Early Release

Embedded Android

Karim Yaghmour

O'REILLY®

Embedded Android

Karim Yaghmour

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Embedded Android
by Karim Yaghmour

Revision History for the :
See *http://oreilly.com/catalog/errata.csp?isbn=9781449308292* for release details.

ISBN: 978-1-449-30829-2
1317666851

Table of Contents

Preface

Android's growth is phenomenal. In a very short time-span, it has succeeded in becoming one of the top mobile platforms in the market. Clearly, the unique combination of open source licensing, aggressive go-to-market, and trendy interface is bearing fruit for Google's Android team. Needless to say, the massive user uptake generated by Android has not gone unnoticed for handset manufacturers, mobile network operators, silicon manufacturers, and app developers. Products, apps and devices "for," "compatible with," or "based on" Android seem to be coming out ever so fast.

Beyond its mobile success, however, Android is also attracting the attention of yet another, unintended crowd: embedded systems developers. While a large number of embedded devices have little to no human interface, a substantial number of devices which would traditionally be considered "embedded" do have user interfaces. For a goodly number of modern machines, in addition to pure technical functionality, developers creating user-facing devices must also contend with human-computer interaction (HCI) factors. Therefore, designers must either present users with an experience they are already familiar with or risk alienating users by requiring them to learn a lesser known or entirely new user experience. Before Android, the user interface choices available to the developers of such devices were fairly limited and limiting.

Clearly, embedded developers prefer offering users an interface they are already familiar with. Although that interface might have been window-based in the past—and hence a lot of embedded devices were based on classic window-centric, desktop-like or desktop-based interfaces—Apple's iOS and Google's Android have forever democratized the use of touch-based iPhone-like graphical interfaces. This shift in user paradgims and expectations, combined with Android's open source licensing, have created a ground-swell of interest for Android within the embedded world.

Unlike Android app developers, however, developers wanting to do any sort of platform work in Android, including porting or adapting Android to an embedded device, rapidly run into quite a significant problem: the almost total lack of documentation on how to do that. So, while Google provides app developers with a considerable amount of online documentation and while there are a number of books on the topic, such as O'Reilly's *Learning Android*, embedded developers have to contend with the minimalistic set of documents provided by Google at *http://source.android.com*. Embedded de-

velopers seriously entertaining the use of Android in their system were essentially reduced to starting with Android's source code.

The purpose of this book is to remedy to that situation and enable you to embed Android in any device. You will therefore learn about Android's architecture, how to navigate its source code, how to modify its various components, and how to create your own version for your particular device. In addition, you will learn how Android integrates into the Linux kernel and how to leverage Android's Linux heritage. For instance, we will discuss how to take "classic" Linux components such as glibc and BusyBox and package them as part of Android. Along the way, you will learn day-to-day tips and tricks, such as how to use Android's *repo* tool and integrate with or modify Android's build system.

Conventions Used in This Book

The following typographical conventions are used in this book:

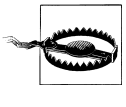
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Embedded Android* by Karim Yaghmour (O'Reilly). Copyright 2011 Karim Yaghmour, 9781449308292."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9781449308292

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

CHAPTER 1
Introduction

Putting Android on an embedded device is a complex task involving an intricate understanding of its internals and a clever mix of modifications to the Android Open Source Project (AOSP) and the kernel on which it runs, Linux. Before we get into the details of embedding Android, however, let's start by covering some essential background that embedded developers should factor in when dealing with Android, such as Android's hardware requirements and the legal framework surrounding Android and its implications within an embedded setting. First let's look at where Android comes from and how it's developed.

History

The story goes* that back in early 2002 Google's Larry Page and Sergey Brin attended a talk at Stanford about the development of the then-new Sidekick phone by Danger Inc. The speaker was Andy Rubin, Danger's CEO at the time, and the Sidekick was one of the first multi-function, Internet-enabled devices. After the talk, Larry went up to look at the device and was happy to see that Google was the default search engine. Soon after, both Larry and Sergey became Sidekick users.

Despite its novelty and enthusiatic users, however, the Sidekick didn't achieve commercial success. By 2003, Rubin and Danger's board agreed it was time for him to leave. After trying out a few things, Rubin decided he wanted to get back into the phone OS business. Using a domain name he previously-owned, *android.com*, he set out to create an open OS for phone manufacturers. After investing most of his savings on the project and having received some additional seed money, he set out to get the company funded. Soon after, in August 2005, Google acquired Android Inc. with little fanfare.

* Coinciding with Android's initial announcement in November 2007, the New York Times ran an article entitled "I, Robot: The Man Behind the Google Phone" by John Markoff giving an insightful background portrait of Andy Rubin and his career, and, by the same token, providing a lot of insight on the story behind Android. This section is partly based on that article.

Between its acquisition and its announcement to the world in November 2007, Google let little to no information out about Android. Instead, the development team worked furiously on the OS while deals and prototypes were being worked on behind the scenes. The initial announcement was made by the *Open Handset Alliance* (OHA), a group of companies unveiled for the occasion with its stated mission being the development of open standards for mobile device and Android being its first product. A year later, in September 2008, the first open source version of Android, 1.0, was made available.

Several Android versions have been released since then, and the OS's progression and development is obviously more public. As we will see later, though, much of the work on Android continues to be done behind closed doors. Table 1-1 provides a summary of the various Android releases and the most notable features found in the corresponding AOSP.

Table 1-1. Android Versions

Version	Release date	Codename	Most notable feature(s)	Open source
1.0	September 2008	*Unknown*		Yes
1.1	February 2009	*Unknown*		Yes
1.5	April 2009	Cupcake	On-screen soft keyboard	Yes
1.6	September 2009	Donut	Battery usage screen and VPN support	Yes
2.0, 2.0.1, 2.1	October 2009	Eclair	Exchange support	Yes
2.2	May 2010	Froyo	Just-In-Time (JIT) compile	Yes
2.3	December 2010	Gingerbread	SIP and NFC support	Yes
3.0	January 2011	Honeycomb	Tablet form-factor support	No
3.1	May 2011	Honeycomb	USB host support and APIs	No
4.0	December 2011 (projected)	Ice-Cream Sandwich	Merged phone and tablet form-factor support	Yes (projected)

Features and Characteristics

Google advertizes the following features about Android:

Application framework
: The application framework used by app developers to create what is commonly-referred to as Android apps. The use of this framework is documented online at *http://developer.android.com* and in books like O'Reilly's *Learning Android*.

Dalvik Virtual Machine
: The clean-room bytecode interpreter implementation used in Android as a replacement for the Sun Java VM. While the latter interprets *.class* and *.jar* files,

Dalvik interprets .dex files. These files are generated by the *dx* utility using the *.class* files generate by the Java compiler part of the JDK.

Integrated browser
: Android includes a WebKit-based browser as part of its standard list of applications. App developers can use the `WebView` class to use the WebKit engine within their own apps.

Optimized graphics
: Android provides its own 2D graphics library but relies on OpenGL ES for its 3D capabilities.

SQLite
: This is the standard SQLite database found at *http://www.sqlite.org* and made available to app developers through the application framework.

Media support
: Android provides support for a wide range of media formats through StageFright, its custom media framework. Prior to 2.2, Android used to rely on PacketVideo's OpenCore framework.

GSM telephony support
: The telephony support is hardware dependent and device manufacturers must provide a HAL module in order to enable Android to interface with their hardware. HAL modules will be discussed in the next chapter.

Bluetooth, EDGE, 3G, and WiFi
: Android includes support for most wireless connection technologies. While some are implemented in Android-specific fashion, such as EDGE and 3G, others are provided in the same as in plain Linux, as in the case of Bluetooth and WiFi.

Camera, GPS, compass, and accelerometer
: Interfacing with the user's environment is key to Android. APIs are made available in the application framework to access these devices, and some HAL modules are required to enable their support.

Rich development environment
: This is likely one of Android's greatest assets. The development environment available to developers makes it very easy to get started with Android development. A full SDK is freely available to download along with an emulator, an Eclipse plugin, and a number of debugging and profiling tools.

There are of course a lot more features that could be listed for Android, such as USB support, multitasking, multi-touch, SIP, tethering, voice-activated commands, etc., but the previous list should give you a good idea of what you'll find in Android. Also note that every new Android release brings in its own new set of features. Check the *Platform Highlights* published with every version for more information on features and enhancements.

In addition to its basic feature-set, the Android platform has a few characteritics that make it an especially interesting basis for embedded use. Here's a quick summary:

Broad app ecosystem
: At the time of this writing there were 200,000 apps in the Android Market. This compares quite favorably to the Apple App Market's 350,000 apps and ensures that you have a large pool to choose from should you want to prepackaged applications with your embedded device.†

Consistent app APIs
: All APIs provided in the application framework are meant to be forward-compatible. Hence, custom apps you develop for inclusion in your embedded system should continue working in future Android versions. In contrast, modifications you make to Android's source code are not guaranteed to continue applying or even working in the next Android release.

Replaceable components
: Because Android is open source, and as a benefit of its architecture, a lot of its components can be replaced outright. For instance, if you don't like the default app Launcher (home screen), you can write your own. More fundamental changes can also be made to Android. The GStreamer‡ developers, for example, were able to replace StageFright, the default media framework in Android, with GStreamer without modifying the app API.

Extendable
: Another benefit from Android's openness and its architecture is that adding support for additional features and hardware is relatively straightforward. You just need to emulate what the platform is doing for other hardware or features of the same type. For instance, you can add support for custom hardware to the HAL by adding a handful of files.

Customizable
: If you'd rather use existing components, such as the existing Launcher app, you can still customize them to your liking. Whether it be tuning their behavior or changing their look and feel, you are again free to modify the AOSP as needed.

Development Model

When considering whether to use Android, it's crucial that you understand the ramifications its development process may have on any modifications you make to it or any dependencies you may have towards its internals.

† Bare in mind that you likely need to enter into some kind of agreement with an app's publisher before you can package that app. The app's availability in the Android Market doesn't imply the right for you as a 3rd party to redistribute it.

‡ GStreamer is the default media framework in the Gnome desktop environment.

Differences With "Classic" Open Source Projects

Android's open source nature is one of its most trumpeted features. Indeed, as we've just seen, many of the software engineering benefits that derive from being open source apply to Android.

Despite its licensing, however, Android is unlike most open source projects in that its development is mostly done behind closed doors. The vast majority of open source projects, for example, have public mailing lists and forums where the main developers can be found interacting with each other, and public source repositories providing access to the main development branch's tip. No such thing can be found for Android.

This is best summarized by Andy Rubin himself: “Open source is different than a community-driven project. Android is light on community-driven, somewhat heavy on open source.”

Whether we like it or not, Android is mainly developed within Google by the Android development team and the public is not privy to either internal discussions or the tip of the development branch. Instead, Google makes code-drops every time a new version of Android ships on a new device, which is usually every 6 months. For instance, a few days after the Samsung Nexus S was released in December 2010, the code for the new version of the Android it was running, 2.3/Gingerbread, was made publicly available at *http://android.git.kernel.org/*.

Obviously there is a certain amount of discomfort within the open source community with the contined use of the term "open source" within the context of project whose development model contradicts the standard modus operandi typically associated with open source projects, especially given Android's popularity. The open source community has not historically been well served by projects that had adopted a similar development model.

Political issues asside, though, Android's development model means that as a developer your ability to make contributions to Android is limited. Indeed, unless you become part of the Android development team at Google, you will not be able to make contributions to the tip of the development branch. Also, save for a handful of exceptions, you will unlikely be able to discuss your enhancements one-on-one with the core development team members. However, you are still free to submit enhancements and fixes to the AOSP code dumps made available at *http://android.git.kernel.org/*.

The worst side-effect of Google's approach is that you have absolutely no way to get inside information about the platform decisions being made by the Android development team. If new features are added within the AOSP, for example, or if modifications are made to core components, you will find out how such changes are made and how they impact changes you might have made to a previous version only by analyzing the next code dump. Furthermore, you will have no way to find out the underlying requirement or restriction or issue that justified the modification or inclusion. Had this

been a true open source project, a public mailing list archive would exist where all this information, or pointers to it, would be available.

That being said, it's important to remember how significant a contribution Google is making by distributing Android under an open source license. Despite its awkward development model from an open source community perspective, it remains that Google's work on Android is a godsend for a large number of developers. Plus, it has accomplished one thing no other open source projects was ever able to: create a massively successful Linux distribution. It would, therefore, be hard to fault Android's development team for their work.

Furthermore, it can easily be argued that from a business and go-to-market perspective a community-driven process would definitely knock the wind out of any product announcements Google would attempt to release, making it impossible to create "buzz" around press announcements and the like since every new feature would be developed in the open. That is to say nothing of the non-deterministic nature of community-driven processes that can see a group of people take years to agree on the best way to implement a given feature set. And, simply based on track record, Android's success has definitely benefited from Google's ability to rapidly move it forward and generate press interest based on releases of cool new products.

Feature Inclusion, Roadmaps, and New Releases

In brief, there is no publicly available roadmap for features and capabilities in future Android releases. At best, Google will announce ahead of time the name and approximate release date of the next version. Usually, you can expect a new Android release to be made in time for the Google I/O conference, which is typically held in May, and another release by year end. What will be in that release, though, is anyone's guess.

Typically, however, Google will choose a single manufacturer to work with on the next Android release. During that period, Google will work very closely with that single manufacturer's engineers to ready up the next Android version to work on a targeted upcoming flagship device. During that period, the manufacturer's team is reported to have access to the tip of the development branch. Once the device is put on the market, the corresponding source code dump is made to the public repositories. For the next release, they choose another manufacturer and start over.

There is one notable exception to that cycle: Android 3.x/Honeycomb. In that specific case, Google has not released the source code to the corresponding flagship product, the Motorola Xoom, and all indications suggest that that code may never be publicly available. The rationale in this case seems to be that the Android development team essentially forked the Android code-base at some point in time in order to start working on getting a tablet-ready version of Android out ASAP based on market timing prerogatives. Hence, in that version, very little regard was made to preserving backward compatibility with the phone form-factor. And given that, Google did not wish to make the code available to avoid fragmentation of its platform. Instead, at the time of this writing,

plans are that both the phone and tablet form factor support will be merged into the upcoming Android 4.0/Ice-Cream Sandwich release.

Ecosystem

As of June 2011:

- 400,000 Android phones are activated each day, up from 300,000 in December 2010 and 200,000 in August of that same year.
- The Android Market contains arond 200,000 apps. In comparison, the Apple App Store has 350,000 apps.
- More than a third of all phones sold in the US are based on Android.

Android is clearly on the upswing. In fact, Gartner predicted in April 2011 that Android would hold about 50% of the smartphone market by 2015. Much as Linux disrupted the embedded market about a decade ago, Android is poised to make its mark. Not only will it flip the mobile market on its head, eliminating or sidelining even some of the strongest players, but in the embedded space it is likely going to become the de-facto standard UI for a vast majority of user-centric embedded devices.

An entire ecosystem is therefore rapidly building around Android. Silicon and system-on-chip (SoC) manufacturers such as ARM, TI, Qualcomm, Freescale, NVidia and TI have added Android support for their products, and handset and tablet manufacturers such as Motorola, Samsung, HTC, Sony-Ericsson, LG, Archos, DELL, ASUS, etc. ship an ever-increasing number of Android-equipped devices. This ecosystem also includes an increasing number of diverse players, such as Amazon, Verizon, Sprint and Barnes&Nobles, creating their own application markets.

Grass-roots communities and projects are also starting to sprout around Android, even though it is developed behind closed doors. Some of those efforts follow in the footsteps of phone modders, which essentially rely on hacking the binaries provided by the manufacturers to create their own modifications or variants, while others have a more open source tint to them, relying on the Android sources to create their own forks or additions. The *xda-developers.com* online forum, for instance, is traditionally frequented by modders, while the *cyanogenmod.com* site hosts an Android fork which modifies the Android sources to provide additional features and enhancements. Other Android forks include Replicant (*http://replicant.us*), which aims at replacing as many of the Android components with Free Software as possible, and MIUI (*http://en.miui.com/*), which provides some cool UI hacks.

A Word on the Open Handset Alliance

As I mentioned earlier, the OHA was the initial vehicle through which Android was first announced. It describes itself on its website as "... a group of 82 technology and mobile companies who have come together to accelerate innovation in mobile and offer

consumers a richer, less expensive, and better mobile experience. Together we have developed Android™, the first complete, open, and free mobile platform."

Beyond the initial announcement, however, it is unclear what role the OHA plays. For example, an attendee at the "Fireside Chat with the Android Team" at Google I/O 2010 asked the panel what privileges were confered to him as a developer for belonging to a company which is part of the OHA. After asking around the panel, the speaker essentially answered that they didn't know because they aren't the OHA. Hence, it would appear that OHA membership benefits are not clear to the Android development team itself.

The role of the OHA is further blurred by the fact that it does not seem to be a full-time organization with board members and permanent staff. Instead, it's just an "alliance." In addition, there is no mention of the OHA within any of Google's Android announcements, nor do any new Android announcements emanate from the OHA. In sum, one would be tempted to speculate that Google likely put the OHA together mainly as a marketing front to show how much industry support there was for Android, but that in practice it has little to no bearing on Android's development.

Getting "Android"

There are two main pieces required to get Android working on your embedded system: an Android-compatible Linux kernel and the Android Platform.

As of this writing, you cannot use a "vanilla" kernel from *kernel.org* to run the Platform. Instead, you need to either use one of the kernels available within the AOSP or patch a "vanilla" kernel for it to be Android-compatible. Unfortunately, while a few attempts have been made to merge the Android modifications into the mainline kernel, these efforts haven't yet succeeded. It is expected that, in the long term, the pending issues will be resolved and the mainline kernel will support Android "out of the box." For the time being, however, we must contend with the fact Android-compatible kernels are essentially forks of the mainline and will continue being so for the foreseable future.

The Android Platform is essentially a custom Linux distribution containing the user-space packages which make up what is typically called "Android." The releases listed in Table 1-1 are actually Platform releases. We will discuss the content and architecture of the Platform in the next chapter. For the moment being, keep in mind that a Platform release has a similar role to standard Linux distributions such as Ubuntu or Fedora. It's a self-coherent set of software packages that, once built, provides a specific user experience with specific tools, interfaces, and developer APIs.

While the proper term to identify the source code corresponding to the Android distribution running on top of an Android-compatible kernel is "Android Platform," it is commonly referred to as "the AOSP"—as is the case in fact throughout this book—even though the Android Open Source Project proper, which is hosted at *http://android.git.kernel.org/*, contains a few more components in addition to the Platform, such as sample Linux kernel trees and additional packages that would not typically be downloaded when the Platform is fetched using the usual *repo* command.

Hacking Binaries

Lack of access to Android sources hasn't discouraged passionate modders from actually hacking and customizing Android to their liking. For example, the fact that Android 3.x/Honeycomb isn't available hasn't precluded modders from getting it to run on the Barnes&Noble Nook. They achieved this by retrieving the executable binaries found in the emulator image provided as part of the Honeycomb SDK and used those as-is on the Nook, albeit forfeiting hardware accelaration. The same type of hack has been used to "root" or update versions of various Android components on actual devices for which the manufacturer provides no source code.

Legal Framework

Like any other piece of software, Android's use and distribution is constricted by a set of licenses, intellectual property restrictions, and market realities. Let's look at a few of these.

Code Licenses

As we discussed earlier, there are two parts to "Android": an Android-compatible Linux kernel and an AOSP release. Even though it is modified to run the AOSP, the Linux kernel continues to be subject to the same GNU GPLv2 license that it has always been under. As such, remember that you are not allowed to distribute any modifications you make to the kernel under any other license than the GPL. Hence, if you take a kernel version from *android.git.kernel.org* and modify it to make it run on your system, you are allowed to distribute the resulting kernel image in your product only so long as you abide by the GPL and, therefore, make the sources used to create the image, including your modifications, available to recipients under the terms of the GPL.

The kernel also includes a notice by Linus Torvalds in its *COPYING* file in its sources that clearly identifies that only the kernel is subject to the GPL and that applications running on top of it are **not** considered "derived works." Hence, you are free to create

applications that run on top of the Linux kernel and distribute them under the license of your choice.

These rules and their applicability are generally well understood and accepted within open source circles and by most companies that opt to support the Linux kernel or use it as the basis for their products. In addition to the kernel, a large number of key components of Linux-based distributions are typically licensed under one form or another of the GPL. The GNU C libary (glibc) and the GNU compiler (GCC), for example, are licensed under the LGPL and the GPL respectively. Important packages commonly used in embedded Linux systems such as uClibc and BusyBox are also licensed under the LGPL and the GPL.

Not everyone is comfortable with the GNU GPL, however. Indeed, the restrictions it imposes on the licensing of derived works can pose a serious challenge to large organizations, especially given geographic distribution, cultural differences amongst the various locations of development sub-units, and the reliance on external subcontractors. A manufacturer selling a product in North America, for example, might have to deal with dozens, if not hundreds, of suppliers to get that product to the market. Each of these suppliers might deliver a piece which may or may not contain GPL'd code. Yet the manufacturer whose name appears on the item sold to the customer will be bound to provide source to GPL components regardless of which supplier originated them. In addition, processes must be put in place to ensure engineers that work on GPL-based projects are abiding by the licenses.

When Google set out to work with manufacturers on their "open" phone OS, therefore, it appears that very rapidly it became clear that the GPL had to be avoided in as much as possible. In fact, other kernels than Linux were apparently considered, but Linux was chosen because it already had strong industry support, particularly from ARM silicon manufacturers, and because it was fairly well isolated from the rest of the system, so that its GPL licensing would have little impact.§

It was decided, though, that every effort would be made to make sure that the vast majority of user-space components would be based on licenses that did not pose the same logistical issues as the GPL. That is why the vast majority of the components created by Google for the AOSP are published under the Apache License 2.0 (a.k.a. ASL) with some key components, such as the Bionic library and the Toolbox utility, licensed under the BSD license. Some classic open source projects are also incorporated, mostly as-is, into the AOSP within the *external/* directory. The assumption is that their distribution in binary form should not pose any problems since they aren't meant to be typically customized by the OEM (i.e., no derived works are expected to be created) and are readily available for all to download at *android.git.kernel.org*, thereby complying, if applicable, with the GPL's requirement that redistribution of derivative works contibue being made under the GPL.

§ See this LWN post by Brian Swetland, a member of Android's kernel development team, for more information on the rationale behind these choices.

Unlike the GPL, the ASL does not require that derivative works be published under a specific license. In fact, you can choose whatever license best suits your needs for the modifications you make. Here are the relevant portions from the ASL (the full license is available at *http://www.apache.org/licenses/*):

- "Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form."
- "... You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License."

Furthermore, the ASL explicitly provides a patent license grant, meaning that you do not require any patent license from Google for using the ASL-licensed Android code. It also imposes a few "administrative" requirements such as the need to clearly mark modified files, to provide recipients with a copy of the ASL license, to preserve *NOTICE* files as-is, etc. Essentially, though, you are free to license your modifications under the terms that fit your purpose. The BSD license that covers Bionic and Toolbox allows similar binary-only distribution.

Hence, manufacturers can take the AOSP and customize it to their needs while keeping those modifications proprietary if they wish, so long as they continue abiding by the rest of the provisions of the ASL. If nothing else, this dimishes the burden of having to implement a process to track all modifications in order to provide those modifications back to recipients who would be entitled to request them had the GPL been used instead.

Adding GPL-licensed components

Although every effort has been made to keep the GPL out of Android's user-space in as much as possible, there are cases where you may want to explicitly add GPL-licensed components to your Android distribution. For example, you want to include either glibc of uClibc which are two POSIX-compliant C libraries—in contrast to Android's Bionic which is not—because you would like to run pre-existing Linux applications on Android without having to port them over to Bionic. Or you may want to use BusyBox in addition to Toolbox since the latter is much more limited in functionality that the former.

These additions may be specific to your development environment and may be removed in the final product or they may be permanent fixtures or your own customized Android. No matter which avenue you decide on, whether it be plain Android or Android

with some additional GPL packages, remember that you must follow the licenses' requirements.

Branding Use

While being very generous with Android's source code, Google controls most Android-related branding elements more strictly. Let's take a look at some of those elements and their associated terms of use. For the official list along with the official terms, have a look at *http://www.android.com/branding.html*.

Android Robot
: This is the familiar green robot seen everywhere around all things Android. Its role is similar to the Linux penguin and the permissions for its use are similarly permissive. In fact Google states that it "can be used, reproduced, and modified freely in marketing communications." The only requirement is that proper attibution be made according to the terms of the Creative Commons Attribution license.

Android Logo
: This is the set of letters in custom typeface that spell out the letters A-N-D-R-O-I-D and that appear during the device and emulator bootup, and on the *android.com*website. You are not authorized to use that logo under any circumstance.

Android Custom Typeface
: This is the custom typeface used to render the Android logo and its use is as restricted as the logo.

"Android" in Official Names
: As Google states, "the word 'Android' may be used only as a descriptor, 'for Android'" and so long as proper trademark attribution is made. You cannot, for instance, name your product "Foo Android" without Google's permission. As the FAQ for the Android Compatiblity Program (ACP), which we will cover later in this chapter, states: "... if a manufacturer wishes to use the Android name with their product ... they must first demonstrate that the device is compatible." Branding your device as being "Android" is therefore a privilege which Google intends to police. In essence, you will have to make sure your device is compliant and then talk to Google and enter into some kind of agreement with them before you can advertize your device as being "Foo Android."

"Droid" in Official Names
: You may not use "Droid" alone in a name, such as "Foo Droid" for example. For some reason the author hasn't yet entirely figured out, "Droid" is a trademark of Lucasfilm. Achieve a Jedi rank you likely must before you can use it.

"Android" in Messaging
: It is permitted to use "Android" "... in text as a descriptor, as long as it is followed by a proper generic term (e.g. "Android™ application")." And here too, proper trademark attribution must be made.

Google's Own Android Apps

While the AOSP contains a core set of applications which are available under the ASL, "Android"-branded phones usually contain an additional set of "Google" applications that are not part of the AOSP, such as the "Android Market", YouTube, "Maps and Navigation", Gmail, etc. Obviously, users expect to have these apps as part of Android, and you might therefore want to make them available on your device. If that is the case, you will need to abide by the ACP and enter in agreement with Google, very much in line with what you need to do to be allowed to use "Android" in your product's name. We will cover the ACP shortly.

