G. Blake **MEIKE**
Larry **SCHIEFER**

Inside the Android™ OS

1

Why Android?

Android was built for small. It has scarcity designed deep into its DNA.

It was created in the early 2000s, a time when mobile devices were divided into categories like "smart phone" and "feature phone"; when flash drive program/erase cycles were counted in tens of thousands; and when 64 megabytes was a lot of RAM. The idea at Android's very core— that because there is no backing store to which to swap running programs, when memory gets tight the operating system has no choice but to terminate them—is the inescapable legacy of its fixation on frugality.

Modern smart phones have all the capabilities that laptops had at that time. Were it designed today, Android would likely be a very different thing. Although battery life is still a subject of much concern, a modern mobile OS could swap to flash memory as effectively as a modern laptop swaps to its SSD. Modern Android developers are supplementing—even replacing—the simple, frugal libraries built into Android with new and powerful but much more resource- intensive libraries such as GSON and RxAndroid.

At the same time that mobile phones, the original target of the Android OS, are outgrowing its architecture, a new and possibly larger opportunity is appearing: the Internet of Things and the smart devices that comprise it. In much the way that even small companies found, around the turn of the century, that they needed a web presence to compete, so many of the same compa- nies are now discovering that their *products* need a web presence to compete. From medical devices and on-board systems in cars, to homes, appliances, and even clothes, all sorts of prod- ucts are being supplemented with built-in intelligence. Many of these devices have substantial constraints on the processor they can support. Price, design, and flexibility make Android an excellent choice for powering these types of devices.

Adopting Android

There are a lot of reason that Android might be a good choice for a new smart device.

Full Stack

The Android OS addresses the full stack of product requirements. From hardware and the kernel to stereo audio and displays on multiple screens, Android offers flexibility and provides a wealth of options. One can think of Android as similar to a distribution of GNU/Linux such as Mint or CentOS. It transforms a device from a warm piece of silicon to a useful computer with basic functionality.

Broad Acceptance

Perhaps the most obvious reason for choosing Android for a hardware project is that it is ubiq- uitous. Some versions of Android run out of the box on nearly any common chipset. In fact, most SoC (system on a chip) vendors provide reference hardware kits with a version of Android and a backing Linux kernel. At least as important is that many developers are familiar with the Android system. Building a team, from front-end application and UI experts to those with the deep understanding of Android necessary to modify its core, should not be an impediment.

Beautiful UI

The Android system is capable of producing stunning user interfaces, which is perhaps its most important feature. Support for most popular audio and video media is baked right in and is relatively easy to use. Offering full lighting and shadowing, the tools for animation and 3D display are top-notch. One has only to look at some of the simply gorgeous applications such as Feedly or Weather View to grasp the nearly unlimited potential of the Android design palette. When the existing Android UI framework is not enough, the system supports both Open GL ES and Vulkan for low overhead, high-performance 3D graphics.

Linux Based

The Android operating system is based on the Linux operating system. Linux is one of the most popular and widely used of all operating systems. It is everywhere. Whether a chip is ARM-based, Intel-based, or something radical, nearly every chipset manufacturer provides a version of Linux that runs on their device. Bear in mind, however, that Android is officially only available for ARM- and Intel-based processors (both 32 and 64 bit.) This means the Android Open Source Project (AOSP) tree's build system, pre-built tools, test suites, and the publicly available native development kit (NDK) only support these architectures. That is not to say Android cannot run on some other architecture, just that the toolchains and build systems for those architectures are not supported.

While simply getting Linux ported and running on a new board is an important first step, quite a bit of work may well be necessary to get all the hardware accessible to software. Frequently, because the Linux kernel is licensed with the GNU Public License (**GPL**), the custom code necessary to support a particular sensor, display, or port will already be available for free online. Even if this is not the case, a large community of developers exists that is very familiar with the

process of building new drivers for Linux. Accommodating a new board or a new device is, if not always simple, at least fairly straightforward.

Powerful Development Environment

The Android toolchains are quite powerful and are undergoing constant improvement. Both toolchains—that for building Android's infrastructure and that for building Java applications that run on top of Android—are based, largely, on common off-the-shelf tools.

