. Even if you don't use the boards directly in your product, you can easily adapt them to your design. For example, the design of an ESP8266 breakout board that might run you $2 can be easily integrated into your own device design. At that point, you're just paying for the ESP8266 modules, which are much cheaper than the breakout boards.

It's at this stage, when it's time to go to market, that you can fall into the temptation of using your prototype as your product blueprint. But that will have consequences.

Off-the-Shelf Components

If you're manufacturing in low volume, you may well want to choose to base your connected device around an off-the-shelf board. While typically more expensive per unit than building your own custom PCB, it's possible you can cut a large amount of up-front development time (and cost) by following this route. Taking an existing board and customizing it, or using a board designed to scale directly into production, such as the Particle Photon, means that your development time is spent adding the features that make your connected device unique and valuable rather than reimplementing an underlying platform. In this case, your final product may look a lot like your prototype.

What About the Prototype?

If your prototype is based around one of the existing single-board computers (such as the Raspberry Pi) or a network-enabled microcontroller (such as the ESP8266 or the Particle Photon), it's tempting to continue with that into the production stage. As you move from your initial prototypes, it's important to take a step back and consider what it is you're trying to build. This is especially true if your prototypes were built around a "kitchen sink" board that will inevitably be large and expensive. While some of these boards can be customized—and ordered from their manufacturers in custom (stripped) configurations, by removing parts to lower the bill-of-materials costs—most cannot.

Managing Risk

There are two main types of risk when manufacturing a new product: technical and product risk. All hardware products share some element of technical risk—engineering constraints (or the laws of physics) might prevent you from being able to deliver the product. Most startups are aware of this and manage the risk fairly well. But fewer manage product risk. This is the risk that the product, once delivered, will fail to live up to expectations. It will work, but it may be unreliable; the look and feel of the product may be poor; or in some other manner the user experience may be below expectations.

The amount of product risk that your device is subject to is normally heavily dependent on how critical the device operation is to the end customer. For instance, an automated irrigation system that only polls the weather hourly may be less critical than a door lock that only works correctly once per hour. If the plants have to wait an hour for water, it's not an inconvenience. But if employees can't get into the building, you'll hear about it right away!

Failing Gracefully

Leslie Lamport, an early pioneer in distributed systems, said that "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable." By their very nature connected devices are distributed systems. There is the smart thing itself, the computer (or smartphone) that the user typically uses to interact with the device, and in many cases a cloud system behind both of these.

To manage product risk successfully, especially in high-risk systems where a small number of failures (say, one or two) over the lifetime of the system can have a severe impact on safety or revenue, it's important to fail gracefully. A user should not be locked out of the warehouse if there's a power outage.

Unlike systems that live purely in the digital world, connected devices live in the physical world, which means they are inherently unreliable. That unreliability must be a factor in the design of any connected system.

CHAPTER 7
Conclusions

The connectivity for Internet-enabled devices is well established, and the technology continues to mature at a rapid pace. Even so, we are still in the very early days of working out guidelines for how we should build connected devices.

Unlike software, there are few established patterns to follow in hardware, and most manufacturers are still feeling their way toward not just how to build a connected device, but how to ship it and support it afterward. The business models that supported the digital Internet do not work well with the IoT.[1]

Going forward, you will need to make sure you completely address the key life-cycle activities of a connected device. First, in the prototyping of your device, you need to understand what the device will do, establish a framework for iteration from prototype to final device, and adopt the network architecture that optimizes for cost, availability, and bandwidth needs.

Next, you will need to recognize your device is a platform for your customers' developers. At a minimum, someone will have to integrate it. More likely, they will have to engage in significant development and customization. Recognize this fact and embrace it, and create affordances and entry points that let your customers build something great.

1 See Alasdair's blog post "The Business of Things" for a discussion of current IoT business models.

You'll also need to understand the constraints of the physical environment the device will operate in. Your enclosure is your first line of defense against a harsh environment, but you also have to think about how you will get power to your device, and how it will physically handle connectivity.

Amid all the other considerations, security is the biggest challenge, if only because it is ongoing. After a device is deployed, keeping it secure and protected against vulnerabilities becomes your duty, to both your customers and their customers.

Finally, you need to balance time to market and the various product risks at key stages of the manufacturing process. The choice of whether to use the same modules in prototyping and production has a big impact on time to market. The decisions you make along the way to market will impact the risks you are subject to when your device is in the field.

Building smart, connected devices gives you opportunities in product development and product management that have never existed before. You can create products that deliver tremendous value to your customer. And after the product is deployed, you and your customers end up being partners in your customers' success. If you build your smart, connected devices in a way that addresses customer needs, contains the building blocks that developers need, offers ongoing security, and is physically constructed for the job, you will create a virtuous cycle powered by the connections between devices, people, and the cloud.

About the Authors

Alasdair Allan is a scientist, author, hacker, tinkerer, and co-founder of a startup working on fixing the Internet of Things. He is the author of a number of books, and from time to time, he also stands in front of cameras. He is a contributing editor for MAKE magazine and a contributor to the O'Reilly Radar. A few years ago, he caused a privacy scandal by uncovering that your iPhone was recording your location all the time. This caused several class action lawsuits and a US Senate hearing. Several years on, he still isn't sure what to think about that. Alasdair is a former academic. As part of his work, he built a distributed peer-to-peer network of telescopes, which, acting autonomously, reactively scheduled observations of time-critical events. Notable successes include contributing to the detection of what—at the time—was the most distant object yet discovered.

Brian Jepson is an O'Reilly editor, hacker, and co-organizer of Providence Geeks and the Rhode Island Mini Maker Faire. He's also active with AS220, a nonprofit arts center in Providence, Rhode Island. AS220 gives Rhode Island artists uncensored and unjuried forums for their work and also provides galleries, performance space, fabrication facilities, and live/work space.