Alternative App Markets

Though the main app market is the one hosted by Google and made available to users through the "Android Market" app installed on "Android"-branded devices, other players are leveraging Android's open APIs and open source licensing to offer their own alternative app markets. Such is the case of online merchants like Amazon and Barnes&Noble, as well as mobile network operators such as Verizon and Sprint. There is, in fact, nothing to the author's knowledge that precludes you from creating your own app store. There is even at least one open source project, FDroid Repository (*http://f-droid.org/repository/*), that provides both an app market application and a corresponding server backend under the GPL.

Oracle v Google

As part of acquiring Sun Microsystem, Oracle also acquired Sun's intellectual property (IP) rights to the Java language and, according to Java creator James Gosling,[‖] it was clear during the acquisition process that Oracle intended from the onset to go after Google with Sun's Java IP portfolio. And in August 2010 it did just that, filing suit against Google, claiming that it infringed on serveral patents and commited copyright violations.

Without going into the merits of the case, it's obvious that Android does indeed heavily rely on Java. And clearly Sun created Java and owned a lot of intellectual property around the language it created. In what appears to have been an effort to anticipate any claims Sun may put forward against Android, nonetheless, the Android development team went out of its way to make the OS use as little of Sun's Java as possible. Java is in fact comprised mainly of three things: the language and its semantics, the virtual machine that runs the Java bytecode generated by the Java compiler, and the class library that contains the packages used by Java applications at run time.

‖ See Gosling's blog postings on the topic at *http://nighthacks.com/roller/jag/entry/the_shit_finally_hits_the* and *http://nighthacks.com/roller/jag/entry/quite_the_firestorm* for more details.

The official versions of the Java components are provided by Oracle as part of the Java Development Kit (JDK) and the Java Runtime Environment (JRE). Android, on the other hand, relies only the Java compiler found in the JDK. Instead of using Oracle's Java VM, Android relies on Dalvik, a VM custom-built for Android, and instead of using the official class library, Android relies on Apache Harmony, a clean-room re-implementation of the class library. Hence, it would seem that Google did every reasonable effort to at least avoid any copyright and/or distribution issues.

Still, it remains to be seen where these legal proceedings will go. And it likely will take a few years to get resolved. In the interim, however, it appears that the judge presiding over the case wants to get the matter resolved in earnest. In May 2011, he ordered Oracle to cut its claims from 132 down to 3 and ordered Google to cut their prior art references down from several hundreds to 8. According to Groklaw,# he even seems to be asking the parties whether "they anticipate that a trial will end up being moot."

Another indirectly related, yet very relevant, development is that IBM joined Oracle's OpenJDK efforts in October 2010. IBM had been the main driving force behind the Apache Harmony, which is the class library used in Android, and its departure pretty much ensures the project has become orphaned. How this development impacts Android is unknown at the time of this writing.

Incidently, James Gosling joined Google in March 2011.

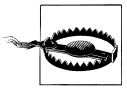
Hardware and Compliance Requirements

In principle, Android should run on any hardware that runs Linux. Android has in fact been made to run on ARM, x86, MIPS, SuperH, and PowerPC, all architectures supported by Linux. A corrolary to this is that if you want to port Android to your hardware, you must first port Linux to it. Beyond being able to run Linux, though, there are few other hardware requirements for running the AOSP, apart from the logical requirement of having some kind of display and pointer mechanism to allow users to interact with the interface. Obviously, you might have to modify the AOSP to make it work on your hardware configuration, if you don't support a peripheral it expects. For instance, if you don't have a GPS, you might want to provide a mock GPS HAL module, as the Android emulator does, to the AOSP. You will also need to make sure that you have enough memory to store the Android images and a sufficiently powerful CPU to give the user a decent experience.

In sum, therefore, there are few restrictions if you just want to get the AOSP up and running on your hardware. If, however, you are working on a device that must carry "Android" branding or must include the standard Google-owned applications found in typical consumer Android devices such as the Maps or Market applications, you need to go through the ACP that I mentioned earlier. There are two separate yet com-

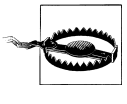
#See the full analysis here: *http://groklaw.net/article.php?story=20110505050150858*.

plementary parts to the ACP: the Compliance Definition Document (CDD) and the Compliance Test Suite (CTS). Even if you don't intend to participate in the ACP, you might still want to take a look at the CDD and the CTS, as they give a very good idea about the general mindset that went into the design goals of the Android version you intend to use.



Every Android release has its own CDD and CTS. You must therefore use the CDD and CTS that match the version you intend to use for your final product. If you switch Android releases mid-way through your project, because for instance a new Android release comes out with cool new features you'd like to have, you will need to make sure you comply with that release's CDD and CTS. Keep in mind also that you need to interact with Google to confirm compliance. Hence, switching may involve jumping through a few hoops and potential product delivery delays.

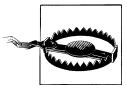
The overarching goal of the ACP, and therefore the CDD and the CTS, is to ensure a uniform ecosystem for users and application developers. Hence, before you are allowed to ship an "Android"-branded device, Google wants to make sure that you aren't fragmenting the Android ecosystem by introducing incompatible or crippled products. This, in turn, makes sense for manufacturers since they are benefiting from the compliance of others when they use the "Android" branding. For more detail about the ACP, have a look at *http://source.android.com/compatibility/*.



Note that Google reserves the right to decline your participation in the Android ecosystem, and therefore be able to ship the "Android Market" app with your device and use the "Android" branding. As stated on their site: "Unfortunately, for a variety of legal and business reasons, we aren't able to automatically license Android Market to all compatible devices."

Compliance Definition Document

The CDD is the policy part of the ACP and is available at the ACP URL above. It specifies the requirements that must be met for a device to be considered compatible. The language in the CDD is based on RFC2119 with a heavy use of "MUST," "SHOULD," "MAY," etc. to describe the different attributes. Around 25 pages in length, it covers all aspects of the device's hardware and software capabilities. Essentially, it goes over every aspect that cannot simply be automatically tested using the CTS. Let's go over some of what the CDD requires.



This discussion is based on the Android 2.3/Gingerbread CDD. The specific version you use will likely have slightly different requirements.

Software

This section lists the Java and Native APIs along with the web, virtual machine and user interface compatibility requirements. Essentially, if you are using the AOSP, you should readily conform to this section of the CDD.

Application Packaging Compatibility

This section specifies that your device must be able to install and run *.apk* files. All Android apps developed using the Android SDK are compiled into *.apk* files, and these are the files that are distributed through the Android Market and installed on users' devices.

Multimedia Compatibility

Here, the CDD describes the media codecs (decoders and encoders), audio recording, and audio latency requirements for the device. The AOSP includes the StageFright multi-media framework, and you can therefore conform to the CDD by using the AOSP. However, you should read the audio recording and latency sections, as they contain specific technical information that may impact the type of hardware or hardware configuration your device must be equipped with.

Developer Tool Compatibility

This section lists the Android-specific tools that must be supported on your device. Basically, these are the common tools used during app development and testing: *adb*, *ddms*, and *monkey*. Typically, developers don't interact with these tools directly. Instead, they usually develop within the Eclipse development environment and use the Android Development Tool (ADT), plugin which takes care of interacting the lower-level tools.

Hardware Compatibility

This is probably the most important section for embedded developers as it likely has profound ramifications on the design decisions made for the targeted device. Here's a summary of what each subsection spells out.

Display and Graphics

- Your device's screen must be at least 2.5" in physical diagonal size.
- Its density must be at least 100dpi.
- Its aspect ratio must be between 4:3 and 16:9.
- It must support dynamic screen orientation from portrait to landscape and vice-versa. If orientation can't be changed, it must support letterboxing since apps may force orientation changes.
- It must support OpenGL ES 1.0 though it may omit 2.0 support

Input Devices

- Your device must support the Input Method Framework, which allows developers to create custom on-screen, soft keyboards.
- It must provide at least one soft keyboard.
- It can't include a hardware keyboard that doesn't conform the API.
- It must provide "HOME," "MENU," and "BACK" buttons.
- It must have a touchscreen, whether it be capacitive or resistive.
- It should support independent tracked points (multi-touch) if possible.

Sensors

While all sensors are qualified using "SHOULD," meaning that they aren't compulsory, your device must accurately report the presence or absence of sensors and must return an accurate list of supported sensors.

Data Connectivity

The most important item here is an explicit statement that Android may be used on devices that don't have telephony hardware. This was added to allow for Android-based tablet devices. Furthermore, the document says that your device should have hardware support for 802.11x, Bluetooth, and NFC. Ultimately, your device must support some form of networking that permits a bandwidth of 200Kbits/s.

Cameras

Your device should include a rear-facing camera and may include a front-facing one as well.

Memory and Storage

- Your device must have at least 128MB for storing the kernel and user space.
- It must have at least 150MB for storing user data.
- It must have at least 1GB of "shared storage," essentially meaning the SD card.
- It must also provide a mechanism to access shared data from a PC. In other words, when the device is connected through USB, the content of the SD card must be accessible on the PC.

USB

This requirement is likely the one that most heavily demonstrates how user-centric "Android"-branded devices must be, since it essentially assumes the user owns the device and therefore requires you to allow users to fully control the device when it's connected to a PC. In some cases this might be a show-stopper for you, as you may not actually want or may not be able to have users connect your embedded device to a user's PC. Nevertheless, the CDD says that:

- Your device must implement a USB client which would be connectable through USB-A.

- It must implement the Android Debug Bridge (ADB) protocol as provided in the *adb* command over USB.
- It must implement USB mass storage, thereby allowing the device's SD card to be accessed on the host.

Performance Compatibility

Although the CDD doesn't specify CPU speed requirements, it does specify app-related time limitations that will impact your choice of CPU speed. For instance:

- The Browser app must launch in less than 1300ms.
- The MMS/SMS app must launch in less than 700ms.
- The AlarmClock app must launch in less than 650ms.
- Relaunching an already running app must take less time than the original launch.

Security Model Compatibility

Your device must conform to the security environment enforced by the Android application framework, Dalvik, and the Linux kernel. Specifically, apps must have access and be submitted to the permission model described as part of the SDK's documentation. Apps must also be constrained by the same sandboxing limitations they have by running as separate processes with distinct UIDs in Linux. The filesystem access rights must also conform to those described in the developer documentation. Finally, if you aren't using Dalvik, whatever VM you use should impose the same security behavior as Dalvik.

Software Compatibility Testing

Your device must pass the CTS, including the human-operated CTS Verifier part. In addition, you device must be able to run specific reference applications from the Android Market.

Updatable Software

There has to be a mechanism for your device to be updated. This may be done over-the-air (OTA) with an offline update via reboot. It also may be done using a "tethered" update via a USB connection to a PC, or be done "offline" using removable storage.

Compliance Test Suite

The CTS comes as part of the AOSP, and we will discuss it in Chapter 10. The AOSP includes a special build target that generates the *cts* command-line tool, the main interface for controling the test suite. The CTS relies on *adb* to push and run tests on the USB-connected target. The tests are build on to the JUnit Java unit testing framework and exercise different parts of the framework, such as the APIs, Dalvik, Intents, Per-

missions, etc. Once the tests are done, they will generate a *.zip* file containing XML files and screen-shots that you need to submit to *cts@android.com*.

Development Setup and Tools

There are two separate sets of tools for Android development: those used for application development and those used for platform development. If you want to set up an application development environment, have a look at *Learning Android* or Google's online documentation. If you want do platform development, as we will do here, your tools needs will vary, as we will see later in this book.

At the most basic level, though, you need to have a Linux-based workstation to build the AOSP. In fact, at the time of this writing, Google's only supported build environment is 64-bit Ubuntu 10.04. That doesn't mean that another Ubuntu version won't work or that you won't be able to build the AOSP on a 32-bit system, but essentially that configuration reflects Google's own Android compile farms configuration. An easy way to get your hands dirty with AOSP work without changing your workstation OS is to create an Ubuntu virtual machine using your favorite virtualization tool. I typically use VirtualBox, since I've found that it makes it easy to access the host's serial ports in the guest OS.

No matter what your setup is, keep in mind that the AOSP is about 4GB in size, uncompiled, and that it will grow to about 10GB once compiled. If you factor in that you are likely going to operate on a few separate versions, for testing purposes if for no other reason, you rapidly realize that you'll need tens of GBs for serious AOSP work. Also note that on what is fairly recent machine at the time of this writing (dual-core high-end laptop), it takes about an hour to build the AOSP from scratch. Even a minor modification may result in a 5 min run to complete the build. You will therefore also likely want to make sure you have a fairly powerful machine when developing Android-based embedded systems.

CHAPTER 2
Internals Primer

As we've just seen, Android's sources are freely available for you to download, modify, and install for any device you choose. In fact, it is fairly trivial to just grab the code, build it, and run it in the Android emulator. To customize the AOSP to your device and its hardware, however, you'll need to first understand Android's internals to a certain extent. So we'll get a high-level view of Android internals in this chapters, and get the opportunity in later chapters to dig into parts of internals in greater detail, including tying said internals to the actual AOSP sources.

The discussion in this book is based on Android 2.3.x/Gingerbread. Although Android's internals have remained fairly stable over its lifetime up to the time of this writing, critical changes can come unannounced thanks to Android's closed development process. For instance, in 2.2/Froyo and previous versions, the Status Bar was an integral part of the System Server. In 2.3/Gingerbread, the Status Bar was moved out of the System Server and now runs indepedently from it.*

App Developer's View

Given that Android's development API is unlike any other existing API, including anything found in the Linux world, it's important to spend some time understanding what "Android" looks like from the app developers' perspective, even though it's very different from what Android looks like for anyone hacking the AOSP. As an embedded developer working on embedding Android on a device, you might not have to actually deal directly with the idiosyncracies of Android's app development API, but some of your colleagues might. If nothing else, you might as well share a common linguo with

* Some speculate that this change was triggered because some app developers were doing too many fancy tricks with notification that were haing negative impacts on the System Server, and that the Android team hence decided to make the Status Bar a separate process from the System Server.

app developers. Of course, this section is merely a summary, and I recommend you read up on Android app development for more in-depth coverage.

Android Concepts

Application developers must take a few key concepts into account when developing Android apps. These concepts shape the architecture of all Android apps and dictate what developers can and cannot do. Overall, they make the users' life better, but they can sometimes be challenging to deal with.

Components

Android applications are made of loosely-tied *components*. Components of one app can invoke/use components of other apps. Most importantly, there is no single entry point to an Android app: no `main()` function or any equivalent. Instead, there are predefined events called *intents* that developers can tie their components to, thereby enabling their components to be activated on the occurrence of the corresponding events. A simple example is the component that handles the user's contacts database, which is invoked when the user presses a Contacts button in the Dialer or another app. An app therefore can have as many entry points as it has components.

There are four main types of components:

Activities
: Just as the "window" is the main building block of all visual interaction in window-based GUI systems, activities are the building block in an Android app. Unlike a window, however, activities cannot be "maximized," "minimized," or "resized." Instead, activities always take the entirety of the visual area and are made to be stacked up on top of each other in the same way as a browser remembers webpages in the sequence they were accessed, allowing the user to go "back" to where he was previously. In fact, as described in the previous chapter, all Android devices must have a "BACK" button to make this behavior available to the user. In contrast to web browsing, though, there is no button corresponding to the "forward" browsing action; only "back" is possible.

  One globally defined Android intent allows an activity to be displayed as an icon on the app launcher (the main app list on a phone.) Because the vast majority of apps want to appear on the main app list, they provide at least one activity that is defined as capable of responding to that intent. Typically, the user will start from a particular activity and move through other end up creating a stack of activities all related to the one they originally launched; this stack of activities is a *task*. The user can then switch to another task by clicking the HOME button and starting another activity stack from the app launcher.

Services
: Android services are akin to background processes or daemons in the Unix world. Essentially, a service is activated when another component requires its services and

typically remains active for the duration required by its caller. Most importantly, though, services can be made available to components outside an app, thereby exposing some of that app's core functionality to other apps. There is usually no visual sign of a service being active.

Broadcast Receivers
: Broadcast receivers are akin to interrupt handlers. When a key event occurs, a broadcast receiver is triggered to handle that event on the app's behalf. For instance, an app might want to be notified when the battery level is low or when "airplane mode" (to shut down the wireless connections) has been activated. When not handling a specific event for which they are registered, broadcast receivers are otherwise inactive.

Content Providers
: Content providers are essentially databases. Usually, an app will include a content provider if it needs to make its data accessible to other apps. If you're building a Twitter client app, for instance, you could give other apps on the device access the tweet feed you're presenting to the user through a content provider. All content providers present the same API to apps, regardless of how they are actually implemented internally. Most content providers rely on the SQLite functionality included in Android, but they can also use files or other types of storage.

Intents

Intents are one of the most important concepts in Android. They are the late-binding mechanisms that allow components to interact. An app developer could send an intent for an activity to "view" a web page or "view" a PDF, hence making it possible for the user to view a designated HTML or PDF document even if the requesting app itself doesn't include the capabilities to do so. More fancy use of intents is also possible. An app developer could, for instance, send a specific intent to trigger a phone call.

Think of intents as polymorphic Unix signals that don't necessarily have to be predefined or require a specific designated target component or app. The intent itself is a passive object. It's its contents (payload), the mechanism used to fire it along with the system's rules that will gate its behavior. One of the system's rules, for instance, is that intents are tied to the type of component they are sent to. An intent sent to a service, for example, can only be received by a service, not an activity or a broadcast receiver.

Components can be declared as capable of dealing with given intent types using filters in the *manifest* file. The system will thereafter match intents to that filter and trigger the corresponding component at runtime. An intent can also be sent to an explicit component, bypassing the need to declare that intent within the receiving component's filter. The explicit invocation, though, requires the app to know about the designated component ahead of time, which typically applies only when intents are sent within components of the same app.

Component Lifecycle

Another central tenant of Android is that the user should never have to manage task switching. Hence, there is no task-bar or any equivalent functionality in Android. Instead, the user is allowed to start as many apps as he wants and "switch" between apps by clicking HOME to go to the home screen and clicking on any other app. The app he clicks may be an entirely new one, or one that he previously started and for which an activity stack (a.k.a. a "task") already exists.

The corrollary to, or consequence of, this design decision is that apps gradually use up more and more system resources as they are started, which can't go on forever. At some point, the system will have to start reclaiming the resources of the least recently used or non-priority components in order to make way for newly-activated components. Yet still, this resource recycling should be entirely transparent to the user. In other words, when a component is taken down to make way for new one, and then the user returns to the original component, it should start up at the point where it was taken down and act as if it was waiting in memory all along.

To make this behavior possible, Android defines a standard *lifecycle* components. An app developer must manage her components' lifecycle by implementing a series of callbacks for each component that are triggered by events related to the component lifecycle. For instance, when an activity is no longer in the foreground (and therefore more likely to be destroyed than if it's in the foreground), its `onPause()` callback is triggered.

Managing component lifecycles is one of the greatest challenges faced by app developers, because they must carefully save and restore component states on key transitional events. The desired end result is that the user never needs to "task switch" between apps or be aware that components from previously-used apps were destroyed to make way for new ones he started.

Manifest File

If there has to be a "main" entry point to an app, the manifest file is likely it. Basically, it informs the system of the app's components, the capabilities required to run the app, the minimum level of the API required, any hardware requirements, etc. The manifest is formatted as an XML file and resides at the top-most directory of the app's sources as *AndroidManifest.xml*. The apps' components are typically all described statically in the manifest file. In fact, apart from broadcast receivers, which can be registered at runtime, all other components must be declared at build time in the manifest file.

Processes and Threads

Whenever an app's component is activated, whether it be by the system or another app, a process will be started to house that app's components. And unless the app developer does anything to overide the system defaults, all other components of that app that start after the initial component is activated will run within the same process as that component. In other words, all components of an app are comprised within a single

Linux process. Hence, developers should avoid making long or blocking operations in standard components and use threads instead.

And because the user is essentially allowed to activate as many components as he wants, several Linux processes are typically active at any time to serve the many apps containing the user's components. When there are too many processes running to allow for new ones to start, the Linux kernel's Out-Of-Memory (OOM) killing mechanisms will kick in. At that point, Android's in-kernel OOM handler will get called and it will determine which processes must be killed to make space.

Put simply, the entirety of Android's behavior is predicated on low-memory conditions.

If the developer of the app whose process is killed by Android's OOM handler has implemented his components' lifecycles properly, the user should see no adverse behavior. For all practical purposes, in fact, the user should not even notice that the process housing the app's components went away and got recreated "automagically" later.

Remote Procedure Calls (RPC)

Much like many other components of the system, Android defines its own RPC/IPC† mechanism: *Binder*. So communication across components is not done using the typical System V IPC or sockets. Instead, components use the in-kernel Binder mechanism, accessible through */dev/binder*, which will be covered later in this chapter.

App developers, however, do not use the Binder mechanism directly. Instead, they must define and interact with interfaces using Android's Interface Definition Language (IDL). Interface definitions are usually stored in an *.aidl* file and are processed by the *aidl* tool to generate the proper stubs and marshalling/unmarshalling code required to transfer objects and data back and forth using the Binder mechanism.

Framework Intro

In addition to the concepts we just discussed, Android also defines its own development framework, which allows developers to access functionality typically found in other development frameworks. Let's take a brief look at this framework and its capabilities.

User Interface

UI elements in Android include traditional widgets such as buttons, text boxes, dialogs, menus, and event handlers. This part of the API is relatively straight-forward and developers usually find their way around it fairly easily if they've already coded for any other UI framework.

All UI objects in Android are built as descendants of the `View` class and are organized within a hierarchy of `ViewGroup`s. An activity's UI can actually be specified either

† Inter-Process Communication

statically in XML (which is the usual way) or declared dynamically in Java. The UI can also be modified at runtime in Java if need be. An activity's UI is displayed when its content is set as the root of a `ViewGroup` hierarchy.

Data Storage

Android presents developers with several storage options. For simple storage needs, Android provides *shared preferences*, which allows developers to store key-pair values either in a data-set shared by all components of the app or within a specific separate file. Developers can also manipulate files directly. These files may be stored privately by the app, and therefore inaccessible to other apps, or made readable and/or writeable by other apps. App developers can also use the SQLite functionality included in Android to manage their own private database. Such a database can then be made available to other apps by hosting it within a content provider component.

Security and Permissions

Security in Android is enforced at the process level. In other words, Android relies on Linux's existing process isolation mechanisms to implement its own policies. To that end, every app installed gets its own UID and GID. Essentially, it's as if every app is a separate "user" in the system. And as in any multi-user Unix system, these "users" cannot access each others' resources unless permissions are explicitely granted to do so. In effect, each app lives in its own separate sandbox.

To exit the sandbox and access key system functionality or resources, apps must use Android's permission mechanisms, which require developers to statically declare the permissions needed by an app in its manifest file. Some permissions, such as the right to access the Internet (i.e. use sockets), dial the phone, or use the camera, are predefined by Android. Other permissions can be declared by app developers and then be required for other apps to interact with a given app's components. When an app is installed, the user is prompted to approve the permissions required to run an app.

Access enforcement is based on per-process operations and requests to access a specific URI,‡ and the decision to grant access to a specific functionality or resource is based on certificates and user prompts. The certificates are the ones used by app developers to sign the apps they make available on the Android Market. Hence, developers can restrict access to their apps' functionality to other apps they themselves created in the past.

The Android development framework provides a lot more functionality, of course, than can be covered here. I invite you to read up on Android app development elsewhere or visit *developer.android.com* for more information on 2D and 3D graphics, multi-media, location and maps, Bluetooth, NFC, etc.

‡ Universal Resource Indentifier.

App Development Tools

The typical way to develop Android applications is to use the freely available Android Software Development Kit (SDK). This SDK, along with Eclipse, its corresponding Android Development Tools (ADT) plugin, and the QEMU-based emulator in the SDK, allow developers to do the vast majority of development work straight from their workstation. Developers will also usually want to test their app on real devices prior to making it available on the Android Market, as there usually are runtime behavior differences between the emulator and actual devices. Some software publishers take this to the extreme and test their apps on several dozens of devices before shipping a new release.

Even if you aren't going to plan to develop any apps for your embedded system, I highly suggest you set up the development environment used by app developers on your workstation. If nothing else, this will allow you to validate the effects of modifications you make to the AOSP using basic test applications. It will also be essential if you plan on extending the AOSP's API and therefore create and distribute your own custom SDK.

To set up an app development environment, follow the instructions provided by Google at the developer kit site just mentioned, or have a look at the book *Learning Android* (O'Reilly).

Native Development

While the majority of apps are developed exclusively in Java using the development environment we just discussed, certain developers need to run some C code natively. To this end, Google has made the Native Development Kit (NDK) available to developers. As advertized, this is mostly aimed at game developers needing to squeeze every last possible bit of performance out of the device their game is running on. And as such, the APIs made available within the context of the NDK are mostly geared towards graphics rendering and sensor input retrieval. The infamous Angry Birds game, for example, relies heavily on code running natively.

Another possible use of the NDK is obviously to port over an existing codebase to Android. If you've developed a lot of legacy C code over several years (a common situation for development houses that have created applications for other mobile devices), you won't necessarily want to rewrite it in Java. Instead, you can use the NDK to compile it for Android and package it with some Java code to use some of the more Android-specific functionality made available by the SDK. The *Firefox* browser, for instance, relies heavily on the NDK to run some of its legacy code on Android.

As I just hinted, the nice thing about the NDK is that you can combine it with the SDK and therefore have part of your app in Java and parts of your app in C. That said, it's crucial to understand that the NDK gives you access only to a very limited subset of the Android API. There is, for instance, no way to presently send an intent from within C code compiled with the NDK; the SDK must be used to do it in Java instead. Again,

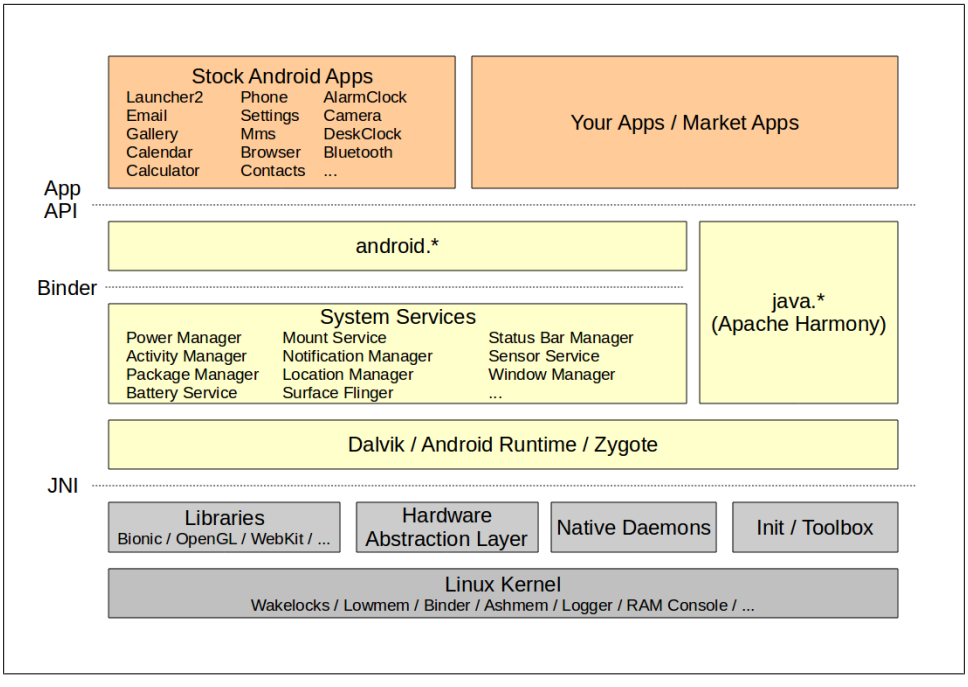


Figure 2-1. Android's architecture

the APIs made available through the NDK are mostly geared towards game development.