The Android source code, the Android Open Source Project (**AOSP**), is well supported. Creating a build of the version of Android used in this book, API 29, is relatively straightforward on recent OSX and Linux platforms. The build system with its directory-based mechanism for per-hardware customizations was originally based on GNU `make`. With the Nougat release, the `soong` build system replaced `make`. `Soong` uses two additional tools, `kati` and `ninja`, to make the build much faster than it was using `make`. Existing makefiles continue to work as-is along- side the new `soong` build files.

Most of the development for an Android system—even most of the system-level programming —is done in Java. Android Java developers will use tools like Android Studio (a fork of IntelliJ's IDEA IDE) and Gradle, the standard build tool for Android applications. Gradle is very definitely sufficient for building, testing, and packaging even system applica- tions with native components.

As mentioned previously, as of about 2014, several new, interesting, and powerful development frameworks are available for use in Java Android application. Although some are large and all are a trade-off in battery life, tools such as Realm DB, RxAndroid, and Retrofit can drastically improve the effectiveness of a development team.

One of the most important advancements in Android development was the announcement of Kotlin as a first-class supported language in May of 2017. Kotlin is clear, succinct, power- ful language that seamlessly integrates with existing Java and even native code. Although it is not widely used within AOSP's core (yet), it can be used by applications written for a custom Android platform.

Open Source

Building an embedded device necessarily involves negotiating a minefield of legal issues. This book is technical, not legal. We are not lawyers and nothing in this or any of the subsequent chapters should be interpreted as legal advice. If you intend to attempt marketing your own device, you will need the help of a qualified lawyer. That said, some broad generalizations might help a budding device developer to understand the moving parts. Figure 1.1 is a very high-level model of code ownership in the Android system.

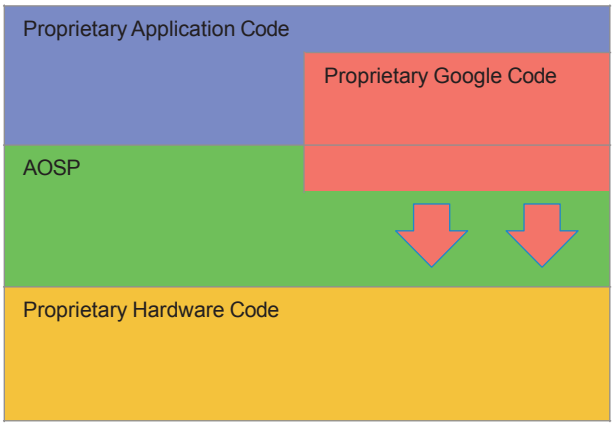

Figure 1.1 Android Code Ownership

At the bottom of the stack is proprietary hardware code. It is usually obtained from the hard- ware vendor, who may license it at no cost or, perhaps, impose some kind of fee. This code is frequently not open source. It is essential, but it may be delivered as pre-built binaries and possibly with strict legal injunctions about reverse engineering. Quite possibly you will never see the source and even the documentation for it might not be very good. Android's primary interface to this code is the Hardware Abstraction Layer (HAL), discussed in Chapters 8 and 10.

At the top of the stack are applications. These are things like the controller for a proprietary home-entertainment system or the Facebook and Twitter apps. Apps are also likely to be propri- etary and, again, unless they are apps you develop yourself, you may never see their source code. If your platform needs specific applications, you will either have to make agreements with their owners or provide some kind of service (a store or marketplace) from which an end user can acquire them.

In between these two proprietary layers is the code base for Android itself, the Android Open Source Project (AOSP). It is completely open source. You can read it, copy it, customize it, and use it pretty much as you like. Nearly all of it is protected with licenses that even allow you to redistribute only pre-built binaries, should you choose to do so. You can take as little or as much as you need.

People are frequently confused about the openness of the AOSP code base, because Google strictly controls contributions. It is true that you are unlikely to be able to contribute a change to the canonical AOSP codebase as you could with most other open-source projects. What you can do, though, is create a fork of the relevant code repository, change it in any way you choose and use it wherever you want.

Although the AOSP code is truly open source, it is not necessarily free of legal encumbrances. Many of the technologies built into Linux and the Android services based on top of it have been the subjects of large and small legal battles. Among the technologies that you might need

to license are things such as Wi-Fi, Bluetooth, multimedia codecs, and other more esoteric things (like a file system!).