Sometimes embedded and system developers coming to Android expect to be able to use the NDK to do platform-level work. The word "native" in the NDK can be misleading in that regard, because the use of the NDK still involves all of the limitations and requirements that I've said to apply to Java app developers. So, as an embedded developer, remember that the NDK is useful for app developers to run C code that they can call from their Java code. Apart from that, the NDK will be of little to no use for the type of work you are likely to undertake.

Overall Architecture

Figure 2-1 is probably one of the most important diagrams presented in this book, and I suggest you find a way to bookmark its location as we will often refer back to it, if not explicitly then implicitely. Although it's a simplified view—and we will get the chance to enrich it as we go—it gives a pretty good idea of Android's architecture and how the various bits and pieces fit together.

If you are familiar with some form of Linux development, the first thing that should strike you is that beyond the Linux kernel itself, there is little in that stack that resembles anything typically seen in the Linux or Unix world. There is no glibc, no X Window

System, no GTK, no BusyBox, and so on. Many veteran Linux and embedded Linux practitioners have indeed noted that Android feels very alien. Though the Android stack starts from a clean slate with regards to user-space, we will discuss how to get "legacy" or "classic" Linux applications and utilities to coexist side-by-side with the Android stack.

The Google developer documentation presents a different architectural diagram from that shown in Figure 2-1. The former is likely well suited for app developers, but omits key information that must be understood by embedded developers. For instance, Google's diagram and developer documentationthere offer little to no reference at the time of this writing to the System Server. Yet, as an embedded developer, you need to know what that component is, because it's one of the most important parts of Android and you might need to extend or interact with it directly.

This is especially important to understand because you'll see Google's diagram presented and copied in several documents and presentations. If nothing else, remember that the System Server is rarely if at all exposed to app developers and that the bulk of information out there is aimed at app developers, not developers doing platform work.

Let's take a deeper look into each part of Android's architecture, starting from the bottom of Figure 2-1 and going up. Once we are done covering the various components, we'll end this chapter by going over the system's startup process.

Linux Kernel

The Linux kernel is the center-piece of all distributions traditionally labeled as "Linux," including mainstream distributions such as Ubuntu, Fedora, and Debian. And while it's available in "vanilla" form from the Linux Kernel Archives, most distributions apply their own patches to it to fix bugs and enhance the performance or customize the behavior of certain aspects of it before distributing it to their users. Android, as such, is no different in that the Android developers patch the "vanilla" kernel to meed their needs.

Android differs from standard practice, however, in relying on several custom functionalities that are significantly different from what is found in the "vanilla" kernel. In fact, whereas the kernel shipped by a Linux distribution can easily be replaced by a kernel from *kernel.org* with little to no impact to the rest of the distribution's components, Android's user-space components will simply not work unless they're running on an "Androidized" kernel. As I had mentioned in the previous chapter, Android kenels are, in sum, forks from the mainline kernel.

Although it's beyond the scope of this book to discuss the Linux kernel's internals, let's go over the main "Androidisms" added to the kernel. You can get information about

the kernel's internals by having a look at Robert Love's *Linux Kernel Development, 3rd ed.* and starting to follow the Linux Weekly News (LWN) site. LWN contains several seminal articles on the kernel's internals and provides the latest news regarding the Linux kernel's development.

Note that the following subsections cover only the most important Androidisms. Android-ified kernels typical contain several hundred patches over the standard kernel, often to provide device-specific functionality, fixes and enhancements. You can use *git* to do an exhaustive analysis on the commit deltas between one of the kernels at *http://android.git.kernel.org* and the mainline kernel they were forked from. Also, note that some of the functionality that appears in some Android-ified kernels, such as the PMEM driver for instance, is device-specific and isn't necessarily used in all Android devices.

Wakelocks

Of all the Androidisms, this is likely the most contentious. The discussion threads covering its inclusion into the mainline kernel generated close to 2,000 emails and yet still, there's no clear path for merging the wakelock functionality.

To understand what wakelocks are and do, we must first discuss how power management is typically used in Linux. The most common use case of Linux's power management is a laptop computer. When the lid is closed on a laptop running Linux, it will usually go into "suspend" or "sleep" mode. In that mode, the system's state is preserved in RAM but all other parts of the hardware are shut down. Hence, the computer uses as little battery power as possible. When the lid is raised, the laptop "wakes up" and the user can resume using it almost instantaneously.

That modus operandi works fine for a laptop and desktop-like devices, but it doesn't fit mobile devices such as handsets as well. Hence, Android's development team devised a mechanism that changes the rules slightly to make them more palatable to such use cases. Instead of letting the system be put to sleep at the user's behest, an Androidized kernel is made to go to sleep as soon and as often as possible. And to keep the system from going to sleep while important processing is being done or while an app is waiting for the user's input, wakelocks are provided to... keep the system awake.

The wakelocks and early suspend functionality are actually built on top of Linux's existing power management functionality. However, they introduce a different development model, since application and driver developers must explicitely grab wakelocks whenever they conduct critical operations or must wait for user input. Usually, app developers don't need to deal with wakelocks directly, because the abstractions they use automatically take care of the required locking. They can, nonetheless, communicate with the Power Manager Service if they require explicit wakelocks. Driver developers, on the other hand, can call on the added in-kernel wakelock primitives to grab and release wakelocks. The downside of using wakelocks in a driver, however, is that

it becomes impossible to push that driver into the mainline kernel, because the mainline doesn't include wakelock support.

The following LWN articles describe wakelocks in more detail and explain the various issues surrounding their inclusion into the mainline kernel:

- Wakelocks and the embedded problem
- From wakelocks to a real solution
- Suspend block
- Blocking suspend blockers
- What comes after suspend blockers
- An alternative to suspend blockers

Low Memory Killer

As I mentioned earlier, Android's behavior is very much predicated on low-memory conditions. Hence, out-of-memory behavior is crucial. For this reason, the Android development team has added an additional low memory killer to the kernel that kicks in before the default kernel OOM killer. Android's low-memory killer applies the policies described in the app development documentation, weeding out processes hosting components that haven't been used in a long time and that are not high-priority.

This low memory killer is based on the OOM adjustments mechanism availalble in Linux that enables the enforcement of different OOM kill priorities for different processes. Basically, the OOM adjustments allow user space to control part of the kernel's OOM killing policies. The OOM adjustments range from -17 to 15, with a higher number meaning the associated process is a better candidate for being killed if the system is out of memory.

Android therefore attributes different OOM adjustment levels to different types of processes according to the components they are running, and configures its own low memory killer to apply different thresholds for each category of process. This effectively allows it to preempt the activation of the kernel's own OOM killer—which only kicks in when the system has no memory left—by kicking in when the given thresholds are reached, not when the system runs out of memory.

The user-space policies are themselves applied by the init process at startup (see “Init” on page 47), and readjusted and partly enforced at runtime by the Activity Manager Service, which is part of the System Server. The Activity Manager is one of the most important services in the System Server and is responsible, amongst many other things, for carrying out the component lifecycle presented earlier.

Have a look at the Taming the OOM killer LWN article if you'd like to get more information regarding the kernel's OOM killer and how Android builds on it.

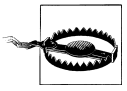
Binder

Binder is an RPC/IPC mechanism akin to COM under Windows. Its roots actually date back to work done within BeOS prior to Be's assets being bought by Palm. It continued life within Palm and the fruits of that work were eventually released as the *OpenBinder* project. Though OpenBinder never survived as a stand-alone project, a few key developers who had worked on it, such as Dianne Hackborn and Arve Hjønnevåg, eventually ended up working within the Android development team.

Android's Binder mechanism is therefore inspired by that previous work, but Android's implementation does not derive from the OpenBinder code. Instead, it's a clean room rewrite of a subset of the OpenBinder functionality. The OpenBinder Documentation remains a must-read if you want to understand the mechanism's underpinings and its design philosophy.

In essence, Binder attempts to provide remote object invocation capabilities on top of a classic OS. In other words, instead of re-engineering traditional OS concepts, Binder "attempts to embrace and transcend them." Hence, developers get the benefits of dealing with remote services as objects without having to deal with a new OS. It therefore becomes very easy to extend a system's functionality by adding remotely-invocable objects instead of implementing new daemons for providing new services, as would usually be the case in the Unix philosophy. The remote object can therefore be implemented in any desired language and may share the same process space as other remote services or have its own separate process. All that is needed to invoke its methods is its interface definition and a reference to it.

And as you can see in Figure 2-1, Binder is a conerstone of Android's architecture. It's what allows apps to talk the System Server and it's what apps use to talk to each others' service components, although, as I mentioned earlier, app developers don't actually talk to the Binder directly. Instead, they use the interfaces and stubs generated by the *aidl* tool. Even when apps interface with the System Server, the `android.*` APIs abstract its services and the developer never actually sees that Binder is being used.



Though they sound semantically similar, there is a very big difference between services running within the System Server and services exposed to other apps through the "service" component model I introduced in “Components” on page 22 as being one of the components available to app developers. Most importantly, service components are subject to the same system mechanics as any other component. Hence, they are lifecycle-managed and run within the same priviledge sandbox associated as the app they are part of. Services running within the System Server, on the other hand, typically run with system priviledges and live from boot to reboot. The only things these two types of services share together are: a) their name, b) the use of Binder to interact with them.

The in-kernel driver part of the Binder mechanism is a character driver accessible through */dev/binder*. It's used to transmit parcels of data in between the communicating parties using calls to `ioctl()`. It also allows one process to designate itself as the "Context Manager." The importance of the Context Manager along with the actual user-space use of the Binder driver will be discussed in more detail later in this chapter.

Anonymous Shared Memory (ashmem)

Another IPC mechanism available in most OSes is shared memory. In Linux, this is usually provided by the POSIX SHM functionality, part the System V IPC mechanisms. If you look at the *ndk/docs/system/libc/SYSV-IPC.html* file included in the AOSP, however, you'll discover that the Android development team seems to have a dislike for SysV IPC. Indeed, the argument is made in that file that the use of SysV IPC mechanisms in Linux can lead to resource leakage within the kernel, opening the door in turn for malicious or misbehaving software to cripple the system.

Though it isn't stated as such by Android developers or any of the documentation within the ashmem code or surrounding its use, ashmem very likely owes part of its existence to SysV IPC's shortcomings as seen by the Android development team. Ashmem is therefore described as being similar to POSIX SHM "but with different behavior." For instance, it does reference counting to destroy memory regions when all processes referring to them have exited and will shrink mapped regions if the system is in need of memory. It will also enable memory regions to be shrunk in case the system is under memory pressure. "Unpinning" a region allows it to be shrunk, whereas "pinning" a region disallows the shrinking.

Typically, a first process creates a shared memory region using ashmem and uses Binder to share the corresponding file descriptor with other processes with which it wishes to share the region. Dalvik's JIT code cache, for instance, is provided to Dalvik instances through ashmem. A lot of System Server components, such as the Surface Flinger and

the Audio Flinger, rely on ashmem, though not directly but through the `IMemory`[§] interface.

Alarm

The alarm driver added to the kernel is another case where the default kernel functionality wasn't sufficient for Android's requirements. Android's alarm driver is actually layered on top of the kernel's existing Real-Time Clock (RTC) and High-Resolution Timers (HRT) functionalities. The kernel's RTC functionality provides a framework for driver developers to create board-specific RTC functions, while the kernel exposes a single hardware-independent interface through the main RTC driver. The kernel HRT functionality, on the other hand, allows callers to get woken up at very specific points in time.

In "vanilla" Linux, application developers typically call the `setitimer()` system call to get a signal when a given time value expires.[‖] The system call allows for a handful of types of timers, one of which, `ITIMER_REAL`, uses the kernel's High-Resolution Timer (HRT). This functionality, however, doesn't work when the system is suspended. In other words, if an application uses `setitimer()` to request being woken up at a given time and then, in the interim, the device is suspended, that application will get its signal only when the device is woken up again.

Separately from the `setitimer()` system call, the kernel's RTC driver is accessible through */dev/rtc* and enables its users to use an `ioctl()`, among other things, to set an alarm that will be activated by the RTC hardware device in the system. That alarm will fire off whether the system is suspended or not, since it's predicated on the behavior or the RTC device, which remains active even when the rest of the system is suspended.

Android's alarm driver cleverly combines the best of both worlds. By default, the driver uses the kernel's High-Resolution Timer (HRT) functionality to provide alarms to its users, much like the kernel's own built-in timer functionality. However, if the system is about to suspend itself, it programs the RTC so that the system gets woken up at the appropriate time. Hence, whenever an application from user space needs a specific alarm, it just needs to use Android's alarm driver to be woken up at the appropriate time, regardless of whether the system is suspended in the interim.

From user-space, the alarm driver appears as the */dev/alarm* character device and allows its users to set up alarms and adjust the system's time (wall time) through `ioctl()` calls. There are a few key AOSP components that rely on */dev/alarm*. For instance, Toolbox and the `SystemClock` class, available through the app development API, rely on it to set/get the system's time. Most importantly, though, the Alarm Manager service part of the

§ `IMemory` is an internal interface available only within the AOSP, not to app developers. The closest class exposed to app developers is `MemoryFile`.

‖ For more information, see the `setitimer()`'s man page.

System Server uses it to provide alarm services to apps that are exposed to app developers through the `AlarmManager` class.

Both the driver and Alarm Manager use the wakelock mechanism wherever appropriate to maintain consistency between alarms and the rest of Android's wakelock-related behavior. Hence, when an alarm is fired, its consuming app gets the chance to do whatever operation is required before the system is allowed to suspend itself again, if need be.

Logger

Logging is yet another essential component of any Linux system, embedded ones included. Being able to analyze a system's logs for errors or warnings either in post-mortem or in real-time can be vital to isolate fatal errors, especially transient ones. By default, most Linux distributions include two logging systems: the kernel's own log, typically accessed through the *dmesg* command, and the system logs, typically stored in files in the */var/log* directory. The kernel's log usually contains the messages printed out by the various `printk()` calls made within the kernel, either by core kernel code or by device drivers. For their part, the system logs contain messages coming from various daemons and utilities running in the system. In fact, you can use the *logger* command to send your own messages to the system log.

With regard to Android, the kernel's logging functionality is used as-is. However, none of the usual system logging software packages typically found in most Linux distributions is found in Android. Instead, Android defines its own logging mechanisms based on the Android logger driver added to the kernel. syslog relies on sending messages through sockets, and therefore generates a task switch. It also uses files to store its information, therefore generating writes to a storage device. In contrast, Android's logging functionality manages a handful of separate kernel-hosted buffers for logging data coming from user-space. Hence, no task-switches or file writes are required for each event being logged. Instead, the driver maintains circular buffers where it logs every incoming event and returns immediately back to the caller.

Because of its light-weight and efficient design, Android's logger can actually be used by user-space components at run-time to regularly log events. In fact, the `Log` class available to app developers more or less directly invokes the logger driver to write to the main event buffer. Obviously, all good things can be abused and it's preferable to keep the logging light, but still the level of use made possible by exposing `Log` through the app API along with the level of use of logging within the AOSP itself would have likely been very difficult to sustain had Android's logging been based on syslog.

Figure 2-2 describes Android's logging framework in more detail. As you can see, the logger driver is the core building block on which all other logging-related functionality relies. Each buffer it manages is exposed as a separate entry within */dev/log/*. However, no user-space component directly interacts with that driver. Instead, they all rely on liblog which provides a number of different logging functions. Depending on the func-

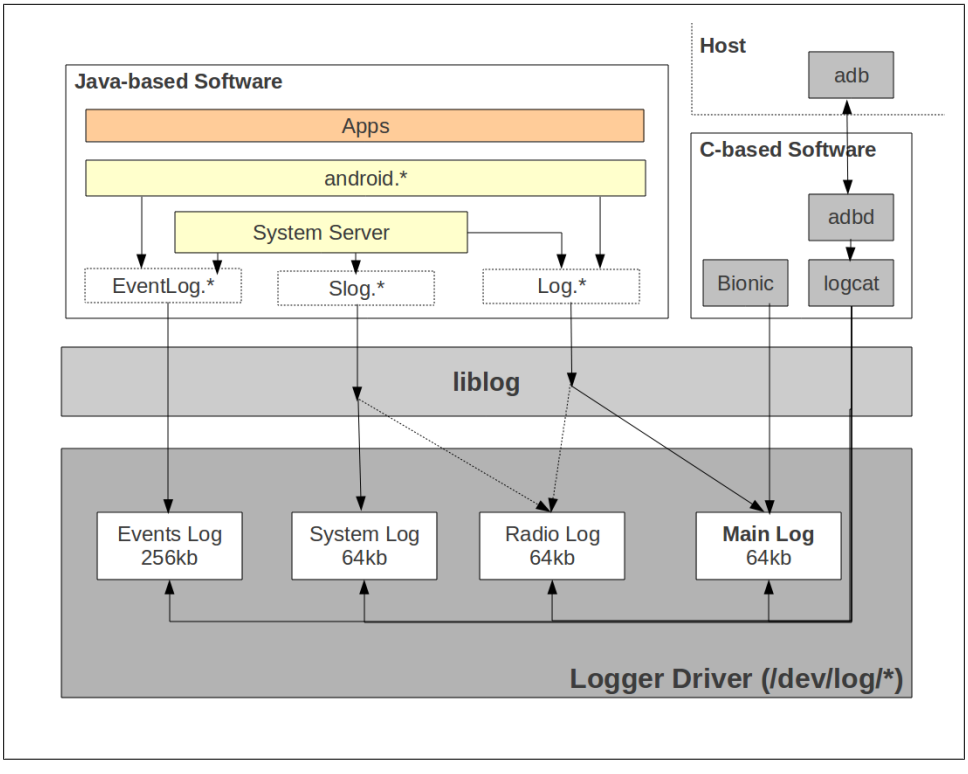


Figure 2-2. Android's logging framework

tions being used and the parameters being passed, events will get logged to different buffers. The liblog functions used by the `Log` and `Slog` classes, for instance, will test whether the event being dispatched comes from a radio-related module. If so, the event is sent to the "radio" buffer. If not, the `Log` class will send the event to the "main" buffer whereas the `Slog` class will send it to the "system" buffer. The "main" buffer is the one whose events are shown by the *logcat* command when it's issued without any parameters.

Both the `Log` and `EventLog` classes are exposed through the app development API, while `Slog` is for internal AOSP use only. Despite being available to app developers, though, `EventLog` is clearly identified in the documentation aas mainly or system integrators, not app developers. In fact, the vast majority of code samples and examples provided as part of the developer documentation use the `Log` class. Typically, `EventLog` is used by system components to log binary events to the Android's "events" buffer. Some system components, especially System Server-hosted services, will use a combination of `Log`, `Slog`, and `EventLog` to log different events. An event that might be relevant to app developers, for instance, might be logged using `Log`, while an event relevant to platform developers or system integrators might be logged using either `Slog` or `EventLog`.

Note that the `logcat` utility, which is commonly used by app developers to dump the Android logs, also relies on liblog. In addition to providing access functions to the logger driver, liblog also provides functionality for formatting events for pretty printing and filtering. Another feature of liblog is that it requires every event being logged to have a priority, a tag, and data. The priority is one of `verbose`, `debug`, `info`, `warn`, or `error`. The tag is a unique string that identifies the component or module writing to the log, and the data is the actual information that needs to be logged. This description should in fact sound fairly familiar to anyone exposed to the app developpment API, as this is exactly what's spelled out by the developer documentation for the `Log` class.

The final piece of the puzzle here is the *adb* command. As we'll discuss later, the AOSP includes an Android Debug Bridge (ADB) daemon that runs on the Android device and that is accessed from the host using the *adb* command-line tool. When you type *adb logcat* on the host, the daemon actually launches the *logcat* command locally on the target to dump its "main" buffer and then transfers that back to the host to be shown on the terminal.

Other Notable Androidisms

A few other Androidisms, in addition to those already covered, are worth mentioning, even if we don't cover them in as much detail.

Paranoid Networking
: Usually in Linux, all processes are allowed to create sockets and interact with the network. Per Android's security model, however, access to network capabilities has to be controlled. Hence, an option is added to the kernel to gate access to socket creation and network interface administration based on whether the current process belongs to a certain group of processes or possesses certain capabilities. This applies to IPv4, IPv6, and Bluetooth.

RAM Console
: As I mentioned earlier, the kernel manages its own log, which you can access using the *dmesg* command. The content of this log is very useful, as it often contains critical messages from drivers and kernel subsystems. On a crash or a kernel panic, its content can be instrumental for post-mortem analysis. Since this information is typically lost on reboot, Android adds a driver that registers a RAM-based console that survives reboots and makes its content accessible through */proc/last_kmsg*.

Physical Memory (pmem)
: Like ashmem, the pmem driver allows for sharing memory between processes. However, unlike ashmem, it allows the sharing of large chunks of physically-contiguous memory regions, not virtual memory. In addition, these memory regions may be shared between processes and drivers. For the G1 handset, for instance, pmem heaps are used for 2D hardware acceleration. Note, though, that pmem isn't used in all devices. In fact, according to Brian Swetland, one of the Android kernel

development team members, it was written to specifically target the MSM7201A's# limitations.

Hardware Support

Android's hardware support approach is significantly different from the classic approach typically found in the Linux kernel and in Linux-based distributions. Specifically, the way hardware support is implemented, the abstractions built on that hardware support, and the mindset surrounding the licensing and distribution of the resulting code are all different.

The Linux Approach

The usual way to provide support for new hardware in Linux is to create device drivers that are either built as part of the kernel or loaded dynamically at runtime through modules. The corresponding hardware is thereafter generally accessible in user-space through entries in */dev*. Linux's driver model defines three basic types of devices: character devices, devices that appear as a stream of bytes, block devices (essentially hard disks), and networking devices. Over the years, quite a few additional device and subsystem types have been added, such as for USB or MTD devices. Nevertheless, the APIs and methods for interfacing with the */dev* entry corresponding to a given type of device have remained fairly standardized and stable.

This, therefore, has allowed various software stacks to be built on top of */dev* nodes to either interact with the hardware directly or expose generic APIs that are used by user applications to provide access to the hardware. The vast majority of Linux distributions in fact ship with a similar set of core libraries and subsystems, such as the ALSA audio libraries and the X Window System, to interface with hardware devices exposed through */dev*.

With regard to licensing and distribution, the general "Linux" approach has always been that drivers should be merged and maintained as part of the mainline kernel and distributed with it under the terms of the GPL. So, while some device drivers are developed and maintained independently and some are even distributed under other licenses, the consensus has been that that isn't the preferred approach. In fact, with regard to licensing, non-GPL drivers have always been a contentious issue. Hence, the conventional wisdom is that users' and distributors' best bet to get the latest drivers is usually to get the latest mainline kernel from *http://kernel.org*. This has been true since the kernel's early days and remains true despite some additions having been made to the kernel to allow the creation of user-space drivers.

#The MSM7201A is the G1's processor.

Android's General Approach

Although Android builds on the kernel's hardware abstractions and capabilities, its approach is very different. On a purely technical level, the most glaring difference is that its subsystems and libraries don't rely on standard */dev* entries to function properly. Instead, the Android stack typically relies on shared libraries provided by manufacturers to interact with hardware. In effect, Android relies on what can be considered a Hardware Abstraction Layer (HAL), although, as we will see, the interface, behavior and function of abstracted hardware components differ greatly from type to type.

In addition, most software stacks typically found in Linux distributions to interact with hardware are not found in Android. There is no X Window System, for instance, and while ALSA drivers are sometimes used—a decision left up to the hardware manufacturer who provides the shared library implementing audio support for the HAL—access to their functionality is different from that on standard Linux distributions.

Figure 2-3 presents the typical way in which hardware is abstracted and supported in Android, and the corresponding distribution and licensing. As you can see, Android still ultimately relies on the kernel to access the hardware. However, this is done through shared libraries that are either implemented by the device manufacturer or provided as part of the AOSP.

One of the main features of this approach is that the license under which the shared library is distributed is up to the hardware manufacturer. Hence, a device manufacturer can create a simplistic device driver that implements the most basic primitives to access a given piece of hardware and make that driver available under the GPL. Not much would be revealed about the hardware, since the driver wouldn't do anything fancy. That driver would then expose the hardware to user-space through `mmap()` or `ioctl()` and the bulk of the intelligence would be implemented within a proprietary shared library in user-space that uses those functions to drive the hardware.

Android does not in fact specify how the shared library and the driver or kernel subsystem should interact. Only the API provided by the shared library to the upper layers is specified by Android. Hence, it's up to you to determine the specific driver interface that best fits your hardware, so long as the shared library you provide implements the appropriate API. Nevertheless, we will cover the typical methods used by Android to interface to hardware in the next section.

Where Android is relatively inconsistent is the way the hardware-supporting shared libraries are loaded by the upper layers. Remember for now that for most hardware types, there has to be a *.so* file that is either provided by the AOSP or that you must provide for Android to function properly.