What makes this even more confusing is the AOSP tree includes software implementations of some of these components (such as multimedia encoders and decoders) as placeholders. They do not come with any kind of patent or license grant from Google or the intellectual property (IP) holders. When you build a new device, these third-party components must be carefully examined to ensure the device is in compliance with technology-specific licenses.

Microsoft, in particular, has a portfolio of patents that it has successfully used in strong- arming an estimated $1 billion in license fees from various Android device manufacturers. The exact contents of this portfolio were a closely held secret for many years. In 2014, however, the Chinese government leaked the contents of the portfolio, and you can now easily find it online.

AOSP and Google

Google holds its control over the Android OS in two ways. First, most consumer Android-labeled systems contain a feature-rich, proprietary Google platform that is not part of AOSP. This platform includes things such as Google Play Services, Google Maps, and the Play Store.

Second, to install any of these proprietary services—or even, for that matter, to label the system as an Android™ system and adorn it with the Android robot icon—a device manufacturer must ensure the device complies with the Android Compatibility Definition Document (CDD) and passes the Android Compatibility Test Suite (CTS) and the Vendor Test Suite (VTS). After a device is verified as compliant, Google Mobile Services may be licensed for the device, allowing it to ship with Google's proprietary add-ons.

These constraints—the Google proprietary code, the licensing agreement, compliance with the CDD, and passing the CTS and VTS—do not affect the use of the AOSP codebase. Developers and device creators are free to use and adapt the AOSP code as long as they neither label the resulting device as "Android" nor need the Google proprietary code and its associated function- ality (marketplace, cloud services, and so on).

Several examples exist of forks of the AOSP codebase. Perhaps the best known of these is Fire OS, used on Amazon devices such as the Kindle, Fire TV, and the Fire Phone. Many, if not most, applications built for Android will run on Fire OS. Nonetheless, Fire OS cannot be labeled as Android, cannot include the Google Play Store, and does not support Google Play Services.

Both Samsung and LineageOS (formerly CyanogenMod) also maintain operating systems that are heavily modified versions of the AOSP codebase. Both of these forks, though, have managed to pass the CTS and stay on Google's good side. Both are labeled as Android.

Many other examples of AOSP code in non-Google devices exist, from popular phones and tablets in China and India to the UI for Comcast's Xfinity service. Although each of these devices has its own legal story and its own legal concerns, the use of the AOSP codebase is not, in itself, a problem for any of them. Depending on whether a manufacturer feels that it can

provide an alternative to the Android label and the accompanying Google proprietary services, it either does or does not invite Google into its AOSP-based product. The devices it creates using AOSP code do not need be visible to Google in any way and do not need any permission or participation from Google.

Where it gets complicated is devices in the middle ground between the two ends of the spec- trum just described: a device that does not need or want the Android label but on which the manufacturer wants to include apps with which users are familiar but are provided by Google. For example, consider a kiosk for renting movies that is powered by Android. The manufacturer would like to bundle YouTube so that users can view video trailers. Google's position is this arrangement is not supported. The manufacturer needs either to ensure the device is CDD/CTS compliant or find an alternative way of providing the desired functionality.

Other Choices

The number of products with embedded systems may be exploding but the idea of embedded computing itself is nothing new. Many alternatives to Android are available as the intelligence for an IoT device; Real Time Operating Systems (**RTOSs**), some much older than Android and some newly developed.

An even better alternative, though, might be no OS at all.

Micro-Controllers

Even with the falling prices and increasing power of single-board computers (**SBC**s), at the time of this writing, a board that can run Android will cost something in the $20–$50 range. It will also occupy around 20 cubic centimeters of space. That can be a lot of overhead for a small device.

When cost and space are of paramount importance, a micro-controller like the wildly popular Arduino may be an attractive alternative. Most micro-controllers are not full-fledged processors and cannot support multiple simultaneous processes, Linux, or a flashy UI, let alone Android.

At the time of this writing, so-called "mini" micro-controllers are a full order of magnitude less expensive than SBCs and may require less than a single cubic centimeter of space. Over time, certainly, the line between SBCs and micro-controllers will blur. SBCs will get smaller, micro-controllers will become more powerful, and RTOS capabilities will scale linearly with the hardware. Even now, though, it is possible to accomplish some very impressive magic with one or more small, simple micro-controllers.