No matter which mechanism is used to load a hardware-supporting shared library, a system service corresponding to the type of hardware is typically responsible for loading and interfacing with the shared libary. That system service will be responsible for interacting and coordinating with the other system services to make the hardware behave

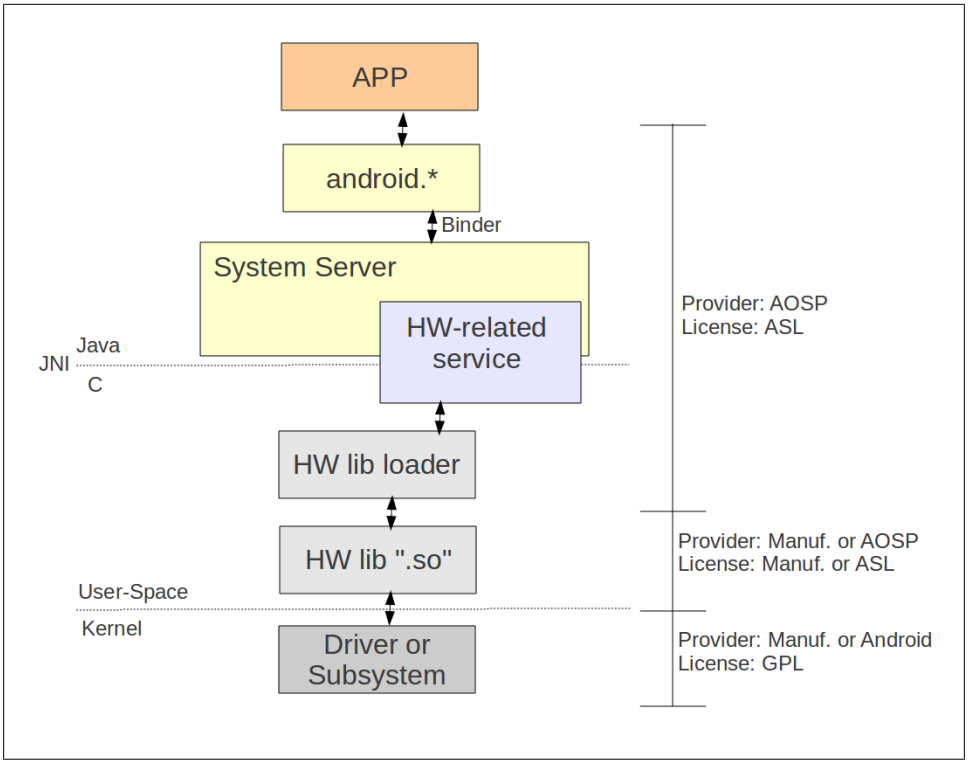


Figure 2-3. Android's "Hardware Abstraction Layer"

coherently with the rest of the system and the APIs exposed to app developers. If you're adding support for a given type of hardware, it's therefore crucial that you try to understand in as much detail as possible the internals of the system service corresponding to your hardware. Usually, the system service will be split in two parts, one part in Java that implements most of the Android-specific intelligence and another part in C whose main job is to interact with the hardware-supporting shared library and other low-level functions.

Loading and Interfacing Methods

As I mentioned earlier, there are various ways in which system services and Android in general interact with the shared libraries implementing hardware support and hardware devices in general. It's difficult to fully understand why there is such a variety of methods, but I suspect that some of them evolved organically. Luckily, there seems to be a movement towards a more uniform way of doing things. Given that Android moves at a fairly rapid pace, this is one area that will require keeping an eye on for the forseeable future, as it's likely to evolve.

Note that the methods described here are not necessarily mutually exclusive. Often a combination of these is used within the Android stack to load and interface with a shared library or some software layer before or after it. I'll cover specific hardware in the next section.

`dlopen()`-*loading through HAL*
: *Applies to: GPS, Lights, Sensors, and Display*

  Some hardware-supporting shared libraries are loaded by the *libhardware* library. This library is part of Android's HAL and exposes `hw_get_module()`, which is used by some system services and subsystems to explicitly load a given specific hardware-supporting shared library (a.k.a. a "module" in HAL terminology*). `hw_get_module()` in turn relies on the classic `dlopen()` to load libraries into the caller's address space.

Linker-loaded .so files
: *Applies to: Audio, Camera, Wifi, Vibrator, and Power Management*

  In some cases, system services are simply linked against a given *.so* file at build time. Hence, when the corresponding binary is run, the dynamic linker automatically loads the shared library into the process's address space.

Hardcoded `dlopen()`*s*
: *Applies to: StageFright and Radio Interface Layer (RIL)*

  In a few cases, the code invokes `dlopen()` directly instead of going through `libhardware` to fetch a hardware-enabling shared library. The rationale for using this method instead of the HAL is unclear.

Sockets
: *Applies to: Bluetooth, Network Management, Disk Mounting, and Radio Interface Layer (RIL)*

  Sockets are sometimes used by system services or framework components to talk to a remote daemon or service that actually interacts with the hardware.

Sysfs entries
: *Applies to: Vibrator and Power Management*

  Some entries in sysfs (*/sys*) can be used to control the behavior of hardware and/or kernel subsystems. In some cases, Android uses this method instead of */dev* entries to control the hardware.

/dev nodes
: *Applies to: Almost every type of hardware*

  Aguably, any hardware abstraction must at some point communicate with an entry in */dev*, because that's how drivers are exposed to user-space. Some of this com-

* Not to be confused with loadable kernel modules, which are a completely different and unrelated software construct, even though they share some similar properties.

munication is likely hidden to Android itself because it interacts with a shared library instead, but in some other cases AOSP components directly access device nodes. Such is the case of input libraries used by the Input Manager.

D-Bus

Applies to: Bluetooth

D-Bus is a classic messaging system found in most Linux distributions for facilitating communication between various desktop components. It's included in Android because it's the prescribed way for a non-GPL component to talk to the GPL-licensed BlueZ stack—Linux's default Bluetooth stack and the one used in Android—without being subject to the GPL's redistribution requirements; D-Bus itself being dual-licensed under the Academic Free License (AFL) and the GPL. Have a look at *http://dbus.freedesktop.org* for more information about D-Bus.

Device Support Details

Table 2-1 summarizes the way in which each type of hardware is supported in Android. As you'll notice, there is a wide variety of combinations of mechanisms and interfaces. If you plan on implementing support for a specific type of hardware, the best way forward is to start from an existing sample implementation. The AOSP typically includes hardware support code for a few handsets, generally those which were used by Google to develop new Android releases and therefore served as flagship devices. Sometimes the sources for the hardware support are quite extensive, as was the case for the Samsung Nexus S (a.k.a. "Crespo", its code-name).

The only type of hardware for which you are unlikely to find publicly-available implementations on which to base your own is the RIL. For various reasons, it's best not to let everyone be able to play with the airwaves. Hence, manufacturers don't make such implementations available. Instead, Google provides a reference RIL implementation in the AOSP should you want to implement a RIL.

Table 2-1. Android's hardware support methods and interfaces

Hardware	System Service	Interface to user-space HW support	Interface to HW
Audio	Audio Flinger	Linker-loaded *libaudio.so*	Up to HW manufacturer, though ALSA is typical
Bluetooth	Bluetooth Service	Socket/D-Bus to BlueZ	BlueZ stack
Camera	Camera Service	Linker-loaded *libcamera.so*	Up to HW manufacturer, sometimes Video4Linux
Display	Surface Flinger	HAL-loaded *gralloc* module	*/dev/fb0* or */dev/graphics/fb0*
GPS	Location Manager	HAL-loaded *gps* module	Up to HW manufacturer
Input	Input Manager	Native library	Entries in */dev/input/*

Hardware	System Service	Interface to user-space HW support	Interface to HW
Lights	Lights Service	HAL-loaded *lights* module	Up to HW manufacturer
Media	N/A, StageFright framework within Media Service	dlopen on *libstagefrighthw.so*	Up to HW manufacturer
Network interfaces[a]	Network Management Service	Socket to *netd*	`ioctl()` on interfaces
Power Management	Power Manager Service	Linker-loaded libhardware_legacy.so	Entries in */sys/android_power/* or */sys/power/*
Radio (Phone)	N/A, entry-point is telephony Java code	Socket to *rild*, which itself does a `dlopen()`on manufacturer-provided *.so*	Up to HW manufacturer
Storage	Mount Service	Socket to *vold*	System calls
Sensors	Sensor Service	HAL-loaded *sensors* module	Up to HW manufacturer
Vibrator	Vibrator Service	Linker-loaded libhardware_legacy.so	Up to HW manufacturer
Wifi	Wifi Service	Linker-loaded libhardware_legacy.so	Classic *wpa_supplicant*[b]

[a] This is for Tether, NAT, PPP, PAN, USB RNDIS (Windows). It isn't for Wifi.

[b] The *wpa_supplicant* is the same software package used on any Linux desktop to manage Wifi networks and connections.

Native User-Space

Now that we've covered the low-level layers on which Android is built, let's start going up the stack. First off, we'll cover the native user-space environment in which Android operates. By "native user-space" I mean all the user-space components that run outside the Dalvik virtual machine. This includes quite a few binaries that are compiled to run natively on the target's CPU architecture. These are generally started either automatically or as needed by the init process according to its configuration files, or are available to to be invoked on the command line once a developer shells into the device. Such binaries usually have direct access the root filesystem and the native libraries included in the system. Their capabilities would be gated by the filesystem rights granted to them and wouldn't be subject to any of the restrictions imposed on a typical Android app by the Android framework because they are running outside of it.

Note that Android's user-space was designed pretty much from a blank slate and differs greatly from what you'd find in a standard Linux distribution. Hence, I will try in as much as possible in the following to explain where Android's user-space is different or similar to what you'd usually find in a Linux-based system.

Filesystem layout

Like any other Linux-based distribution, Android uses a root filesystem to store applications, libraries, and data. Unlike the vast majority of Linux-based distributions, however, the layout of Android's root filesystem does not adhere to the Filesystem Hierarchy Standard (FHS).† The kernel itself doesn't enforce the FHS, but most software packages built for Linux assume that the root filesystem they are running on conforms to the FHS. Hence, if you intend to port a standard Linux application to Android, you'll likely need to do some legwork to ensure that the filepaths it relies on are still valid.

Given that most of the packages running in Android's user space were written from scratch specifically for Android, this lack of conformity is of little to no consequence to Android itself. In fact, it has some benefits, as we'll see shortly. Still, it's important to learn how to navigate Android's root filesystem. If nothing else, you'll likely have to spend quite some time inside of it as you bring Android up on your hardware or customize it for that hardware.

The two main directories in which Android operates are */system* and */data*. These directories do not emanate from the FHS. In fact, I can't think of any mainstream Linux distribution that uses either of these directories. Rather, they reflect the Android development team's own design. This is one of the first signs hinting to the fact that it might be possible to host Android side-by-side with a common Linux distribution on the same root filesystem. As I said earlier, we'll actually examine this possibility in more detail later in the book.

/system is the main Android directory for storing immutable components generated by the build of the AOSP. This includes native binaries, native libraries, framework packages, and stock apps. It's usually mounted from a separate image from the root filesystem, which is itself mounted from a RAM disk image. */data*, on the other hand, is Android's main directory for storing data and apps that change over time. This includes the data generated and stored by apps installed by the user alongside data generated by Android system components at runtime. It too is usually mounted from its own separate image.

Android also includes many directories commonly found in any Linux system, such as: */dev*, */proc*, */sys*, */sbin*, */root*, */mnt*, and */etc*. These directories often serve similar if not identical purposes to the the ones they serve on any Linux system, although they are very often trimmed down, as is the case of */sbin* and */etc*, and in some cases are empty, such as */root*.

Interestingly, Android doesn't include any */bin* or */lib* directories. These directories are typically crucial in a Linux system, containing, respectively, essential binaries and es-

† The FHS is a community standard that describes the contents and use of the various directories within a Linux root filesystem.

sential libraries. This is yet another artefact that opens the door for making Android coexist with standard Linux components.

There is of course more to be said about Android's root filesystem. The directories just mentioned, for instance, contain their own hierarchies. Also, Android's root filesystem contains other directories that I haven't covered here. We will revist the Android root filesystem and its make-up in more detail in Chapter 5.

Libraries

Android relies on about a hundred dynamically-loaded libraries, all stored in the */system/lib* directory. A certain number of these come from external projects that were merged into Android's codebase to make their functionality available within the Android stack, but a large portion of the libraries in */system/lib* are actually generated from within the AOSP itself. Table 2-2 lists the libraries included in the AOSP that come from external projects, whereas Table 2-3 summarizes the Android-specific libraries generated from within the AOSP.

Table 2-2. Libraries generated from external projects imported into the AOSP

Library(ies)	External Project	Original Location	License
libcrypto.so and *libssl.so*	OpenSSL	*http://www.openssl.org*	Custom, BSD-like
libdbus.so	D-Bus	*http://dbus.freedesktop.org*	AFL and GPL
libexif.so	Exif Jpeg header manipulation tool	*http://www.sentex.net/~mwandel/jhead/*	Public Domain
libexpat.so	Expat XML Parser	*http://expat.sourceforge.net*	MIT
libFFTEm.so	neven face recognition library	N/A	ASL
libicui18n.so and *libicuuc.so*	International Components for Unicode	*http://icu-project.org*	MIT
libiprouteutil.so and *libnetlink.so*	iproute2 TCP/IP networking and traffic control	*http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2*	GPL
libjpeg.so	libjpeg	*http://www.ijg.org*	Custom, BSD-like
libnfc_ndef.so	NXP Semiconductor's NFC library	N/A	ASL
libskia.so and *libskiagl.so*	skia 2D graphics library	*http://code.google.com/p/skia/*	ASL
libsonivox	Sonic Network's Audio Synthesis library	N/A	ASL
libsqlite.so	SQLite database	*http://www.sqlite.org*	Public Domain
libSR_AudioIn.so and *libsrec_jni.so*	Nuance Communications' Speech Recognition engine	N/A	ASL

Library(ies)	External Project	Original Location	License
libstlport.so	Implementation of the C++ Standard Template Library	*http://stlport.sourceforge.net*	Custom, BSD-like
libttspico.so	SVOX's Text-To-Speech speech synthesizer engine	N/A	ASL
libvorbisidec.so	Tremolo ARM-optimized Ogg Vorbis decompression library	*http://wss.co.uk/pinknoise/tremolo/*	Custom, BSD-like
libwebcore.so	WebKit Open Source Project	*http://www.webkit.org*	LGPL and BSD
libwpa_client	Library based on wpa_supplicant	*http://hostap.epitest.fi/wpa_supplicant/*	GPL and BSD
libz.so	zlib compression library	*http://zlib.net*	Custom, BSD-like

Table 2-3. Android-specific libraries generated from within the AOSP

Category	Library(ies)	Description
Bionic	*libc.so*	C library
	libm.so	Math library
	libdl.so	Dynamic linking library
	libstdc++.so	Standard C++ library
	libthread_db.so	Threads library
Core[a]	*libbinder.so*	The Binder library
	libutils.so, *libcutils.so*, *libnetutils.so*, and *libsysutils.so*	Various utility libraries
	libsystem_server.so, *libandroid_servers.so*, *libaudioflinger.so*, *libsurfaceflinger.so*, *linsensorservice.so*, and *libcameraservice.so*	System-services-related libraries
	libcamera_client.so and *libsurfaceflinger_client.so*	Client libraries for certain system services
	libpixelflinger.so	The PixelFlinger library
	libui.so	Low-level user-interface-related functionalities, such as user input events handling and dispatching and graphics buffer allocation and manipulation
	libgui.so	Sensors-related functions library
	liblog.so	The logging library
	libandroid_runtime.so	The Android runtime library
Dalvik	*libdvm.so*	The Dalvik VM library
	libnativehelper.so	JNI-related helper functions
Hardware	*libhardware.so*	The HAL library that provides `hw_get_module()` uses `dlopen()` to load hardware support modules (i.e. shared libraries

Category	Library(ies)	Description
		that provide hardware support to the HAL) on demand.
	libhardware_legacy.so	Library providing hardware support for wifi, power-management and vibrator
	Various hardware-supporting shared libraries.	Libraries that provide support for various hardware components, some of which are loaded using through the HAL, while others are loaded automatically by the linker
Media	*libmediaplayerservice.so*	The Media Player service library
	libmedia.so	The low-level media functions used by the Media Player service
	libstagefright.so*	The many libraries that make-up the StageFright media framework
	libeffects.so and the libraries in the *soundfx/* directory	The sound effects libraries
	libdrm1.so and *libdrm1_jni.so*	The DRM[b] framework libraries
OpenGL	*libEGL.so*, *libETC1.so*, *libGLESv1_CM.so*, *libGLESv2.so*, and *egl/ligGLES_android.so*	Android's OpenGL implementation

[a] I'm using this category as catch-all for many core Android functionalities.

[b] Digital Rights Management

Init

When the kernel finishes booting, it starts just one process, the *init* process. This process is then responsible for spawning all other processes and services in the system and for conducting critical operations such as reboots. The package traditionally provided by Linux distributions for the init process uses SystemV init, although in recent years many distributions have created their own variants. Ubuntu, for instance, uses Upstart. In embedded Linux systems, the classic package that provides init is BusyBox.

Android introduces its own custom init, which brings with it a few novelties.

Configuration language

Unlike traditional inits, which are predicated on the use of scripts that run per the current run-levels' configuration or on request, Android's init defines its own configuration semantics and relies mostly on changes to global properties to trigger the execution of specific instructions.

The main configuration file for init is usually stored as */init.rc*, but there's also usually a device-specific configuration file stored as */init.`device_name`.rc* and device-specific script stored as */etc/init.`device_name`.sh*, where `device_name` is the name of the device. You can get a high degree of control over the system's startup and its behavior by modifying those files. For instance, you can disable the Zygote from starting up auto-

matically and then starting it manually yourself after having used *adb* to shell into the device.

Global properties

A very interesting aspect of Android's init is how it manages a global set of properties that can be accessed and set from many parts of the system, with the appropriate rights. Some of these properties are set at build time, while others are set in init's configuration files and still others are set at runtime. Some properties are also persisted to storage for permanent use. Since init manages the properties, it can detect any changes and therefore trigger the execution of a set of commands based on its configuration.

The OOM adjustments mentioned earlier, for instance, are set on startup by the *init.rc* file. So are network properties. Properties set at build time are stored in the */system/build.prop* file and include the build date and build system details. At runtime, the system will have over a hundred different properties, ranging from IP and GSM configuration parameters to the battery's level. Use the *getprop* command to get the current list of properties and their values.

udev events

As I explained earlier, access to devices in Linux is done through nodes within the */dev* directory. In the older days, Linux distributions would ship with thousands of entries in that directory to accomodate all possible device configurations. Eventually, though, a few schemes were proposed to make the creation of such nodes dynamic. For some time now, the system in use has been *udev*, which relies on runtime events generated by the kernel every time hardware is added or removed from the system.

In most Linux distributions, the handling of udev hotplug events is done by the *udevd* daemon. In Android, these events are handled by the *ueventd* daemon built as part of Android's init and accessed through a symbolic link from */sbin/ueventd* to */init*. To know which entries to create in */dev*, *ueventd* relies on the */ueventd.rc* and */ueventd.device_name.rc* files.

Toolbox

Much like the root filesystem's directory hierarchy, there are essential binaries on most Linux system, listed by the FHS for the */bin* and */sbin* directories. In most Linux distributions, the binaries in those directories are built from separate packages coming from different sites on the net. In an embedded system, it doesn't make sense to have to deal with so many packages, nor necessarily have that many separate binaries.

The approach taken by the classic BusyBox package is to build a single binary that essentially has what amounts to a huge `switch-case`, which checks for the first parameter on the command line and executes the corresponding functionality. All commands are then made to be symbolic links the *busybox* command. So when you type *ls*, for

example, you're actually invoking BusyBox. But since BusyBox's behavior is predicated on the first parameter on the command line and that parameter is *ls*, it will behave as if you had run that command from a standard Linux shell.

Android doesn't use BusyBox, but includes its own tool, Toolbox, that basically functions in the very same way using symbolic links to the *toolbox* command. Unfortunately, Toolbox is nowhere as feature-full as BusyBox. In fact, if you've ever used BusyBox, you're likely going to be very disappointed when using Toolbox. The rationale for creating a tool from scratch in this case seems to make most sense when viewed from the licensing angle, BusyBox being GPL licensed. In addition, some Android developers have stated that their goal was to create a minimal tool for shell-based debugging and not to provide a full replacement for shell tools as BusyBox does. At any rate, Toolbox is BSD licensed and manufacturers can therefore modify it and distribute it without having to track the modifications made by their developers or making any sources available to their customers.

You might still want to include BusyBox alongside Toolbox to benefit from its capabilities. If you don't want to ship it as part of your final product because of its licensing, you could include it temporarily during development and strip it in the final production release. We'll cover this in more detail later.

Daemons

As part of the system startup, Android's init starts a few key daemons that continue to run throughout the lifetime of the system. Some daemon, such as *adbd*, are started on demand, depending on changes to global properties.

Table 2-4. Native Android daemons

Daemon	Description
servicemanager	The Binder Context Manager. Acts as an index of all Binder services running in the system.
vold	The volume manager. Handles the mounting and formatting of mounted volumes and images.
netd	The network manager. Handles tethering, NAT, PPP, PAN, and USB RNDIS.
debuggerd	The debugger daemon. Invoked by Bionic's linker when a process crashes to do a postmortem analysis. Allows *gdb* to connect from the host.
Zygote	The Zygote process. It's responsible for warming up the system's cache and starting the System Server. We'll discuss it in more detail later in this chapter.
mediaserver	The Media server. Hosts most media-related services. We'lll discuss it in more detail later in this chapter.
dbus-daemon	The D-Bus message daemon. Acts as an intermediary between D-Bus users. Have a look at its man page for more information.
bluetoothd	The Bluetooth daemon. Manages Bluetooth devices. Provides services through D-Bus.
installd	The *.apk* installation daemon. Takes care of installing and uninstalling *.apk* files and managing the related filesystem entries.
keystore	The KeyStore daemon. Manages an encrypted key-pair value store for cryptographic keys, SSL certs for instance.

Daemon	Description
system_server	Android's System Server. This daemon hosts the vast majority of system services that run in Android.
adbd	The ADB daemon. Manages all aspects of the connection between the target and the host's *adb* command.

Command-Line Utilities

More than 150 command-line utilities are scattered over Android's root filesystem. */system/bin* contains the majority of them, but some "extras" are in */system/xbin* and a handful are in */sbin*. Around 50 of those in */system/bin* are actually symbolic links to */system/bin/toolbox*. The majority of the rest come from the Android base framework, from external projects merged into the AOSP, or from various other parts of the AOSP. We'll get the chance to cover the various binaries found in the AOSP in more detail in Chapter 5.

Dalvik and Android's Java

In a nutshell, Dalvik is Android's Java virtual machine. It allows Android to run the byte-code generated from Java-based apps and Android's own system components, and provides both with the required hooks and environment to interface with the rest of the system, including native libraries and the rest of the native user-space. There's more to be said about Dalvik and Android's brand of Java, though. But before we can delve into that explanation, we must first cover some Java basics.

Without boring you with yet another history lesson on the Java language and its origins, suffice it to say that Java was created by James Gosling at Sun in the early '90s, that it rapidly became very popular, and that it was, in sum, more than well established before Android came around. From a developer perspective, the are two aspects that are important to keep in mind with regard to Java: its differences with a traditional language such as C and C++, and the components that make up what we commonly refer to as "Java."

By design, Java is an interpreted language. Unlike C and C++, where the code you write gets compiled by a compiler into binary assembly instructions to be executed by a CPU matching the architecture targeted by the compiler, the code that you write in Java gets compiled by a Java compiler into architecture-independent byte-code that is executed at a run-time by a byte-code interpreter, also commonly referred to as a "virtual machine."‡ This modus operandi, along with Java's semantics, enable the language to include quite a few features not traditionally found in previous languages, such as reflection§ and anonymous classes.‖ Also, unlike C and C++, Java doesn't require you to

‡ This term was less ambiguous when Java came out, because "virtual machine" software such as VMWare and VirtualBox weren't as common or as popular as they are today. Such virtual machines do far more than interpret byte-code, as Java virtual machines do.

§ The ability to ask an object whether it implements a certain method.

keep track of objects you allocate. In fact, it requires you to lose track of all unused objects, since it's got an integrated garbage-collector that will ensure all such objects are destroyed when no active code holds a reference to them any longer.

At a practical level, Java is actually made up of a few distinct things: the Java compiler, the Java byte-code interpreter—more commonly known as the Java Virtual Machine (JVM)—and the Java libraries commonly used by Java developers. Together, these are usually obtained by developers through the Java Development Kit (JDK) provided free of charge by Oracle. Android actually relies on the JDK for the Java compiler, but it doesn't use the JVM or the libraries found in the JDK. Instead of the JVM, it relies on Dalvik, and instead of the JDK libraries, it relies on the Apache Harmony project, a clean-room implementation of the Java libraries hosted under the umbrella of the Apache project.

According to its developer, Dan Bornstein, Dalvik distinguishes itself from the JVM by being specifically designed for embedded systems. Namely, it targets systems that have slow CPUs and relatively litte RAM, run OSes that don't use swap space, and are battery powered.

While the JVM munches on *.class* files, Dalvik prefers the *.dex* delicatessen. *.dex* files are actually generated by postprocessing the *.class* files generated by the Java compiler through Android's *dx* utility. Among other things, an uncompressed *.dex* file is 50% smaller than its originating *.jar*# file. Another interesting factoid is that Dalvik is register-based whereas the JVM is stack-based, though that is likely to have little to no meaning to you unless you're an avid student of VM theory, architecture, and internals.

If you'd like to get more information about the features and internals of Dalvik, I strongly encourage you to take a look at Dan Bornstein's Google I/O 2008 presentation entitled "Dalvik Virtual Machine Internals." It's about one hour long and available on YouTube. You can also just go to YouTube and search for "Dan Bornstein Dalvik."

If you'd like to get the inside track on the benefits and tradeoffs between stack-based VMs versus register-based VMs, have a look at the paper entitled "Virtual Machine Showdown: Stack Versus Registers" by Shi et al. in proceedings of VEE'05, June 11-12, 2005, Chicago, IL, p. 153-163.

A feature of Davlik very much worth highlighting, though, is that since 2010 it has included a Just-In-Time (JIT) compiler for ARM. This means that Dalvik converts apps' byte-codes to binary assembly instructions that run natively on the target's CPU instead of being interpreted one instruction at a time by the VM. The result of this conversion

‖ Snippets of code that are passed as a parameter to a method being invoked. An anonymous class might be used, for instance, as a callback registration method, thereby enabling the developer to visualize the code handling an event at the same location they invoke the callback registration method.

#*.jar* files are actually Java ARchives (JAR) containing many *.class* files, each of which contain only a single class.

is then stored for future use. Hence, apps take longer to load the first time, but once they've been JIT'ed, they load and run much faster. The only caveat here is that JIT isn't available for any other architecture than ARM. So, in sum, the fastest architecture to run Android on is, for now, ARM.

As an embedded developer, you're unlikely to need to do anything specific to get Dalvik to work on your system. Dalvik was written to be architecture-independent. It has been reported that some of the early ports of Dalvik suffered from some endian issues. However, these issues seem to have subsided since.

Java Native Interface (JNI)

Despite its power and benefits, Java can't always operate in a vacuum, and code written in Java sometimes needs to interface to code coming from other languages. This is especially true in an embedded environment such as Android, where low-level functionality is never too far away. To that end, the Java Native Interface (JNI) mechanism is provided. It's essentially a call gate to other languages such as C and C++. It's an equivalent to *pinvoke* in the .NET/C# world.

App developers sometimes use JNI to call the native code they compile with the NDK from their regular Java code built using the SDK. Interally, though, the AOSP massively relies on JNI to enable Java-coded services and components to interface with Android's low-level functionality, which is mostly written in C and C++. Java-written system services, for instance, very often use JNI to communicate with matching native code that interfaces with a given service's corresponding hardware.

A large part of the heavy lifting to allow Java to communicate with other languages through JNI is actually done by Dalvik. If you go back to Table 2-3 in the previous section, for instance, you'll notice the *libnativehelper.so* library, which is provided as part of Dalvik for facilitating JNI calls.

In later parts of the book, we'll actually get the chance to use JNI to interface Java and C code. For the moment being, keep in mind that JNI is central to platform work in Android and that it can be a relatively complex mechanism to use, especially to make sure you use the appropriate call semantics and function parameters.

Unfortunately, JNI seems to be a dark art reserved to the initiated. In other words, it's rather difficult to find good documentation on the topic. There is one authorative book on the topic, *The Java™ Native Interface Programmer's Guide and Specification* by Sheng Liang (Addison-Wesley). You can purchase a copy from your favorite online bookstore, but it's also freely available for download as a PDF. Given how precious this document is, I suggest you grab a copy in earnest for posterity, just in case it spontaneously evaporates from the net for one reason or another.

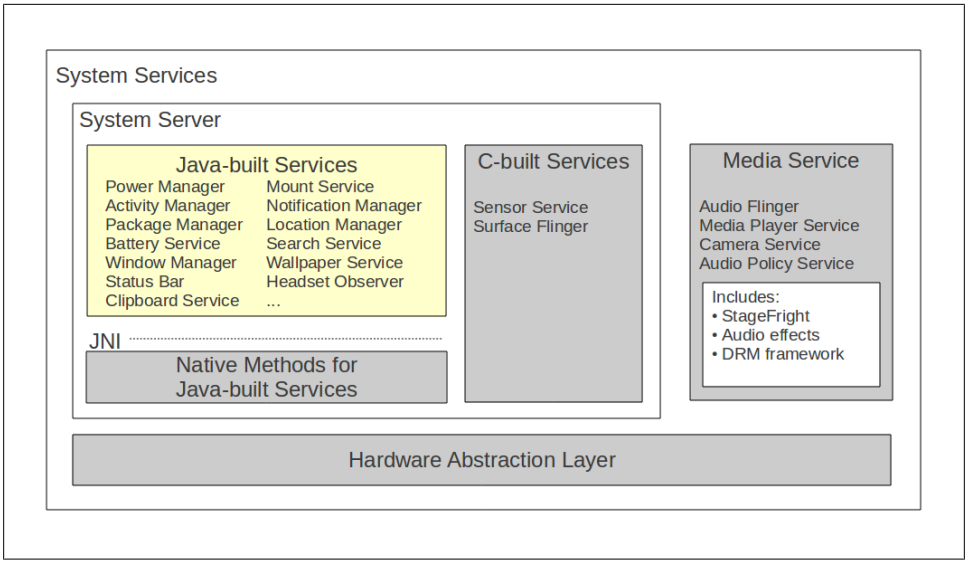


Figure 2-4. System Services

System Services

System services are Android's man behind the curtain. Even if they aren't explicitly mentioned in Google's app development documentation, anything remotely interesting in Android goes through one of about 50 system services. These services cooperate together to collectively provide what essentially amounts to an object-oriented OS built on top of Linux, which is exactly what Binder— the mechanism on which all system services are built—was intended for. The native user-space we just covered is actually designed very much as a support environment for Android's system services. It's therefore crucial to understand what system services exist, and how they interact with each other and with the rest of the system. We've already covered some of this as part of discussing Android's hardware support.

Figure 2-4 illustrates in greater detail the system services first introduced in Figure 2-1. As you can see, there are in fact a couple of major processes involved. Most prominent is the System Server, whose components all run under the sames process, *system_server*, and which is mostly made up of Java-coded services with two services written in C/C++. The System Server also includes some native code access through JNI to allow some of the Java-based services to interface to Android's lower layers. The rest of the system services are housed within the Media Service which runs as *mediaserver*. These services are all coded in C/C++ and are packaged alongside media-related components such as the StageFright and audio effects.

Note that despite there being only two processes to house the entirety of the Android's system services, they all appear to operate independently to anyone connecting to their

services through Binder. Here's the output of the *service* utility on the Android emulator:

```
# service list
Found 50 services:
0    phone: [com.android.internal.telephony.ITelephony]
1    iphonesubinfo: [com.android.internal.telephony.IPhoneSubInfo]
2    simphonebook: [com.android.internal.telephony.IIccPhoneBook]
3    isms: [com.android.internal.telephony.ISms]
4    diskstats: []
5    appwidget: [com.android.internal.appwidget.IAppWidgetService]
6    backup: [android.app.backup.IBackupManager]
7    uimode: [android.app.IUiModeManager]
8    usb: [android.hardware.usb.IUsbManager]
9    audio: [android.media.IAudioService]
10    wallpaper: [android.app.IWallpaperManager]
11    dropbox: [com.android.internal.os.IDropBoxManagerService]
12    search: [android.app.ISearchManager]
13    location: [android.location.ILocationManager]
14    devicestoragemonitor: []
15    notification: [android.app.INotificationManager]
16    mount: [IMountService]
17    accessibility: [android.view.accessibility.IAccessibilityManager]
18    throttle: [android.net.IThrottleManager]
19    connectivity: [android.net.IConnectivityManager]
20    wifi: [android.net.wifi.IWifiManager]
21    network_management: [android.os.INetworkManagementService]
22    netstat: [android.os.INetStatService]
23    input_method: [com.android.internal.view.IInputMethodManager]
24    clipboard: [android.text.IClipboard]
25    statusbar: [com.android.internal.statusbar.IStatusBarService]
26    device_policy: [android.app.admin.IDevicePolicyManager]
27    window: [android.view.IWindowManager]
28    alarm: [android.app.IAlarmManager]
29    vibrator: [android.os.IVibratorService]
30    hardware: [android.os.IHardwareService]
31    battery: []
32    content: [android.content.IContentService]
33    account: [android.accounts.IAccountManager]
34    permission: [android.os.IPermissionController]
35    cpuinfo: []
36    meminfo: []
37    activity: [android.app.IActivityManager]
38    package: [android.content.pm.IPackageManager]
39    telephony.registry: [com.android.internal.telephony.ITelephonyRegistry]
40    usagestats: [com.android.internal.app.IUsageStats]
41    batteryinfo: [com.android.internal.app.IBatteryStats]
42    power: [android.os.IPowerManager]
43    entropy: []
44    sensorservice: [android.gui.SensorServer]
45    SurfaceFlinger: [android.ui.ISurfaceComposer]
46    media.audio_policy: [android.media.IAudioPolicyService]
47    media.camera: [android.hardware.ICameraService]
48    media.player: [android.media.IMediaPlayerService]
```

```
49    media.audio_flinger: [android.media.IAudioFlinger]
```

There is unfortunately not much documentation on how each of these services operates. You'll have to look at each service's source code to get a precise idea of how it works and how it interacts with other services.

Reverse Engineering Source Code

Fully understanding the internals of Android's system services is like trying to swallow a whale. There are about 85k lines of Java code in the System Server alone, spread across 100 different files. And that doesn't count any system service code written in C/C++. To add insult to injury, so to speak, the comments are few and far between and the design documents non-existent. Arm yourself with a good dose of patience if you want to dig further here.

One trick is to create a new Java project in Eclipse and import the System Server's code inside that project. This won't compile in any way, but it'll allow you to benefit from Eclipse's Java browsing capabilities to help in trying to understand the code. For instance, you can open a single Java file, right-click on the source browsing scrollbar area, and select Folding → Collapse All. This will essentially collapse all methods into a single line next to a plus sign (+) and will allow you to see the trees (the method names lined-up one after another) instead of the leaves (the actual content of each method.) You'll very much still be in a forest, though.

You can also try using one of the commercial source code analysis tools on the market from vendors such as Imagix, Rationale, Lattix, or Scitools. Although there are some open source analysis tools out there, most seem geared towards locating bugs, not reverse-engineering the code being analyzed.

Service Manager and Binder Interaction

As I explained earlier, the Binder mechanism used as system services' underlying fabric enables object-oriented remote method invocation. For a process in the system to invoke a system service through Binder, though, it must first have a handle to it. For instance, Binder will enable an app developer to request a wakelock from the Power Manager by invoking the `acquire()` method of its `WakeLock` nested class. Before that call can be made, though, the developer must first get a handle to the Power Manager service. As we'll see in the next section, the app development API actually hides the details of how it gets this handle in an abstraction to the developer, but under the hood all system service handle lookups are done through the Service Manager, as illustrated in Figure 2-5.

Think of the Service Manager as a YellowPages book of all services available in the system. If a system service isn't registered with the Service Manager, it's effectively invisible to the rest of the system. To provide this indexing capability, the Service Manager is started by *init* before any other service. It then opens */dev/binder* and uses a

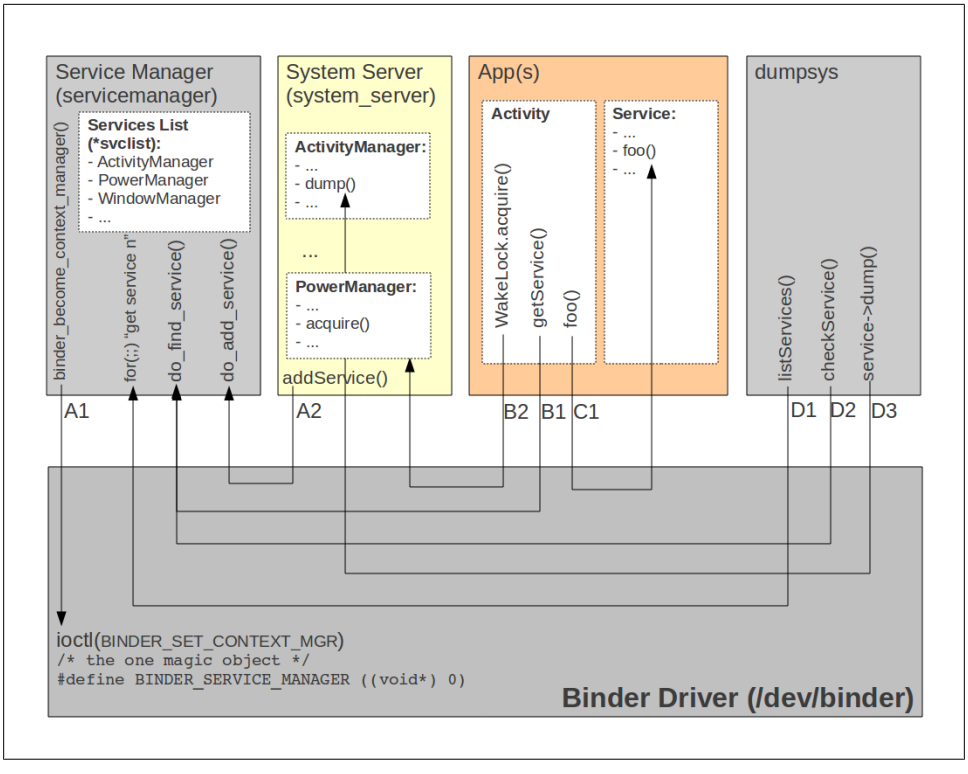


Figure 2-5. Service Manager and Binder interaction

special `ioctl()` call to set itself as the Binder's *Context Manager* ("A1" in Figure 2-5.) Thereafter, any process in the system that attempts to communicate with Binder ID 0 (a.k.a. the "magic" Binder or "magic object" in various parts of the code), is actually communicating through Binder to the Service Manager.

When the System Server starts, for instance, it registers every single service it instantiates with the Service Manager ("A2".) Later, when an app tries to talk to a system service, such as the Power Manager service, it first asks the Service Manager for a handle to the service ("B1") and then invokes that service's methods ("B2"). In contrast, a call to a service component running within an app goes directly through Binder ("C1"), and is noy looked up through the Service Manager.

The Service Manager is also used in a special way by the *dumpsys* utility, which allows you to dump the status of a single or all system services. To get the list of all services, it loops around to get every system service ("D1"), requesting the n^{th} plus one at every iteration until there aren't any more. To get each service, it just asks the Service Manager to locate that specific one ("D2".) With a service handle in hand, it invokes that service's `dump()` function to dump its status ("D3") and displays that on the terminal.

Calling on Services

All of what I just explained is, as I said earlier, almost invisible to the user. Here's a snippet, for instance, that allows us to grab a wakelock within an app using the regular application development API:

```
PowerManager pm = (PowerManager) getSystemService(POWER_SERVICE);
PowerManager.WakeLock wakeLock = pm.newWakeLock(PowerManager.FULL_WAKE_LOCK, "myPreciousWakeLock");
wakeLock.acquire(100);
```

Notice that we don't see any hint of the Service Manager here. Instead, we're using `getSystemService()` and passing it the `POWER_SERVICE` parameter. Internally, though, the code that implements `getSystemService()` does actually use the Service Manager to locate the Power Manager service so that we create a wakelock and acquire it.

A Service Example: the Activity Manager

Although covering each and every system service is outside the scope of this book, let's have a quick look at the Activity Manager, one of the key system services. The Activity Manager's sources actually span over 30 files and 20k lines of code. If there's a core to Android's internals, this service is very much near it. It takes care of the starting of new components, such as Activities and Services, along with the fetching of Content Providers and intent broadcasting. If you ever got the dreaded ANR (Application Not Responding) dialog box, know that the Activity Manager was behind it. It's also involved in the maintenance of OOM adjustments used by the in-kernel low-memory handler, permissions, task management, etc.

For instance, when the user clicks on a icon to start an app from his home screen, the first that happens is that the Launcher's* `onClick()` callback is called. To deal with the event, the Launcher will then call, through Binder, the `startActivity()` method of the Activity Manager service. The service will then call the `startViaZygote()` method, which will open a socket to the Zygote and ask it to start the Activity. All this may make more sense after you read the final section of this chapter.

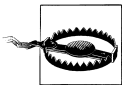
If you're familiar with Linux's internals, a good way to think of the Activity Manager is that it's to Android what the content of the *kernel/* directory in the kernel's sources is to Linux. It's that important.

Stock AOSP Packages

The AOSP ships with a certain number of default packages that are found in most Android devices. As I mentioned in the previous chapter, though, some apps such as

* The Launcher is the default app packaged with the AOSP that takes care of the main interface with the user, the home screen.

Maps, YouTube, and Gmail aren't part of the AOSP. Let's take a look at some of those packages included by default. Table 2-5 lists the stock apps included in the AOSP, Table 2-6 lists the stock content providers included in the AOSP, and Table 2-7 lists the stock IMEs (Input Method Editors) included in the AOSP.



While these are coded very much like standard apps, most won't build outside the AOSP using the standard SDK. Hence, if you'd like to create your own version of one of these apps (i.e., fork it), you'll either have to do it inside the AOSP or invest some time in getting the app to build outside the AOSP with the standard SDK. For one thing, these apps sometimes use APIs that are accessible within the AOSP but aren't exported through the standard SDK.

Table 2-5. Stock AOSP Apps

App in AOSP	Name displayed in Launcher	Description
AccountsAndSettings	N/A	Accounts management app
Bluetooth	N/A	Bluetooth manager
Browser	Browser	Default Android browser, includes bookmark widget
Calculator	Calculator	Calculator app
Camera	Camera	Camera app
CertInstaller	N/A	UI for installing certificates
Contacts	Contacts	Contacts manager app
DeskClock	Clock	Clock and alarm app, including the clock widget
DownloadsUI	Downloads	UI for DownloadProvider
Email	Email	Default Android email app
Development	Dev Tools	Miscellaneous dev tools
Gallery	Gallery	Default gallery app for viewing pictures
Gallery3D	Gallery	Fancy gallery with "sexier" UI
HTMLViewer	N/A	App for viewing HTML files
Launcher2	N/A	Default home screen
Mms	Messaging	SMS/MMS app
Music	Music	Music player
PackageInstaller	N/A	App install/uninstall UI
Phone	Phone	Default phone dialer/UI
Protips	N/A	Home screen tips
Provision	N/A	App for setting a flag indicating whether a device was provisioned
QuickSearchBox	Search	Search app and widget
Settings	Settings	Settings app, also accessible through home screen menu

App in AOSP	Name displayed in Launcher	Description
SoundRecorder	N/A	Sound recording app[a]
SpeechRecorder	Speech Recorder	Speech recording app
SystemUI	N/A	Status bar

[a] This one is activated when a recording intent is sent. It can't be accessed directly by the user.

Table 2-6. Stock AOSP Providers

Provider	Description
ApplicationProvider	Provider for search installed apps
CalendarProvider	Main Android calendar storage and provider
ContactsProvider	Main Android contacts storage and provider
DownloadProvider[a]	Download management, storage and access
DrmProvider	Management and access of DRM-protected storage
MediaProvider	Media storage and provider
TelephonyProvider	Carrier and SMS/MMS storage and provider
UserDictionnaryProvider	Storage and provider for user-defined words dictionary

[a] Interestingly, this package's source code includes a design document, a rarety in the AOSP.

Table 2-7. Stock AOSP Input Methods

Input Method	Description
LatinIME	Latin keyboard
OpenWnn	Japanese keyboard
PinyinIME	Chinese keyboard

System Startup

The best way to bring together all that we discussed is to look at Android's startup. As you can see in Figure 2-6, the first cog to turn is the CPU. It typically has a hard-coded address from which it fetches its first instructions. That address usually points to a chip that has the bootloader programmed on it. The bootloader then initializes the RAM, puts basic hardware in a quiescent state, loads the kernel and RAM disk, and jumps into the kernel. More recent System-on-Chip (SoC) devices, which include a CPU and a slew of peripherials in a single chip, can actually boot straight from a properly formatted SD card or SD-card-like chip. The PandaBoard and recent editions of the BeagleBoard, for instance, don't have any on-board flash chips because they boot straight from an SD card.

The initial kernel startup is very hardware dependent, but its purpose is to set things up so that the CPU can start executing C code as early as possible. Once that's done, the kernel jumps to the architecture-independent `start_kernel()` function, initializes

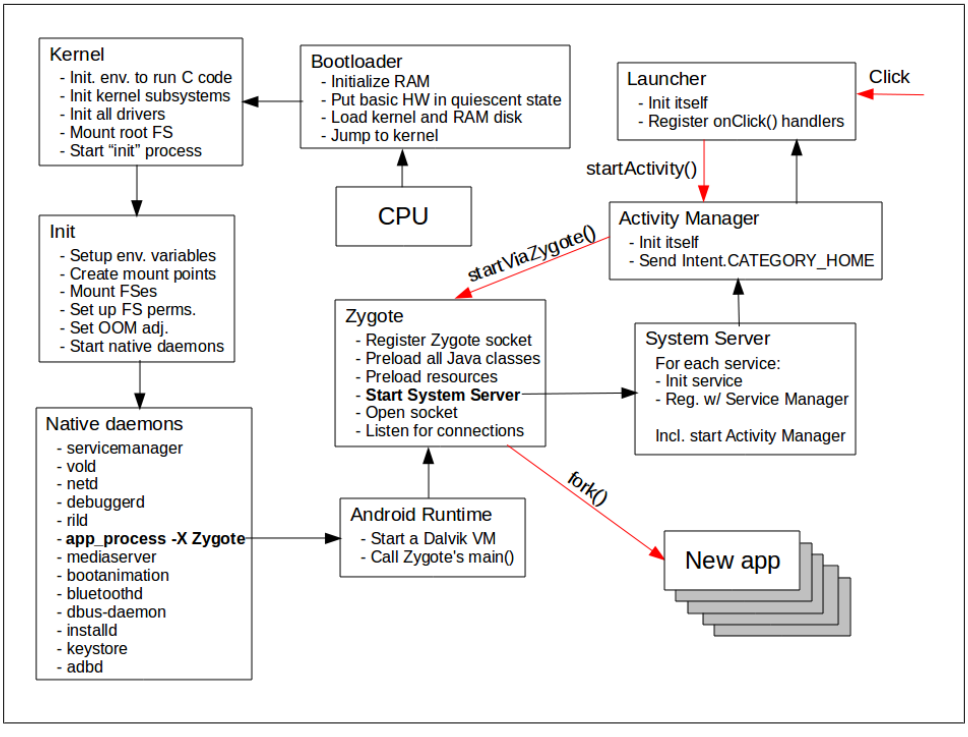


Figure 2-6. Android's boot sequence

its various subsystems, and proceed to call the "init" functions of all built-in drivers. The majority of messages printed out by the kernel at startup come from these steps. The kernel then mounts its root filesystem and starts the init process.

That's when Android's init kicks in and executes the instructions stored in its */init.rc* file to set up environment variables such as the system path, create mount points, mount filesystems, set OOM adjustments, and start native daemons. We've already covered the various native daemons active in Android, but it's worth focusing a little on the Zygote. The Zygote is a special daemon whose job is to launch apps. Its functionality is centralized here in order to unify the components shared by all apps and to shorten their start-up time. init doesn't actually start the Zygote directly; instead it uses the *app_process* command to get Zygote started by the Android runtime. The runtime then starts the first Dalvik VM of the system and tells it to invoke the Zygote's `main()`.

Zygote is active only when a new app needs to be launched. To achieve a speedier app launch, the Zygote starts by preloading all Java classes and resources that an app may potentially need at runtime. This effectively loads those into the system's RAM. The Zygote then listens for connections on its socket (*/dev/socket/zygote*) for requests to start new apps. When it gets a request to start an app, it forks itself and launches the new app. The beauty of having all apps fork from the Zygote is that it's a "virgin" VM that has all the system classes and resources an app may need preloaded and ready to

be used. In other words, new apps don't have to wait until those are loaded to start executing.

All of this works because the Linux kernel implements a Copy-On-Write (COW) policy for forks. As you may know, forking in Unix involves creating a new process that is an exact same copy of the parent process. With COW, Linux doesn't actually copy anything. Instead, it maps the pages of the new process over to those of the parent process and makes copies only when the new process writes to a page. But in fact the classes and resources loaded are never written to, because they're the default ones and are pretty much immutable within the lifetime of the system. So all processes directly forking from the Zygote are essentially using its own mapped copies. And therefore, regardless of the number of apps running on the system, only one copy of the system classes and the resources is ever loaded in RAM.

Although the Zygote is designed to listen to connections for requests for forking new apps, there is one "app" that the Zygote actually starts explicitly: the System Server. This is the first app started by the Zygote and it continues to live on as an entirely separate process from its parent. The System Server then starts initializing each system service it houses and registering it with the previously-started Service Manager. One of the services it starts, the Activity Manager, will end its initialization by sending an intent of type `Intent.CATEGORY_HOME`. This starts the Launcher app, which then displays the home screen familiar to all Android users.

When the user clicks on an icon on the home screen, the process I described in "System Services" on page 53 takes place. The Launcher asks the Activity Manager to start the process, which in turn "forwards" that request on to the Zygote, which itself forks and starts the new app, which is then displayed to the user.

Once the system has finished starting up, the process list will look something like this:

```
# ps
USER     PID   PPID  VSIZE  RSS     WCHAN    PC         NAME
root      1     0     268    180   c009b74c 0000875c S /init
root      2     0     0      0     c004e72c 00000000 S kthreadd
root      3     2     0      0     c003fdc8 00000000 S ksoftirqd/0
root      4     2     0      0     c004b2c4 00000000 S events/0
root      5     2     0      0     c004b2c4 00000000 S khelper
root      6     2     0      0     c004b2c4 00000000 S suspend
root      7     2     0      0     c004b2c4 00000000 S kblockd/0
root      8     2     0      0     c004b2c4 00000000 S cqueue
root      9     2     0      0     c018179c 00000000 S kseriod
root      10    2     0      0     c004b2c4 00000000 S kmmcd
root      11    2     0      0     c006fc74 00000000 S pdflush
root      12    2     0      0     c006fc74 00000000 S pdflush
root      13    2     0      0     c0079750 00000000 D kswapd0
root      14    2     0      0     c004b2c4 00000000 S aio/0
root      22    2     0      0     c017ef48 00000000 S mtdblockd
root      23    2     0      0     c004b2c4 00000000 S kstriped
root      24    2     0      0     c004b2c4 00000000 S hid_compat
root      25    2     0      0     c004b2c4 00000000 S rpciod/0
```

```
root      26    1     232   136   c009b74c 0000875c S /sbin/ueventd
system    27    1     804   216   c01a94a4 afd0b6fc S /system/bin/servicemanager
root      28    1     3864  308   ffffffff afd0bdac S /system/bin/vold
root      29    1     3836  304   ffffffff afd0bdac S /system/bin/netd
root      30    1     664   192   c01b52b4 afd0c0cc S /system/bin/debuggerd
radio     31    1     5396  440   ffffffff afd0bdac S /system/bin/rild
root      32    1     60832 16348 c009b74c afd0b844 S zygote
media     33    1     17976 1104  ffffffff afd0b6fc S /system/bin/mediaserver
bluetooth 34    1     1256  280   c009b74c afd0c59c S /system/bin/dbus-daemon
root      35    1     812   232   c02181f4 afd0b45c S /system/bin/installd
keystore  36    1     1744  212   c01b52b4 afd0c0cc S /system/bin/keystore
root      38    1     824   272   c00b8fec afd0c51c S /system/bin/qemud
shell     40    1     732   204   c0158eb0 afd0b45c S /system/bin/sh
root      41    1     3368  172   ffffffff 00008294 S /sbin/adbd
system    65    32    123128 25232 ffffffff afd0b6fc S system_server
app_15    115   32    77232 17576 ffffffff afd0c51c S com.android.inputmethod.latin
radio     120   32    86060 17952 ffffffff afd0c51c S com.android.phone
system    122   32    73160 17656 ffffffff afd0c51c S com.android.systemui
app_27    125   32    80664 22900 ffffffff afd0c51c S com.android.launcher
app_5     173   32    74404 18024 ffffffff afd0c51c S android.process.acore
app_2     212   32    73112 17032 ffffffff afd0c51c S android.process.media
app_19    284   32    70336 16672 ffffffff afd0c51c S com.android.bluetooth
app_22    292   32    72752 17844 ffffffff afd0c51c S com.android.email
app_23    320   32    70276 15792 ffffffff afd0c51c S com.android.music
app_28    328   32    70744 16444 ffffffff afd0c51c S com.android.quicksearchbox
app_14    345   32    69708 15404 ffffffff afd0c51c S com.android.protips
app_21    354   32    70912 17152 ffffffff afd0c51c S com.cooliris.media
root      366   41    2128  292   c003da38 00110c84 S /bin/sh
root      367   366   888   324   00000000 afd0b45c R /system/bin/ps
```

This output actually comes from the Android emulator, so it contains some emulator-specific artefacts such as the *qemud* daemon. Notice that the apps running all bare their fully-qualified package names despite being forked from the Zygote. This is a neat trick that can be pulled in Linux by using the `prctl()` system call with `PR_SET_NAME` to tell the kernel to change the calling process' name. Have a look at `prctl()`'s man page if you're interested in it. Note also that the first process started by init is actually *ueventd*. All processes prior to that are actually started from within the kernel by subsystems or drivers.