An important limitation of micro-controllers, to be considered before choosing one as the brains for a project, is its upgradeability. While it is certainly possible to update a microcontroller-based system over the air, it can be difficult and might require additional specialized hardware. If over-the-air (**OTA**) updates are part of your device strategy, you might need a full-fledged OS.

Simplicity is a double-edged sword. A system that can be updated can be hacked. Hacking a micro-controller is entirely possible. Think Stuxnet. However, doing so is probably difficult and not interesting to an attacker. A simple read-only memory (**ROM**)-based micro-controller that is just sufficient to power your project may save you from a substantial security budget and keep your product out of the headlines during the next distributed denial of service (**DDoS**) incident.

Other RTOSs

The list of operating system alternatives to Android is long. Each of them solves some set of problems and introduces others.

QNX

QNX was the most popular embedded OS in the world before the advent of Android. Originally called QUNIX, it is a micro-kernel-based system and was developed by two students at the University of Waterloo. It was released in 1984 as QNX to avoid trademark infringement. Since that time it has been rewritten several times, and sold, first to Harman International and then to Research In Motion, now Blackberry. Shortly after Blackberry acquired QNX, it restricted access to the source.

VxWorks

If your device needs the kind of reliability and dependability that powers the Martian probes and military aircraft, you should consider VxWorks. VxWorks is a proprietary OS originally developed by RTOS pioneers, Wind River. Wind River is now a wholly owned subsidiary of Intel.

The VxWorks kernel is monolithic (unlike QNX), but the system is nicely modularized and the toolchain well developed. All of this, of course, comes at a price: Vxworks is proprietary and closed source. Wind River also produces Wind River Linux, a hardened kernel with a custom build system.

Android Things

Android Things is Google's version of Android stripped down for IoT. Codenamed Brillo, this stripped down version of Android was designed to be used by low-power IoT devices with significantly less RAM (as low as 32 MB) while still including Bluetooth Low Energy (BLE) and Wi-Fi support. Android Things requires manufacturers to use supported single-board computers (SBCs) or System on Modules (SoMs). Such devices would automatically receive OS and security updates from Google. Additionally, Android Things included a standard framework for develop- ing custom hardware interfaces without requiring changes to the underlying kernel or Android framework. This approach allowed IoT manufacturers to focus on their specific purpose and not worry about the underlying OS, its security, or system-level updates.

Unfortunately, in early 2019 Google announced that Android Things had been refocused on smart TV and smart speaker systems and that broader long-term support is at end of life.

Others

Windows CE has been a very popular embedded OS as is evidenced by the blue screens of death on everything from subway and traffic signs to vending machines and museum kiosks. Microsoft recently introduced a successor to CE, called Windows 10 for IoT.

Nucleus RTOS, from Wind River's long-time rival, Mentor Graphics, found a home in a number of Samsung, LG, and Motorola phones. Riot OS, Arm Mbed OS, and Green Hills Integrity are all also players.

Nearly all of these popular RTOSs are proprietary and closed source. Free and open-sourced RTOSs are out there, though—FreeRTOS, MontaVista, and Contiki, to name a few—but none of them has the kind of history and support that Android has.

Summary

You can find many alternatives to Android. Few, however have the collection of features and support that make it such a great choice for an IoT project:

- It is free. Take as much or as little as you like; use it as you please.
- It is portable. Android can be made to run on virtually any type of hardware. Getting any operating system running on a new device can be very difficult. Android is no exception. There is, though, a lot of existing code and a large community with lots of experience with porting it.
- It is adaptable. Plumbing support for new peripherals into the Android framework is a straightforward and usually simple task. Doing so is the subject of the rest of this book.
- The toolchain is good. The low-level C and C++ code use standard tools augmented with a baroque but useful build system. Most of the code—high level and written in Java—is supported by Gradle, a couple custom plug-ins, and Android Studio. All of these tools are undergoing constant improvement.
- It supports reactive and beautiful UIs. Android can handle a variety of media, both audio and video. It has powerful tools for animation, and supports three-dimensional layouts and both touchscreen and D-pad input.