CHAPTER 3
AOSP Jumpstart

Now that you have a solid understanding of the basics, let's start getting our hands dirty with the AOSP. We'll start by covering how to get the AOSP *repo* from *http://android.git.kernel.org/*. Before actually building and running the AOSP, we'll spend some time exploring the AOSP's contents and explain how the sources reflect what we just saw in the previous chapter. Finally, we'll close the chapter by covering the use of *adb* and the emulator, two very important tools when doing any sort of platform work.

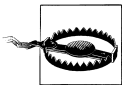
Above all, this chapter is meant to be fun. The AOSP is an exciting piece of software with a tremendous amount of innovations. Ok, ok, I'll admit it's not all rosy and some parts do have rough edges. Still, some other parts are pure genius. The most amazing thing of all obviously is that we can all download it, modify it, and ship our own custom products based on it. So roll up your sleeves and let's get started.

Getting the AOSP

As I had mentioned earlier, the official AOSP is available at *http://android.git.kernel.org*, which sports a Gitweb interface.* When you visit the site, you will see a fairly large number of git repositories you can pull from that location. Needless to say, pulling each and every one of these manually would be rather tedious; there are over a hundred. And, in fact, pulling them all would be quite useless because only a subset of these projects is needed. The right way to pull the AOSP is to use the *repo* tool which is available at the very same location. First, though, you'll need to get *repo* itself:

```
$ sudo apt-get install curl
$ curl https://android.git.kernel.org/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
```

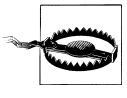
* The web interface for the *git* tool.



Under Ubuntu, *~/bin* is automatically added to your path when you log in, *if it already exists*. So, if you don't have a *bin/* directory in your home directory, create it, then log out and log back in to make it part of your path. Otherwise, the shell won't be able to find *repo*, even if you fetch it as I just showed.

You don't have to put *repo* in *~/bin*, but it has to be in your path. So regardless of where you put it, just make sure it's available to you in all locations in the filesystem from the command line.

Despite its structure as a single shell script, *repo* is actually quite an intricate tool and we'll take a deeper look at it later. For now, though, consider *repo* as a tool that can simultaneously pull from multiple *git* repositories to create an Android distribution. The repositories it pulls from are given to it through a *manifest* file, which is an XML file describing the projects that need to pulled from and their location. *repo* is in fact layered on top of *git* and each project it pulls from is an independent *git* repository.



Confusing as it may be, note that *repo*'s "manifest" file has absolutely **nothing** to do with "manifest" files (*AndroidManifest.xml*) used by app developers to describe their apps to the system. Their formats and uses are completely different. Fortunately, they rarely have to be used within the same context, so while you should keep this fact in mind we won't need to worry too much about it in the coming explanations.

Now that we've got *repo*, let's get ourselves a copy of the AOSP:

```
$ mkdir -p ~/android/aosp-2.3.x
$ cd ~/android/aosp-2.3.x
$ repo init -u git://android.git.kernel.org/platform/manifest.git -b gingerbread
$ repo sync
```

The last command should run for quite some time as it goes and fetches the sources of all the projects described in the manifest file. After all, the AOSP is about 4GB in size uncompiled. Keep in mind therefore that network bandwidth and latencies will play a big role in how long this takes. Note also that we are fetching a specific branch of the tree, Gingerbread. That's the `-b gingerbread` part of the third command. If you omit that part, you will be getting the *master* branch. It's been the experience of many people that the master branch doesn't always build or run properly, because it contains the tip of the open development branch. Tagged branches, on the other hand, mostly work out of the box.

Inside the AOSP

Now that we've got a copy of the AOSP, let's start looking at what's inside and, most importantly, connect that to what we just saw in the previous chapter. Feel free to skip over this section and come back to it after the next section if you're too eager to get your own custom Android running. For those of you still reading, have a look at Table 3-1 for a summary of the AOSP's top-level directory.

Table 3-1. AOSP content summary

Directory	Content	Size (in MB)
bionic	Android's custom C library	14
bootable	Reference bootloader and recovery mechanism	4
build	Build system	4
cts	Comptability Test Suite	78
dalvik	Dalvik VM	35
development	Development tools	65
device	Device-specific files and components	17
external	Copy of external projects used within AOSP	854
frameworks	Core components such as system services	361
hardware	HAL and hardware support libraries	27
libcore	Apache Harmony	54
ndk	Native Development Kit	13
packages	Stock Android apps, providers and IMEs	117
prebuilt	Prebuilt binaries, including toolchains	1,389
sdk	Software Development Kit	14
system	"Embedded Linux" platform that houses Android	32

As you can see, *prebuilt* and *external* are the two largest directories in the tree, accounting for close to 75% of its size. Interestingly, both these directories are mostly made up of content from other open source projects and include things like various GNU toolchain versions, kernel images, common libraries and frameworks such as OpenSSL and WebKit, etc. *libcore* is also from another open source project, Apache Harmony. In essence, this is further evidence of how much Android relies heavily on the rest of the open source ecosystem to exist. Still, Android contains a fair bit of "original" (or near to) code; about 800MB of it in fact.

To best understand Android's sources, it's useful to refer back to Figure 2-1, which illustrated Android's architecture in the previous chapter. Figure 3-1 is a variant of that figure that illustrates the location of each Android component in the AOSP sources. Obviously, a lot of key components come from *frameworks/base/*, which is where the

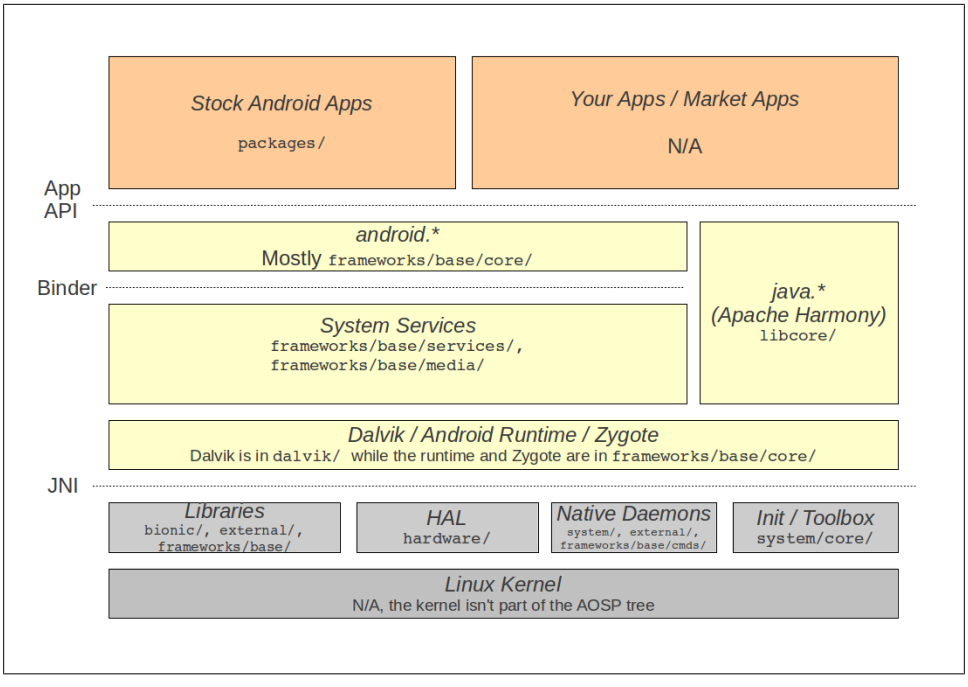


Figure 3-1. Android's architecture

bulk of Android's "brains" are located. It's in fact worth taking a closer look at that directory and *system/core/*, in Table 3-2 and Table 3-3 respectively, as they contain a large chunk of the moving parts you'll likely be interested in interfacing with or modifying as an embedded developer.

Table 3-2. Content summary for frameworks/base/

Directory	Content
cmds	Native commands and daemons
core	The android.* packages
data	Fonts and sounds
graphics	2D graphics and Renderscript
include	C-language include files
keystore	Security key store
libs	C libraries
location	Location provider
media	Media Service, StageFright, codecs, etc.
native	Native code for some framework components
obex	Bluetooth Obex

Directory	Content
opengl	OpenGL library and Java code
packages	A few core packages such as the Status Bar
services	System services
telephony	Telephony API, which talks to the *rild* radio layer interface
tools	A few core tools such as *aapt* and *aidl*
voip	RTP and SIP APIs
vpn	VPN Manager
wifi	Wifi Manager and API

Table 3-3. Content summary for system/core/[a]

Directory	Content
adb	The ADB daemon and client
cpio	*mkbootfs* tool used to generate RAM disk images[b]
debuggerd	*debuggerd* command covered in Chapter 2
fastboot	*fastboot* utility used to communicate with Android bootloaders using the "fastboot" protocol
include	C-language headers for all things "system"
init	Android's *init*
libacc	"Almost" C Compiler library for compiling C-like code; used by RenderScript[c]
libcutils	Various C utility functions not part of the standard C library; used throughout the tree
libdiskconfig	For reading and configuring disks; used by *vold*
liblinenoise	BSD-licensed `readline()` replacement from *http://github.com/antirez/linenoise*; used by Android's shell
liblog	Logting library that interfaces with the Android kernel logger as seen in Figure 2-2; used throughout the tree
libmincrypt	Basic RSA and SHA functions; used by the recovery mechanism and *mkbootimg* utility
libnetutils	Network configuration library; used by *netd*
libpixelflinger	Low-level graphic rendering functions
libsysutils	Utility functions for talking with various components of the system, including the framework; used by *netd* and *vold*
libzipfile	Wrapper around zlib for dealing with zip files
logcat	The *logcat* utility
logwrapper	Utility that forks and runs the command passed to it while redirecting stdout and stderr to Android's logger
mkbootimg	Utility for creating a boot image using a RAM disk and a kernel
netcfg	Network configuration utility
rootdir	Default Android root directory structure and content
run-as	Utility for running a program as a given user ID
sh	Android shell

Directory	Content
toolbox	Android's Toolbox (BusyBox replacement)

[a] Some entries have been omitted because they aren't currently used by any part of the AOSP. They are likely legacy components.

[b] This is used to create both the default RAM disk image used to boot Android and the recovery image.

[c] This description might not make any sense to you unless you know what RenderScript is. Have a look at Google's documentation for RenderScript, the relevance of *libacc* in that context should be clearer.

In addition to *base/*, *frameworks/* contains a few other directories, but they are nowhere near as fundamental as *base/*. Likewise, in addition to *core/*, *system/* also includes a few more directories such as *netd/* and *vold/*, which contain the *netd* and *vold* daemons respectively.

In addition to the top-level directories, the root directory also includes a single Makefile. That file is however mostly empty, its main use being to include the entry point to Android's build system:

```
### DO NOT EDIT THIS FILE ###
include build/core/main.mk
### DO NOT EDIT THIS FILE ###
```

As you've likely figured already, there's far more to the AOSP than what I just presented to you. There are, after all, more than 14,000 directories and 100,000 files in 2.3.x/Gingerbread. By most standards, it's a fairly large project. In comparison, early 3.0.x releases of the Linux kernel have about 2,000 directories and 35,000 files. We will certainly get the chance to explore more parts of the AOSP's sources as we move forward. I highly recommend, though, you start exploring and experimenting with the sources in earnest as it will likely take you several months before you can comfortably navigate your way through.

Build Basics

So now we have an AOSP, and a general idea of what's inside, so let's get it up and running. There's one last thing we need to do before we can build it, though. We need to make sure we've got all the packages necessary on our Ubuntu install. Here are the instructions based on Ubuntu 11.04. Even it you are using a older or newer version of some Debian-based Linux distribution, the instructions will be fairly similar. (See also "Building on Virtual Machines or Non-Ubuntu Systems" on page 72 for other systems on which you can build the AOSP.)

Build System Setup

First, let's get some of the basic packages installed on our development system. You might have some of these already installed as part of other development work you've

been doing and that's fine. Ubuntu's package management system will ignore your request to install those packages.

```
$ sudo apt-get install bison flex gperf git-core gnupg zip tofrodos \
> build-essential g++-multilib libc6-dev libc6-dev-i386 ia32-libs mingw32 \
> zlib1g-dev lib32z1-dev x11proto-core-dev libx11-dev \
> lib32readline5-dev libgl1-mesa-dev lib32ncurses5-dev
```

You might also need to fix a few symbolic links:

```
$ sudo ln -s /usr/lib32/libstdc++.so.6 /usr/lib32/libstdc++.so
$ sudo ln -s /usr/lib32/libz.so.1 /usr/lib32/libz.so
```

Finally, you need to install Sun's JDK:†

```
$ sudo add-apt-repository "deb http://archive.canonical.com/ natty partner"
$ sudo apt-get update
$ sudo apt-get install sun-java6-jdk
```

Your system is now ready to build Android. Obviously you don't need to do this package installation process every time you build Android. You'll need to do it only once for every Android development system you set up.

Building Android

We are now ready to build Android. Let's go to the directory where we downloaded Android and configure the build system:

```
$ cd ~/android/aosp-2.3.x
$ . build/envsetup.sh
$ lunch

You're building on Linux

Lunch menu... pick a combo:
     1. generic-eng
     2. simulator
     3. full_passion-userdebug
     4. full_crespo4g-userdebug
     5. full_crespo-userdebug

Which would you like? [generic-eng] ENTER

============================================
PLATFORM_VERSION_CODENAME=REL
```

† The OpenJDK and *gcj* won't do.

```
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=generic
TARGET_BUILD_VARIANT=eng
TARGET_SIMULATOR=false
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=GINGERBREAD
============================================
```

Note that we typed a period (.), SPACE, and then `build/envsetup.sh`. This forces the shell to run the *envsetup.sh* script within the current shell. If we were to just run the script, the shell would spawn a new shell and run the script in that new shell. That would be useless since *envsetup.sh* defines new shell commands, such as *lunch*, and sets up environment variables required for the rest of the build.

We will explore *envsetup.sh* and *lunch* in more detail later. For the moment, though, note that the `generic-eng` *combo* means that we configured the build system to create images for running in the Android emulator. This is the same QEMU emulator software used by app developers to test their apps when developing using the SDK on a workstation, albeit here it will be running our own custom images instead of the default ones shipped with the SDK. It's also the same emulator that was used by the Android development team to develop Android while there were no devices for it yet. So while it's not real hardware and is therefore by no means a pefect target, it's still more than sufficient to cover most of the terrain we need to cover. Once you know your specific target, you should be able to adapt the instructions found in the rest of this book, with possibly some help from the book *Building Embedded Linux Systems*, to get your custom Android images loaded on your device and your hardware to boot them.

Now that the environment has been set up, we can actually build Android:

```
$ make -j16
============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=generic
TARGET_BUILD_VARIANT=eng
TARGET_SIMULATOR=false
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=GINGERBREAD
============================================
Checking build tools versions...
```

```
find: `frameworks/base/frameworks/base/docs/html': No such file or directory
find: `out/target/common/docs/gen': No such file or directory
find: `frameworks/base/frameworks/base/docs/html': No such file or directory
find: `out/target/common/docs/gen': No such file or directory
find: `frameworks/base/frameworks/base/docs/html': No such file or directory
find: `out/target/common/docs/gen': No such file or directory
find: `frameworks/base/frameworks/base/docs/html': No such file or directory
find: `out/target/common/docs/gen': No such file or directory
find: `frameworks/base/frameworks/base/docs/html': No such file or directory
find: `out/target/common/docs/gen': No such file or directory
host Java: apicheck (out/host/common/obj/JAVA_LIBRARIES/apicheck_intermediates/classes)
Header: out/host/linux-x86/obj/include/libexpat/expat.h
Header: out/host/linux-x86/obj/include/libexpat/expat_external.h
Header: out/target/product/generic/obj/include/libexpat/expat.h
Header: out/target/product/generic/obj/include/libexpat/expat_external.h
Header: out/host/linux-x86/obj/include/libpng/png.h
Header: out/host/linux-x86/obj/include/libpng/pngconf.h
Header: out/host/linux-x86/obj/include/libpng/pngusr.h
Header: out/target/product/generic/obj/include/libpng/png.h
Header: out/target/product/generic/obj/include/libpng/pngconf.h
Header: out/target/product/generic/obj/include/libpng/pngusr.h
Header: out/target/product/generic/obj/include/libwpa_client/wpa_ctrl.h
Header: out/target/product/generic/obj/include/libsonivox/eas_types.h
Header: out/target/product/generic/obj/include/libsonivox/eas.h
Header: out/target/product/generic/obj/include/libsonivox/eas_reverb.h
Header: out/target/product/generic/obj/include/libsonivox/jet.h
Header: out/target/product/generic/obj/include/libsonivox/ARM_synth_constants_gnu.inc
host Java: clearsilver (out/host/common/obj/JAVA_LIBRARIES/clearsilver_intermediates/classes)
target Java: core (out/target/common/obj/JAVA_LIBRARIES/core_intermediates/classes)
host Java: dx (out/host/common/obj/JAVA_LIBRARIES/dx_intermediates/classes)
Notice file: frameworks/base/libs/utils/NOTICE -- out/host/linux-x86/obj
    /NOTICE_FILES/src//lib/libutils.a.txt
Notice file: system/core/libcutils/NOTICE -- out/host/linux-x86/obj/NOTICE_FILES/src//lib
    /libcutils.a.txt
...
```

Now is a good time to go for a snack or watch tonight's hockey game.‡ On a more serious note, though, your build time will obviously depend on your system's capabilities. On a laptop equiped with a quad-core CORE i7 Intel processor with hyper-threading enabled and 8GB of RAM, this actua command will take about 20 minutes to build the AOSP. On an older laptop with a dual-core Centro 2 Intel processor and 2GB of RAM, a *make -j4* would take about an hour to build the same AOSP. Note that the `-j` parameter of *make* allows you to specify how many jobs to run in parallel. Some say that it's best to use your number of processors times 2, which is what I'm doing here. Others say it's best to add 2 to the number of processors you have. Following that advice, I would've used 10 and 4 instead of 16 and 4.

‡ It's a Canadian thing, I can't help it.

Building on Virtual Machines or Non-Ubuntu Systems

I often get asked about building the AOSP in virtual machines; most often because the development team, or their IT department, is standardized on Windows. While I've seen this work and have put together images to do that myself, your results will vary. It'll usually take more than twice as much time to build in a VM than building natively on the same system. So if you're going to do a lot of work on the AOSP, I highly suggest you build it natively. And yes, this involves having a Linux machine at hand.

An increasing number of developers also prefer MacOS X over Linux and Windows, including many at Google itself. Hence, the official instructions at *http://source.android.com* also describe how to build on a Mac. These instructions, though, tend to break after Mac OS updates. Fortunately for Mac-based developers, they are many and they are rather zealous. Hence, you'll eventually find updated instructions on the web or on the various Google Groups about how to build the AOSP on your new version of MacOS X. Here's one posting explaining how to build Gingerbread on MacOS X Lion: Building Gingerbread on OS X Lion. Bear in mind, though, that as I mentioned in Chapter 1, Google's own Android build farms are Ubuntu-based. If you choose to build on MacOS X, you'll likely always be playing catch-up. At worst, you can use a VM as in the Windows case.

If you do choose to go down the VM route, make sure you configure the VM to use as many CPUs as there are available in your system. Most BIOSes I've seen seem to disable by the default the option for enabling CPU instructions sets allowing mutlitple CPU virtualization. VirtualBox, for instance, will complain about some obscur error if you try to allocate more than one CPU to an image while those instruction sets are disabled. You must go to the BIOS and enable those options for your VM software to be able to grant the guest OS multiple CPUs.

There are a few other things to consider regarding the build. First, note that in between printing out the build configuration and the printing of the first output of the actual build (where it prints out: "host Java: apicheck (out/host/common/o..."), there will be a rather long delay where nothing will get printed out, save for the "No such file or directory" warnings. I'll explain this delay in more detail later, but suffice it to say for now that during that time the build system is figuring out the rules of how to build every part of the AOSP.

Note also that you'll see plenty of warning statements. These are rather "normal," not so much in terms of maintaining software quality, but in that they are pervasive in Android's build. They usually won't have an impact on the final product being compiled. So, contrary to the best of my software engineeering instincts, I have to recommend you completely ignore warnings and stick to fixing errors only. Unless, of course, those warnings stem from software you added yourself. Then, by all means, make sure you get rid of **those** warnings.

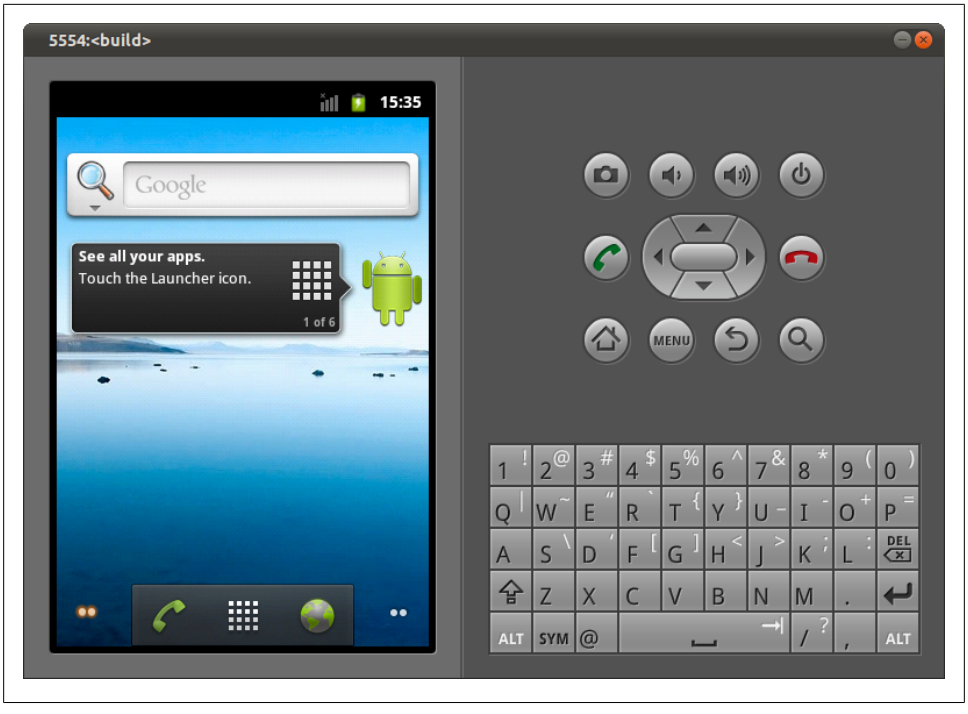


Figure 3-2. Android emulator running custom images

Running Android

With the build completed, all you need to do is start the emulator to run your own custom-built images:

```
$ emulator &
```

This will start the emulator window that will boot into a full Android environment as illustrated in Figure 3-2.

You can then interact with the AOSP you just built very much in the same way as if it were running on a real device. Since your monitor is likely not a touch screen, however, you will need to use your mouse as if it was your finger. A single touch is a click and swipping is done by holding down the mouse button, moving around and letting go of the mouse button to signify that your finger has been removed from the touchscreen. You also have a full keyboard at your disposal, with all the buttons you would find on a phone equipped with a QWERTY keyboard, although you can use your regular keyboard to input text in text boxes.

Despite its features and realism, the emulator does have its issues. For one thing, it takes some time to boot. It will take longest to boot the first time, because Dalvik is

creating a JIT cache for the apps running on the phone. Even later, though, you might find it heavy, especially if you're in a modify-compile-test loop. Also, the emulator doesn't perfectly emulate everything. For instance, it traditionally has a hard time firing off rotation change events when it's made to rotate using F11 or F12. This issue, though, is mostly an issue for app developers.

If for any reason you close the shell where you had configured, built, and started Android, or if you need to start a new one and have access to all the tools and binaries created from the build, you must invoke the *envsetup.sh* script and the *lunch* commands again in order to set up environment variables. Here are commands from a new shell, for instance:

```
$ cd ~/android/aosp-2.3.x
$ emulator &
No command 'emulator' found, did you mean:
 Command 'qemulator' from package 'qemulator' (universe)
emulator: command not found
$ . build/envsetup.sh
$ lunch

You're building on Linux

Lunch menu... pick a combo:
     1. generic-eng
     2. simulator
     3. full_passion-userdebug
     4. full_crespo4g-userdebug
     5. full_crespo-userdebug

Which would you like? [generic-eng] ENTER

============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=generic
TARGET_BUILD_VARIANT=eng
...
============================================
$ emulator &
$
```

Note that the second time we issued *emulator*, the shell didn't complain that the command was missing anymore. The same goes for a lot of other Android tools such as the *adb* command we're about to look at. Note also that we didn't need to issue any *make* commands, because we had already built Android. In this case, we just needed to make sure the environment variables were properly set in order for the results of the previous build to be available to us again.

Using ADB

One of the most interesting aspects of the development environment put together by the Android development team is that you can shell into the running emulator, or any real device connected through USB for that matter, using the *adb* tool:

```
$ adb shell ❶
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
# cat /proc/cpuinfo ❷
Processor       : ARM926EJ-S rev 5 (v5l)
BogoMIPS        : 405.50
Features        : swp half thumb fastmult vfp edsp java
CPU implementer : 0x41
CPU architecture: 5TEJ
CPU variant     : 0x0
CPU part        : 0x926
CPU revision    : 5

Hardware        : Goldfish
Revision        : 0000
Serial          : 0000000000000000
```

❶ This is issued in the same shell where you started the emulator from.

❷ This is the target's shell, and *cat* is actually running on the "target" (i.e., the emulator.)

As you can see, the kernel running in the emulator reports that it's seeing an ARM processor, which is in fact the predominant platform used with Android. Also, the kernel says it's running on a platform called *Goldfish*. This is the code-name for the emulator and you will see it in quite a few places.

Now that you've got a shell into the emulator and you are root, which is the default in the emulator, you can run any command much like if you had shelled into a remote machine or a traditional network-connected embedded Linux system. ADB is what makes this possible and Figure 3-3 illustrates its many components and how they're connected.

To exit an ADB shell session, all you need to do is type CTRL-D:

```
# CTRL-D ❶
$ ❷
```

❶ This is in the target shell

❷ This is back on the host

When you start *adb* for the first time on the host, it starts a server in the background whose job is to manage the connections to all Android devices connected to the host.

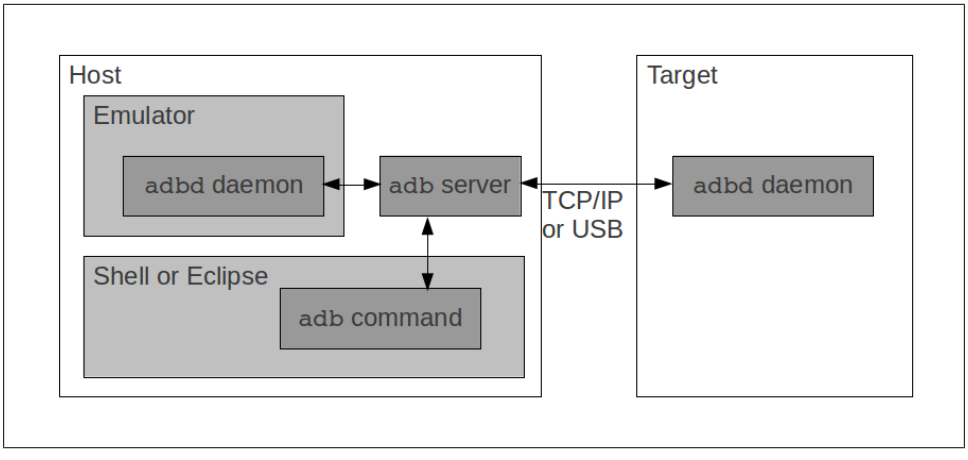


Figure 3-3. How ADB's parts are interconnected

That was the part of the earlier output that said that a daemon was being started on port 5037. You can actually ask that daemon what devices it sees:

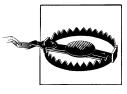
```
$ adb devices
List of devices attached
emulator-5554    device
0000021459584822    device
emulator-5556    offline
```

This is the output with one emulator instance running, one device connected through USB, and another emulator instance starting up. If there are multiple devices connected, you can tell it which device you want to talk to using the `-s` flag to identify the serial number of the device:

```
$ adb -s 0000021459584822 shell
$ id
uid=2000(shell) gid=2000(shell) groups=1003(graphics),1004(input), ...
$ su
su: permission denied
```

Note that in this case, I'm getting a `$` for my shell prompt instead of a `#`. This means that contrary to the earlier interaction, I'm not running as root, as can also be seen from the output of the *id* command. This is actually a real commercial Android phone, and my inability above to gain root priviledges using the *su* command is typical. Hence, my ability to make any modifications to this device will be fairly limited. Unless, of course, I find some way to "root" the phone (i.e. gain root access). Unfortunately, device manufacturers have been historically very reluctant for various reasons to give root access to their devices and have put in a number of provisions to make that as difficult as possible, if not impossible. That's why "rooting" devices is held up as a holy grail by

many power users and hackers. As of the summer of 2011, though, some manufacturers such as Motorola and HTC have spelled out a change in policy where they seem to be intent on making it easier for users to root their devices, with caveats of course. But this isn't mainstream yet.



You may be tempted to try to root a commercial phone or device for experimenting with Android platform development. I would suggest you think this through carefully. While there are plenty of instructions out there explaining how to replace your standard images with what is often referred to as "custom ROMs" such as Cyanogenmod and others, you need to be aware that any false step could well result in your "bricking" the device (i.e. rendering it unbootable or erasing critical boot-time code). You then have an expensive paper-weight (hence the term "bricking") instead of a phone.

If you want to experiment with running custom AOSP builds on real hardware, I suggest you get yourself something like a BeagleBoard xM or a PandaBoard. These boards are made for tinkering. If nothing else, they don't have a built-in flash chip that you may risk damaging. Instead, the SoCs on those devices boot straight from SD cards. Hence, fixing a broken image is simply a matter of unplugging the SD card from the board, connecting it to your workstation, reprogramming it, and plugging it back to the board.

adb can of course do a lot more than just give you a shell, and I encourage you to start it without any parameters to look at its usage output:

```
$ adb
Android Debug Bridge version 1.0.26

 -d                            - directs command to the only connected USB device
                                 returns an error if more than one USB device is present.
 -e                            - directs command to the only running emulator.
                                 returns an error if more than one emulator is running.
 -s <serial number>            - directs command to the USB device or emulator with
                                 the given serial number. Overrides ANDROID_SERIAL
...
device commands:
  adb push <local> <remote>    - copy file/dir to device
  adb pull <remote> [<local>]  - copy file/dir from device
  adb sync [ <directory> ]     - copy host->device only if changed
                                 (-l means list but don't copy)
                                 (see 'adb help all')
  adb shell                    - run remote shell interactively
  adb shell <command>          - run remote shell command
  adb emu <command>            - run emulator console command
...
```

You can, for instance, use *adb* to dump the data contained in the main logger buffer:

```
$ adb logcat
I/DEBUG   (   30): debuggerd: Sep 10 2011 13:44:19
I/Netd    (   29): Netd 1.0 starting
I/Vold    (   28): Vold 2.1 (the revenge) firing up
D/qemud   (   38): entering main loop
D/Vold    (   28): USB mass storage support is not enabled in the kernel
D/Vold    (   28): usb_configuration switch is not enabled in the kernel
D/Vold    (   28): Volume sdcard state changing -1 (Initializing) -> 0 (No-Media)
D/qemud   (   38): fdhandler_accept_event: accepting on fd 9
D/qemud   (   38): created client 0xe078 listening on fd 10
D/qemud   (   38): client_fd_receive: attempting registration for service 'boot-properties'
D/qemud   (   38): client_fd_receive:    -> received channel id 1
D/qemud   (   38): client_registration: registration succeeded for client 1
I/qemu-props(   54): connected to 'boot-properties' qemud service.
I/qemu-props(   54): receiving..
I/qemu-props(   54): received: qemu.sf.lcd_density=160
I/qemu-props(   54): receiving..
I/qemu-props(   54): received: dalvik.vm.heapsize=16m
I/qemu-props(   54): receiving..
D/qemud   (   38): fdhandler_event: disconnect on fd 10
I/qemu-props(   54): exiting (2 properties set).
D/AndroidRuntime(   32):
D/AndroidRuntime(   32): >>>>>> AndroidRuntime START com.android.internal.os.ZygoteInit <<<<<<
D/AndroidRuntime(   32): CheckJNI is ON
I/        (   33): ServiceManager: 0xad50
...
```

This is very useful to observe the runtime behavior of key system components, including services run by the System Server.

You can also copy files to and from the device:

```
$ adb push data.txt /data/local
1 KB/s (87 bytes in 0.043s)
$ adb pull /proc/config.gz
95 KB/s (7087 bytes in 0.072s)
```

Again, given its centrality to Android development, I invite you to read up on ADB's use. We will continue using it throughout the book and introduce more of its functionalities as we go. Keep in mind, though, that ADB can have its quirks. First and foremost, many have found its host-side daemon to be somewhat flaky. For some reason or another, it sometimes doesn't correctly identify the state of connected devices and continues to state that they are offline while you try connecting to them. Or *adb* might just hang on the command line waiting for the device while the device is clearly active and able to receive ADB commands. The solution to those issues is almost invariably to kill the host-side daemon:§

§ It's actually somewhat interesting that the Android development team felt the need to build such functionality right into *adb*. Clearly they were encountering issues with that daemon themselves.

```
$ adb kill-server
```

Not to worry—the next time you issue any *adb* command, the daemon will get automatically restarted. It's unclear what causes this behavior, and maybe this problem will get resolved at some point in the future. In the mean time, keep in mind that if you see some odd behavior when using ADB, killing the host-side daemon is usually something you want to try before investigating other potential issues.

In addition to the command itself, another source of information on *adb* is the Android Debug Bridge part of Google's Android Developers Guide. As Tim Bird‖ recommends, you want to print a copy and put it under your pillow.

Mastering the Emulator

As I said earlier, you can go a long way in platform development by simply using the emulator. It effectively emulates an ARM target with a minimal set of hardware. We'll spend some time here going through some more advanced aspects of dealing with the emulator. As many Android pieces, the emulator is quite a complex piece of software in and of itself. Still, we can get a very good idea of it capabilities by surveying a few key features.

Earlier we started the emulator by simply typing:

```
$ emulator &
```

But the *emulator* command can also take quite a few parameters. You can see the online help by adding the `-help` flag on the command line:

```
$ emulator -help
Android Emulator usage: emulator [options] [-qemu args]
  options:
    -sysdir <dir>                 search for system disk images in <dir>
    -system <file>                read initial system image from <file>
    -datadir <dir>                write user data into <dir>
    -kernel <file>                use specific emulated kernel
    -ramdisk <file>               ramdisk image (default <system>/ramdisk.img
    -image <file>                 obsolete, use -system <file> instead
    -init-data <file>             initial data image (default <system>/userdata.img
    -initdata <file>              same as '-init-data <file>'
    -data <file>                  data image (default <datadir>/userdata-qemu.img
    -partition-size <size>        system/data partition size in MBs
...
```

‖ Tim is the maintainer of *http://elinux.org*, the guy behind the Embedded Linux Conference, the chair of the Linux Foundation's CE Workgroup, etc. and he's been doing a lot of cool Android stuff at Sony.

One especially useful flag is `-kernel`. It allows you to tell the emulator to use another kernel than the default prebuilt one found in *prebuilt/android-arm/kernel/*:

```
$ emulator -kernel path_to_your_kernel_image/zImage
```

If you want to use a kernel that has module support, for instance, you'll need to build your own, because the prebuilt one doesn't have module support enabled by default. Also, by default, the emulator won't show you the kernel's boot messages. You can, however, pass the `-show-kernel` flag to see them:

```
$ emulator -show-kernel
Uncompressing Linux.....................................................................................
Initializing cgroup subsys cpu
Linux version 2.6.29-00261-g0097074-dirty (digit@digit.mtv.corp.google.com)
(gcc version 4.4.0 (GCC) ) #20 Wed Mar 31 09:54:02 PDT 2010
CPU: ARM926EJ-S [41069265] revision 5 (ARMv5TEJ), cr=00093177
CPU: VIVT data cache, VIVT instruction cache
Machine: Goldfish
Memory policy: ECC disabled, Data cache writeback
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 24384
Kernel command line: qemu=1 console=ttyS0 android.checkjni=1 android.qemud=ttyS1 android.ndns=3
Unknown boot option `android.checkjni=1': ignoring
Unknown boot option `android.qemud=ttyS1': ignoring
Unknown boot option `android.ndns=3': ignoring
PID hash table entries: 512 (order: 9, 2048 bytes)
Console: colour dummy device 80x30
Dentry cache hash table entries: 16384 (order: 4, 65536 bytes)
Memory: 96MB = 96MB total
Memory: 91548KB available (2616K code, 681K data, 104K init)
Calibrating delay loop... 403.04 BogoMIPS (lpj=2015232)
Mount-cache hash table entries: 512
Initializing cgroup subsys debug
Initializing cgroup subsys cpuacct
Initializing cgroup subsys freezer
CPU: Testing write buffer coherency: ok
...
```

You can also have the emulator print out information about its own execution using the `-verbose` flag, thereby allowing you to see, for example, which images files it's using:

```
$ emulator -verbose
emulator: found Android build root: /home/karim/android/aosp-2.3.x
emulator: found Android build out:  /home/karim/android/aosp-2.3.x/out/target/product/generic
emulator:     locking user data image at /home/karim/android/aosp-2.3.x/out/target/product
    /generic/userdata-qemu.img
emulator: selecting default skin name 'HVGA'
emulator: found skin-specific hardware.ini: /home/karim/android/aosp-2.3.x/sdk/emulator/skins
    /HVGA/hardware.ini
emulator: autoconfig: -skin HVGA
emulator: autoconfig: -skindir /home/karim/android/aosp-2.3.x/sdk/emulator/skins
```

```
emulator: keyset loaded from: /home/karim/.android/default.keyset
emulator: trying to load skin file '/home/karim/android/aosp-2.3.x/sdk/emulator/skins
    /HVGA/layout'
emulator: skin network speed: 'full'
emulator: skin network delay: 'none'
emulator: no SD Card image at '/home/karim/android/aosp-2.3.x/out/target/product/generic
    /sdcard.img'
emulator: registered 'boot-properties' qemud service
emulator: registered 'boot-properties' qemud service
emulator: Adding boot property: 'qemu.sf.lcd_density' = '160'
emulator: Adding boot property: 'dalvik.vm.heapsize' = '16m'
emulator: argv[00] = "emulator"
emulator: argv[01] = "-kernel"
emulator: argv[02] = "/home/karim/android/aosp-2.3.x/prebuilt/android-arm/kernel/kernel-qemu"
emulator: argv[03] = "-initrd"
emulator: argv[04] = "/home/karim/android/aosp-2.3.x/out/target/product/generic/ramdisk.img"
emulator: argv[05] = "-nand"
emulator: argv[06] = "system,size=0x4200000,initfile=/home/karim/android/aosp-2.3.x/out
    /target/product/generic/system.img"
emulator: argv[07] = "-nand"
emulator: argv[08] = "userdata,size=0x4200000,file=/home/karim/android/aosp-2.3.x/out/target
    /product/generic/userdata-qemu.img"
emulator: argv[09] = "-nand"
...
```

Up to this point, I've used the terms QEMU and emulator interchangeably. The reality, though, is that the *emulator* command isn't actually QEMU: it's a custom wrapper around it created by the Android development team. You can, however, interact with the emulator's QEMU by using the `-qemu` flag. Anything you pass after that flag is passed on to QEMU and not the *emulator* wrapper:

```
$ emulator -qemu -h
QEMU PC emulator version 0.10.50Android, Copyright (c) 2003-2008 Fabrice Bellard
usage: qemu [options] [disk_image]

'disk_image' is a raw hard image image for IDE hard disk 0

Standard options:
-h or -help     display this help and exit
-version        display version information and exit
-M machine      select emulated machine (-M ? for list)
-cpu cpu        select CPU (-cpu ? for list)
-smp n          set the number of CPUs to 'n' [default=1]
-numa node[,mem=size][,cpus=cpu[-cpu]][,nodeid=node]
-fda/-fdb file  use 'file' as floppy disk 0/1 image
-hda/-hdb file  use 'file' as IDE hard disk 0/1 image
...
$ emulator -qemu -...
```

We saw earlier how we can use *adb* to interact with the AOSP running within the emulator, and we just saw how we can use various options to change the way the

emulator is started. Interestingly, we can also control the emulator's behavior at runtime by *telnet*ing into it. Every emulator instance that starts is assigned a port number on the host. Go back to Figure 3-2 and check the top-left corner of the emulator's window. That number up there (5554 in this case) is the port number at which that emulator instance is listening. The next emulator that starts simultaneously will get 5556, the next 5558, and so on. To get access to the emulator's special console, you can use the regular *telnet* command:

```
$ telnet localhost 5554
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Android Console: type 'help' for a list of commands
OK
help
Android console command help:

    help|h|?         print a list of commands
    event            simulate hardware events
    geo              Geo-location commands
    gsm              GSM related commands
    kill             kill the emulator instance
    network          manage network settings
    power            power related commands
    quit|exit        quit control session
    redir            manage port redirections
    sms              SMS related commands
    avd              manager virtual device state
    window           manage emulator window

try 'help <command>' for command-specific help
OK
```

Using that console you can do some nifty tricks like redirecting a port from the host to the target:

```
redir add tcp:8080:80
OK
redir list
tcp:8080  => 80
OK
```

From here on, anything accessing 8080 on your host will actually be speaking to whatever is listening to port 80 on that emulated Android. Nothing listens to that port by default on Android, but you can, for example, have BusyBox's *httpd* running on Android and connect to it in this way.

The emulator also exposes a few "magic" IPs to the emulated Android. IP address 10.0.2.2, for instance, is an alias to your workstation's 127.0.0.1. If you have Apache

running on your workstation, you can open the emulator's browser and type `http://10.0.2.2` and you'll be able to browse whatever content is served up by Apache.

For more information on how to operate the emulator and its various options, have a look at the Using the Android Emulator section of Google's Android Developers Guide. It's written for an app developer audience, but it will still be very useful to you even if you're doing platform work.

CHAPTER 4

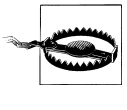
The Build System

The goal of the past chapter was to get you up and running as fast as possible with custom AOSP development. There's nothing precluding you from closing this book at this point and start to dig in and modify your AOSP tree to fit your needs. All you need to do to test your modifications is rebuild the AOSP, start the emulator again and, if need be, shell back into it using ADB. If you want to maximize your efforts, however, you'll likely want some insight on Android's build system.

Despite it being modular, Android's build system is fairly complex and doesn't resemble any of the mainstream build systems out there; none that are used for most open source projects at least. Specifically, it uses *make* in a fairly unconventional way and doesn't provide any sort of menuconfig-based configuration (or equivalent for that matter.) Android very much has its own build paradigm that takes some time to get used to. So pack yourself a good coffee or two, things are about to get serious.

Comparisons With Other Build Systems

Before I start explaining how Android's build system works, allow me to begin by emphasizing how it differs from what you might already know. First and foremost, unlike most *make*-based build systems, the Android build system doesn't rely on recursive makefiles. Unlike the Linux kernel for instance, there isn't a top-level makefile that will recursively invoke subdirectories' makefiles. Instead, there is a script that explores all directories and subdirectories until it finds an *Android.mk* file, whereupon it stops and doesn't explore the subdirectories underneath that file's location. Note that Android doesn't rely on makefiles called *Makefile*. Instead, it's the *Android.mk* files that specify how the local "module" is built.



Android build "modules" have nothing to do with kernel "modules." Within the context of Android's build system, a "module" is any component of the AOSP that needs to be built. This might be a binary, an app package, a library, etc. and it might have to be built for the target or the host, but it's still a "module" with regards to the build system.

Another Android specificity is the way the build system is configured. While most of us are used to systems based on kernel-style menuconfig or GNU autoconf/automake, Android relies on a set of variables that are either set dynamically as part of the shell's environment by way of *envsetup.sh* and *lunch*, or are defined statically ahead of time in a *buildspec.mk* file. Also—always seeming to be a surprise to newcomers—the level of configurability made possible by Android's build system is fairly limited. So while you can specify the properties of the target for which you want the AOSP to be built and, to a certain extent, which apps should be included by default in the resulting AOSP, there is no way for you enable/disable features as is possible a-la menuconfig. You can't, for instance, decide that you don't want Wifi support or that you don't want the Location Service to start by default.

Also, the build system doesn't generate object files or any sort of intermediate output within the same location as the source files. You won't find the *.o* files alongside their *.c* source files within the source tree, for instance. In fact, none of the existing AOSP directories are used in any of the output. Instead, the build system creates an *out/* directory where it stores everything it generates. Hence, a *make clean* is very much the same thing as a *rm -rf out/*. In other words, removing the *out/* directory wipes out anything that was built.

The last thing to say about the build system before we start exploring it in more detail is that it's heavily tied to *GNU make*. And, more to the point, versions 3.81 or newer of it. The build system in fact heavily relies on many GNU make-specific features such as the `define`, `include`, and `ifndef` directives.

Some Background on the Design of Android's Build System

If you would like to get more insight as to the design choices that were made when putting together Android's build system, have a look at the *build/core/build-system.html* file in the AOSP. It's dated May 2006 and seems to have been the document that went around within the Android dev team to get consensus on a rework of the build system. Some of the information and hypothesis are out of date or have been obsoleted, but most of the nuggets of the current build system are there. In general, I've found that the further back the document was created by the Android dev team, the more insightful it is regarding raw motivations and technical background. Newer documents tend to be "cleaned up" and abstract, if they exist at all.

If you want to understand the technical underpinnings of why Android's build system doesn't use recursive *make*, have a look at the paper entitled "Recursive Make Considered Harmful" by Peter Miller in AUUGN Journal of AUUG Inc., 19(1), pp. 14-25. The paper explores the issues surrounding the use of recursive makefiles and explains a different approach involving the use of a single global makefile for building the entire project based on module-provided *.mk* files, which is exactly what Android does.

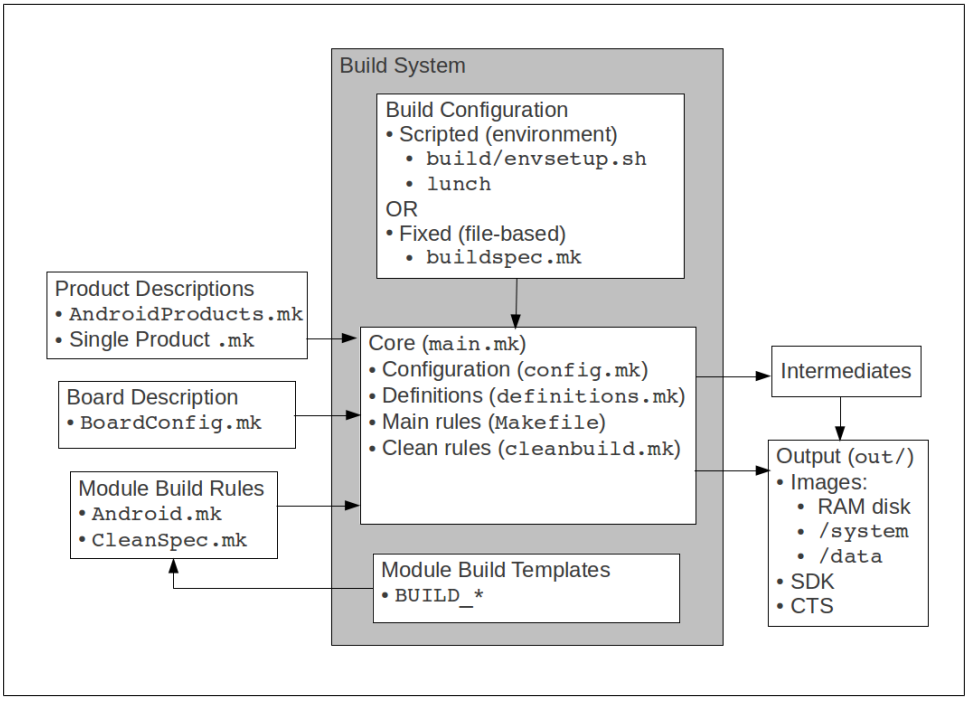


Figure 4-1. Android's build system

Architecture

As illustrated in Figure 4-1, the entry point to making sense of the build system is the *main.mk* file found in the *build/core/* directory, which is invoked through the top-level makefile, as we saw earlier. The *build/core/* directory actually contains the bulk of the build system, and we'll cover key files from there. Again, remember that Android's build system pulls everything into a single Makefile; it isn't recursive. Hence, each *.mk* file you see eventually becomes part of single huge makefile that contains the rules for building all the pieces in the system.

Why does *make* hang?

Every time you type *make*, you witness the aggregation of the *.mk* files into a single set through what might seem like an annoying build artifact: the build system prints out the build configuration and seems to hang for quite some time without printing anything to the screen. After these long moments of screen silence, it then actually starts proceeding again and builds every part of the AOSP, at which point you see regular output to your screen as you'd expect from any regular build system. Anyone who's built the AOSP has wondered what in the world is the build system doing during that time. What it's doing is incorporating every *Android.mk* file it can find the AOSP.

If you want to see this in action, edit the *build/core/main.mk* and replace this line:

```
include $(subdir_makefiles)
```

with this:

```
$(foreach subdir_makefile, $(subdir_makefiles), \
  $(info Including $(subdir_makefile)) \
  $(eval include $(subdir_makefile)) \
 )
subdir_makefile :=
```

The next time you type *make*, you'll actually see what's happening:

```
$ make -j16
============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=generic
...
============================================
Including ./bionic/Android.mk
Including ./development/samples/Snake/Android.mk
Including ./libcore/Android.mk
Including ./external/elfutils/Android.mk
Including ./packages/apps/Camera/Android.mk
Including ./device/htc/passion-common/Android.mk
...
```

Configuration

One of the first things the build system does is pull in the build configuration through the inclusion of *config.mk*. The build can be configured either by the use of the *envsetup.sh* and *lunch* commands or by providing a *buildspec.mk* file at the top-level directory. In either case, some of the following variables need to be set.

`TARGET_PRODUCT`
: Android flavor to be built. Each recipe can, for instance, include a different set of apps or locales or build different parts of the tree. Have a look at the various single product *.mk* files included by the *AndroidProducts.mk* files in *build/target/product/*, *device/samsung/crespo/*, and *device/htc/passion/* for examples. Values include:

  `generic`
  : The "vanilla" kind, the most basic build of the AOSP parts you can have.

  `full`
  : The "all dressed" kind, with most apps and the major locales enabled.

  `full_crespo`
  : Same as `full` but for Crespo (i.e. Samsung Nexus S.)

`sim`
: Android simulator (see sidebar.)

`sdk`
: The SDK; includes a vast number of locales.

`TARGET_BUILD_VARIANT`
: Selects which modules to install. Each module is supposed to have a `LOCAL_MODULE_TAGS` variable set in its *Android.mk* to at least one of:* `user`, `debug`, `eng`, `tests`, `optional`, or `samples`. By selecting the variant, you will tell the build system which module subsets should be included. Specifically:

  `eng`
  : Includes all modules tagged as `user`, `debug` or `eng`.

  `userdebug`
  : Includes both modules tagged as `user` and `debug`.

  `user`
  : Includes only modules tagged as `user`.

`TARGET_BUILD_TYPE`
: Dictates whether or not special build flags are used or `DEBUG` variables are defined in the code. The possible values here are either `release` or `debug`. Most notably, the *frameworks/base/Android.mk* file chooses between either *frameworks/base/core/config/debug* or *frameworks/base/core/config/ndebug*, depending on whether or not this variable is set to `debug`. The former causes the `ConfigBuildFlags.DEBUG` Java constant to be set to `true`, whereas the latter causes it to be set to `false`. Some code in parts of the system services, for instance, is conditional on `DEBUG`. Typically, `TARGET_BUILD_TYPE` is set to `release`.

`TARGET_TOOLS_PREFIX`
: By default, the build system will use one of the cross-development toolchains shipped with it underneath the *prebuilt/* directory. However, if you'd like it to use another toolchain, you can set this value to point to its location.

`OUT_DIR`
: By default, the build system will put all build output into the *out/* directory. You can use this variable to provide an alternate output directory.

`BUILD_ENV_SEQUENCE_NUMBER`
: If you use the template *build/buildspec.mk.default* to create your own *buildspec.mk* fiile, this value will be properly set. However, if you create a *buildspec.mk* with an older AOSP release and try to use it in a future AOSP release that contains important changes to its build system and, hence, a different value, this

* If you do not provide a value, defaults will be used. For instance, all apps are set to `optional` by default. Also, some modules are part of `GRANDFATHERED_USER_MODULES` in *user_tags.mk*. No `LOCAL_MODULE_TAGS` need be specified for those; they're always included.

variable will act as a safety net. It will cause the build system to inform you that your *buildspec.mk* file doesn't match your build system.

Android Simulator

If you go back to the menu printed by *lunch* in "Building Android" on page 69, you'll notice an entry called `simulator`. In fact you'll find references to the simulator at a number of locations, including quite a few *Android.mk* files and subdirectories in the tree. The most important thing you need to know about the simulator is that it has *nothing* to do with the emulator. They are two completely different things.

That said, the simulator appears to be a remnant of the Android's team early work to create Android. Since at the time they didn't even have Android running in QEMU, they used their desktop OSes and the `LD_PRELOAD` mechanism to simulate an Android device, hence the term "simulator." It appears that they stopped using it as soon as running Android on QEMU became possible. It's still there, though, as it can be useful for building parts of the AOSP for development and testing on developer workstations.

That doesn't mean that you run the AOSP on your desktop. In fact you can't, if nothing else because you need a kernel that has Binder included and you would need to be using Bionic instead of your system's default C library. But, if you want to run parts of what's built from the AOSP on your desktop, this product target will allow you to do so.

Various parts of the code build very differently if the target is the simulator. When browsing the code, for example, you'll sometimes find conditional builds around the `HAVE_ANDROID_OS` C macro.† The code that talks to the Binder is one of these. If `HAVE_ANDROID_OS` is not defined, that code will return an error to its caller instead of trying to actually talk to the Binder driver.

For the full story behind the simulator, have a look at Android developer Andrew McFadden's response to a post entitled "Android Simulator Environment" on the android-porting mailing list in April 2009.

In addition to selecting which parts of the AOSP to build and which options to build them with, the build system also needs to know about the target it's building for. This is provided through a *BoardConfig.mk* file which will specify things such as the command line to be provided to the kernel, the base address at which the kernel should be loaded, or the instruction set version most appropriate for the board's CPU (`TARGET_ARCH_VARIANT`.) Have a look at *build/target/board/* for a set of per-target directories that each contain a *BoardConfig.mk* file. Also have a look at the various *device/*/TARGET_DEVICE/BoardConfig.mk* files included in the AOSP. The latter are much richer than the former because they contain a lot more hardware-specific information. The device name (i.e. *`TARGET_DEVICE`*) is derived from the `PRODUCT_DEVICE` specified in the product *.mk* file provided for the `TARGET_PRODUCT` set in the configuration. For example, *device/samsung/crespo/AndroidProducts.mk* includes *device/samsung/crespo/*

† `HAVE_ANDROID_OS` is only defined when compiling for the simulator.

full_crespo.mk, which sets `PRODUCT_DEVICE` to `crespo`. Hence, the build system looks for a *BoardConfig.mk* in *device/*/crespo/*, and there happens to be one at that location.

The final piece of the puzzle with regard to configuration is the CPU-specific options used to build Android. For ARM, those are contained in *build/core/combo/arch/arm/armv*.mk*, with `TARGET_ARCH_VARIANT` determining the actual file to use. Each file lists CPU-specific cross-compiler and cross-linker flags used for building C/C++ files. They also contain a number of `ARCH_ARM_HAVE_*` variables that enable others parts of the AOSP to build code conditionally based on whether a given ARM feature is found in the target's CPU.

envsetup.sh

Now that you understand the kinds of configuration input the build system needs, we can actually discuss the role of *envsetup.sh* in more detail. As its name implies, *envsetup.sh* actually is for setting up a build environment for Android. It does only part of the job, though. Mainly, it defines a series of shell commands that are useful to any sort of AOSP work:

```
$ cd ~/android/aosp-2.3.x
$ . build/envsetup.sh
$ help
Invoke ". build/envsetup.sh" from your shell to add the following functions to your environment:
- croot:   Changes directory to the top of the tree.
- m:       Makes from the top of the tree.
- mm:      Builds all of the modules in the current directory.
- mmm:     Builds all of the modules in the supplied directories.
- cgrep:   Greps on all local C/C++ files.
- jgrep:   Greps on all local Java files.
- resgrep: Greps on all local res/*.xml files.
- godir:   Go to the directory containing a file.

Look at the source to view more functions. The complete list is:
add_lunch_combo cgrep check_product check_variant choosecombo chooseproduct choosetype
choosevariant cproj croot findmakefile gdbclient get_abs_build_var getbugreports
get_build_var getprebuilt gettop godir help isviewserverstarted jgrep lunch m mm mmm
pgrep pid printconfig print_lunch_menu resgrep runhat runtest set_java_home setpaths
set_sequence_number set_stuff_for_environment settitle smoketest startviewserver
stopviewserver systemstack tapas tracedmdump
```

You'll likely find the *croot* and *godir* commands quite useful for traversing the tree. Some parts of it are quite deep, given the use of Java and its requirement that packages be stored in directory trees bearing the same hierarchy as each sub-part of the corresponding fully-qualified package name.‡ Hence, it's not rare to find yourself 7 to 10 directories underneath the AOSP's top-level directory, and it rapidly becomes tedious to type something like *cd ../../../* ... to return to an upper part of the tree.

‡ For instance, a file part of the `com.foo.bar` package must be stored under the *com/foo/bar/* directory.

m and *mm* are also quite useful since they allow you to, respectively, build from the top-level regardless of where you are or just build the modules found in the current directory. For example, if you made a modification to the Launcher and are in *packages/apps/Launcher2*, you can rebuild just that module by typing *mm* instead of *cd*'ing back to the top-level and typing *make*. Note that *mm* doesn't rebuild the entire tree and, therefore, won't regenerate AOSP images even if a dependent module has changed. *m* will do that, though. Still, *mm* can be useful to test whether your local changes break the build or not until you're ready to regenerate the full AOSP.

Although the online help doesn't mention *lunch*, it is one of the commands defined by *envsetup.sh*. When you run *lunch* without any parameters, it shows you a list of potential choices:

```
$ lunch

You're building on Linux

Lunch menu... pick a combo:
     1. generic-eng
     2. simulator
     3. full_passion-userdebug
     4. full_crespo4g-userdebug
     5. full_crespo-userdebug

Which would you like? [generic-eng]
```

These choices are not static. Most depend on what's in the AOSP at the time *envsetup.sh* runs. They're in fact individually added using the `add_lunch_combo()` function that the script defines. So, for instance, by default *envsetup.sh* adds `generic-eng` and `simulator`:

```
# add the default one here
add_lunch_combo generic-eng

# if we're on linux, add the simulator.  There is a special case
# in lunch to deal with the simulator
if [ "$(uname)" = "Linux" ] ; then
    add_lunch_combo simulator
fi
```

envsetup.sh also includes all the vendor supplied scripts it can find:

```
# Execute the contents of any vendorsetup.sh files we can find.
for f in `/bin/ls vendor/*/vendorsetup.sh vendor/*/build/vendorsetup.sh device/*/*/
vendorsetup.sh 2> /dev/null`
do
    echo "including $f"
    . $f
```

```
done
```

The *device/samsung/crespo/vendorsetup.sh* file, for instance, does this:

```
add_lunch_combo full_crespo-userdebug
```

So that's how you end up with the menu we saw earlier. Note that the menu asks you to choose a *combo*. Essentially, this is a combination of a `TARGET_PRODUCT` and `TARGET_BUILD_VARIANT`, with the exception of the `simulator`. The menu provides the default combinations, but the others remain valid still and can be passed to *lunch* as parameters on the command line:

```
$ lunch generic-user

============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=generic
TARGET_BUILD_VARIANT=user
TARGET_SIMULATOR=false
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=GINGERBREAD
============================================

$ lunch full_crespo-eng

============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=full_crespo
TARGET_BUILD_VARIANT=eng
TARGET_SIMULATOR=false
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=GINGERBREAD
============================================
```

Once *lunch* has finished running for a `generic-eng` combo, it will set up environment variables described in Table 4-1 in your current shell to provide the build system with the required configuration information.

Table 4-1. Environment variables set by lunch (in no particular order)

Variable	Value
PATH	$ANDROID_JAVA_TOOLCHAIN:$PATH:$ANDROID_BUILD_PATHS
ANDROID_EABI_TOOLCHAIN	*aosp-root*/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin
ANDROID_TOOLCHAIN	$ANDROID_EABI_TOOLCHAIN
ANDROID_QTOOLS	*aosp-root*/development/emulator/qtools
ANDROID_BUILD_PATHS	*aosp-root*/out/host/linux-x86:$ANDROID_TOOLCHAIN: $ANDROID_QTOOLS:$ANDROID_TOOLCHAIN:$ANDROID_EABI_TOOL CHAIN
ANDROID_BUILD_TOP	*aosp-root*
ANDROID_JAVA_TOOLCHAIN	$JAVA_HOME/bin
ANDROID_PRODUCT_OUT	*aosp-root*/out/target/product/generic
OUT	ANDROID_PRODUCT_OUT
BUILD_ENV_SEQUENCE_NUMBER	10
OPROFILE_EVENTS_DIR	*aosp-root*/prebuilt/linux-x86/oprofile
TARGET_BUILD_TYPE	release
TARGET_PRODUCT	generic
TARGET_BUILD_VARIANT	eng
TARGET_BUILD_APPS	empty
TARGET_SIMULATOR	false
PROMPT_COMMAND	\"\033]0;[${TARGET_PRODUCT}-${TARGET_BUILD_VARIANT}] ${USER}@${HOSTNAME}: ${PWD}\007\"
JAVA_HOME	/usr/lib/jvm/java-6-sun

Using ccache

If you've already done any AOSP building while reading these pages, you've noticed how long the process is. Obviously, unless you can construct yourself a bleeding edge build farm, any sort of speedup on your current hardware would be greatly appreciated. As a sign that the Android development team might itself also feel the pain of the rather long builds, they've added support for `ccache`. `ccache` stands for *Compiler Cache* and is part of the Samba Project. It's a mechanism that caches the object files generated by the compiler based on the preprocessor's output. Hence, if under two separate builds the preprocessor's output is identical, use of `ccache` will result in the second build not actually using the compiler to build the file. Instead, the cached object file will be copied to the destination where the compiler's output would have been.

To enable the use of `ccache`, all you need to do is make sure that the `USE_CCACHE` environment variable is set to 1 before you start your build:

```
$ export USE_CCACHE=1
```

You won't gain any acceleration the first time you run since the cache will be empty at that time. Every other time you build from scratch, though, the cache will help accelerate the build process. The only downside is that `ccache` is for C/C++ files only. Hence, it can't accelerate the build of any Java file, I must add sadly. There are about 15,000 C/C++ files in the AOSP and 18,000 Java files. So while the cache isn't a panacea, it's interesting.

If you'd like to learn more about `ccache`, have a look at the article titled "Improve collaborative build times with ccache" by Martin Brown on IBM's developerWorks' site. The article also explores the use of *distcc*, which allows you to distribute builds over several machines, allowing you to pool your team's workstations caches together.

Of course, if you get tired of always typing . *build/envsetup.sh* and *lunch*, all you need to do is copy the *build/buildspec.mk.default* into the top-level directory, rename it to *buildspec.mk*, and edit it to match the configuration that would have otherwise set by running those commands. The file already contains all the variables that you need to provide; it's just a matter of uncommenting the corresponding lines and setting the values appropriately. Once you've done that, all you have to do is go to the AOSP's directory and invoke *make* directly. You can skip *envsetup.sh* and *lunch*.

Directive Definitions

Because the build system is fairly large—there are about 40 files in *build/core/* alone—there are benefits in being able to reuse as much code as possible. This is why the build system defines a large number of *directives*[§] in the *definitions.mk* file. That file is actually the largest one in the build system at about 60KB, with about 140 directives on ~1,800 lines of makefile code. Directives offer a variety of functionalities, including file lookup (e.g., `all-makefiles-under` and `all-c-files-under`), transformation (e.g., `transform-c-to-o` and `transform-java-to-classes.jar`), copying (e.g., `copy-file-to-target`) and utility (e.g., `my-dir`.)

Not only are these directives used throughout the rest of the build system's components and act as its core library, but they're sometimes also directly used in modules' *Android.mk* files. Here's an example snippet from the Calculator app's *Android.mk*:

```
LOCAL_SRC_FILES := $(call all-java-files-under, src)
```

Although thoroughly describing *definitions.mk* is outside the scope of this book, it should be fairly easy for you to explore it on your own. If nothing else, most of the directives in it are preceeded with a comment explaining what they do. For example:

§ Makefile directives are very much akin to functions in a programming language.

```
############################################################
## Find all of the java files under the named directories.
## Meant to be used like:
##    SRC_FILES := $(call all-java-files-under,src tests)
############################################################

define all-java-files-under
$(patsubst ./%,%, \
  $(shell cd $(LOCAL_PATH) ; \
          find $(1) -name "*.java" -and -not -name ".*") \
 )
endef
```

Main Make Recipes

At this point you might be wondering where any of the goodies are actually generated. How are the various images such as RAM disk generated or how is the SDK put together, for example? Well, I hope you won't hold a grudge, but I've been keeping the best for last. So without further ado, have a look at the *Makefile* in *build/core/* (not the top-level one). The file start with an innocuous-looking comment:

```
# Put some miscellaneous rules here
```

But don't be fooled. This is where some of the best meat is. Here's the snippet that takes care of generating the RAM disk for example:

```
# -------------------------------------------------------------------
# the ramdisk
INTERNAL_RAMDISK_FILES := $(filter $(TARGET_ROOT_OUT)/%, \
    $(ALL_PREBUILT) \
    $(ALL_COPIED_HEADERS) \
    $(ALL_GENERATED_SOURCES) \
    $(ALL_DEFAULT_INSTALLED_MODULES))

BUILT_RAMDISK_TARGET := $(PRODUCT_OUT)/ramdisk.img

# We just build this directly to the install location.
INSTALLED_RAMDISK_TARGET := $(BUILT_RAMDISK_TARGET)
$(INSTALLED_RAMDISK_TARGET): $(MKBOOTFS) $(INTERNAL_RAMDISK_FILES) | $(MINIGZIP)
    $(call pretty,"Target ram disk: $@")
    $(hide) $(MKBOOTFS) $(TARGET_ROOT_OUT) | $(MINIGZIP) > $@
```

And here's the snippet that creates the certs packages for checking OTA‖ updates:

```
# -------------------------------------------------------------------
```

‖ Over-The-Air

```
# Build a keystore with the authorized keys in it, used to verify the
# authenticity of downloaded OTA packages.
#
# This rule adds to ALL_DEFAULT_INSTALLED_MODULES, so it needs to come
# before the rules that use that variable to build the image.
ALL_DEFAULT_INSTALLED_MODULES += $(TARGET_OUT_ETC)/security/otacerts.zip
$(TARGET_OUT_ETC)/security/otacerts.zip: KEY_CERT_PAIR := $(DEFAULT_KEY_CERT_PAIR)
$(TARGET_OUT_ETC)/security/otacerts.zip: $(addsuffix .x509.pem,$(DEFAULT_KEY_CERT_PAIR))
    $(hide) rm -f $@
    $(hide) mkdir -p $(dir $@)
    $(hide) zip -qj $@ $<

.PHONY: otacerts
otacerts: $(TARGET_OUT_ETC)/security/otacerts.zip
```

Obviously there's a lot more than I can fit here, but have a look at *Makefile* for information on how any of the following are created:

- Properties (including the target's */default.prop* and */system/build.prop*)
- RAM disk
- Boot image (combining the RAM disk and a kernel image)
- *NOTICE* files
- OTA keystore
- Recovery image
- System image (the target's */system* directory)
- Data partition image (the target's */data* directory)
- OTA update package
- SDK

Nevertheless, some things **aren't** in this file:

Kernel images
: Don't look for any rule to build these. There is no kernel part of the AOSP. Instead, you need to find an Androidized kernel for your target, build it separately from the AOSP, and feed it to the AOSP. You can find a few examples of this in the devices in the *device/* directory. *device/samsung/crespo/*, for example, includes a kernel image (file called *kernel*) and a loadable module for the Crespo's Wifi (*bcm4329.ko* file.) Both of these are built outside the AOSP and copied in binary form into the tree for inclusion with the rest of the build.

NDK
: While the code to build the NDK is in the AOSP, it's entirely separate from the AOSP's build system in *build/*. Instead, the NDK's build system is in *ndk/build/*. We'll discuss how to build the NDK shortly.

CTS
: The rules for building the CTS are in *build/core/tasks/cts.mk*.

Cleaning

As I mentioned earlier, a *make clean* is very much the equivalent of wipping out the *out/* directory. The `clean` target itself is defined in *main.mk*. There are, however, other clean up targets. Most notably, `installclean`, which is defined in *cleanbuild.mk*, is automatically invoked whenever you change `TARGET_PRODUCT` or `TARGET_BUILD_VARIANT`. For instance, if I had first built the AOSP for the `generic-eng` combo and then used *lunch* to switch the combo to `full-eng`, the next time I start *make*, some of the build output will be automatically pruned using `installclean`:

```
$ make -j16
============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=full
TARGET_BUILD_VARIANT=eng
...
============================================
*** Build configuration changed: "generic-eng-{mdpi,nodpi}" -> "full-eng-{en_US,en_GB,
fr_FR,it_IT,de_DE,es_ES,mdpi,nodpi}"
*** Forcing "make installclean"...
*** rm -rf out/target/product/generic/data/* out/target/product/generic/data-qemu/*
out/target/product/generic/userdata-qemu.img out/host/linux-x86/obj/NOTICE_FILES
out/host/linux-x86/sdk out/target/product/generic/*.img out/target/product/generic/*.txt
out/target/product/generic/*.xlb out/target/product/generic/*.zip
out/target/product/generic/data out/target/product/generic/obj/APPS
out/target/product/generic/obj/NOTICE_FILES out/target/product/generic/obj/PACKAGING
out/target/product/generic/recovery out/target/product/generic/root
out/target/product/generic/system out/target/product/generic/dex_bootjars
out/target/product/generic/obj/JAVA_LIBRARIES
*** Done with the cleaning, now starting the real build.
```

In contrast to `clean`, `installclean` doesn't wipe out the entirety of *out/*. Instead, it only nukes the parts that need rebuilding given the combo configuration change. There's also a `clobber` target which is essentially the same thing as a `clean`.

Module Build Templates

What I just described is the build system's architecture and the mechanics of its core components. Having read that, you should have a much better idea of how Android is built from a top-down perspective. Very little of that, however, permeates down to the level of AOSP modules' *Android.mk* files. The system has in fact been architectured so that module build recipes are pretty much independent from the build system's internals. Instead, build templates are provided so that module authors can get their modules built appropriately. Each template is tailored for a specific type of module and module authors can use a set of documented variables, all prefixed by `LOCAL_`, to modulate the templates' behavior and output. Of course, the templates and underlying support files (mainly *base_rules.mk*) closely interact with the rest of the build system

to deal properly with each module's build output. But that's invisible to the module's author.

The templates are themselves found in the same location as the rest of the build system in *build/core/*. *Android.mk* gets access to them through the `include` directive. Here's an example:

```
include $(BUILD_PACKAGE)
```

As you can see, *Android.mk* files don't actually include the *.mk* templates by name. Instead, they include a variable that is set to the corresponding *.mk* file. Table 4-2 provides the full list of available module templates.

Table 4-2. Module build templates list

Variable	Template	What the template builds	Most notable use
`BUILD_EXECUTABLE`	*executable.mk*	Target binaries	Native commands and daemons
`BUILD_HOST_EXECUTABLE`	*host_executable.mk*	Host binaries	Development tools
`BUILD_RAW_EXECUTABLE`	*raw_executable.mk*	Target binaries that run on bare metal	Code in the *bootloader/* directory
`BUILD_JAVA_LIBRARY`	*java_library.mk*	Target Java libaries	Apache Harmony and Android framework
`BUILD_STATIC_JAVA_LIBRARY`	*static_java_library.mk*	Target static Java libraries	N/A, few modules use this
`BUILD_HOST_JAVA_LIBRARY`	*host_java_library.mk*	Host Java libraries	Development tools
`BUILD_SHARED_LIBRARY`	*shared_library.mk*	Target shared libraries	A vast number of modules, including many in *external/* and *frameworks/base/*
`BUILD_STATIC_LIBRARY`	*static_library.mk*	Target static libraries	A vast number of modules, including many in *external/*
`BUILD_HOST_SHARED_LIBRARY`	*host_shared_library.mk*	Host shared libraries	Development tools
`BUILD_HOST_STATIC_LIBRARY`	*host_static_library.mk*	Host static libraries	Development tools
`BUILD_RAW_STATIC_LIBRARY`	*raw_static_library.mk*	Target static libraries that run on bare metal	Code in *bootloader/*
`BUILD_PREBUILT`	*prebuilt.mk*	For copying prebuilt target files	Configuration files and binaries
`BUILD_HOST_PREBUILT`	*host_prebuilt.mk*	For copying prebuilt host files	Tools in *prebuilt/* and configuration files
`BUILD_MULTI_PREBUILT`	*multi_prebuilt.mk*	For copying prebuilt modules of	Rarely used

Variable	Template	What the template builds	Most notable use
		multiple but known type, like Java libraries or executables	
BUILD_PACKAGE	*package.mk*	Built-in AOSP apps (i.e. anything that ends up being an *.apk*	All stock AOSP apps
BUILD_KEY_CHAR_MAP	*key_char_map.mk*	Device character maps	All device character maps in AOSP

These build templates allow *Android.mk* files to be usually fairly light-weight:

```
LOCAL_PATH := $(call my-dir) ❶
include $(CLEAR_VARS) ❷

LOCAL_VARIABLE_1 := value_1 ❸

LOCAL_VARIABLE_2 := value_2

...

include $(BUILD_MODULE_TYPE) ❹
```

❶ Tells the build template where the current module is located.

❷ Clears all previously set LOCAL_* variables that might have been set for other modules.

❸ Sets various LOCAL_* variables to module-specific values.

❹ Invokes the build template that corresponds to the current module's type.

Note that CLEAR_VARS, which is provided by *clear_vars.mk*, is very important. Recall that the build system includes all *Android.mk* into what amounts to a single huge makefile. Including CLEAR_VARS ensures that the LOCAL_* values set for modules preceeding yours are zeroed out by the time your *Android.mk* is included. Also, a single *Android.mk* can describe multiple modules one after the other. Hence, CLEAR_VARS ensures that previous module recipes don't pollute subsequent ones.

Here's the Service Manager's *Android.mk* for instance (*frameworks/base/cmds/servicemanager/*):#

```
LOCAL_PATH:= $(call my-dir)
```

```
include $(CLEAR_VARS)

LOCAL_SHARED_LIBRARIES := liblog
LOCAL_SRC_FILES := service_manager.c binder.c
LOCAL_MODULE := servicemanager
ifeq ($(BOARD_USE_LVMX),true)
    LOCAL_CFLAGS += -DLVMX
endif

include $(BUILD_EXECUTABLE)
```

And here's the one from the Desk Clock app (*packages/app/DeskClock/*):*

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE_TAGS := optional
LOCAL_SRC_FILES := $(call all-java-files-under, src)
LOCAL_PACKAGE_NAME := DeskClock
LOCAL_OVERRIDES_PACKAGES := AlarmClock
LOCAL_SDK_VERSION := current

include $(BUILD_PACKAGE)

include $(call all-makefiles-under,$(LOCAL_PATH))
```

As you can see, essentially the same structure is used in both modules, even though they provide very different input and result in very different output. Notice also the last line from the Desk Clock's *Android.mk*, which basically includes all subdirectories' *Android.mk* files. As I said earlier, the build system looks for the first makefile in a hierarchy and doesn't look in any subdirectories underneath the directory where one was found, hence the need to manually invoke those. Obviously the code here just goes out and looks for all makefiles underneath. However, some parts of the AOSP either explicitly list subdirectories or conditionnally select them based on configuration.

The documentation at *http://source.android.com* used to provide an exhaustive list of all the `LOCAL_*` variables with their meaning and use. Unfortunately, at the time of this writing, this list is no longer available. The *build/core/build-system.html* file, however, contains an earlier version of that list and you should refer to that one until up-to-date lists become available again. Here are some of the most frequently-encountered `LOCAL_*` variables:

#This version is cleaned up a little (removed commented code for instance) and slightly reformatted for pretty-print.

* Also slightly modified to remove white-space and comments.

`LOCAL_PATH`
: The path of the current module's sources, typically provided by invoking `$(call my-dir)`.

`LOCAL_MODULE`
: The name to attribute to this module's build output. The actual filename or output and its location will depend on the build template you include. If this is set to `foo`, for example, and you build an executable, the final executable will be a command called *foo* and it will be put in the target's */system/bin/*. If `LOCAL_MODULE` is set to `libfoo` and you include `BUILD_SHARED_LIBRARY` instead of `BUILD_EXECUTABLE`, the build system will generate *libfoo.so* and put it in */system/lib/*.

`LOCAL_SRC_FILES`
: The source files used to build the module. You may provide those by using one the build system's defined directives, as the Desk Clock uses `all-java-files-under`, or you may list the files explicitly, as the Service Manager does.

`LOCAL_PACKAGE_NAME`
: Unlike all other modules, apps use this variable instead of `LOCAL_MODULE` to provide their names, as you can witness by comparing the two *Android.mk* shown earlier.

`LOCAL_SHARED_LIBRARIES`
: Use this to list all the libraries your module depends on. As mentioned earlier, the Service Manager depends on *liblog* instead of this variable.

`LOCAL_MODULE_TAGS`
: As I mentioned earlier, this allows you to control under which `TARGET_BUILD_VARIANT` this module is built.

`LOCAL_MODULE_PATH`
: Use this to override the default install location for the type of module you're building.

A good way to find out about more `LOCAL_*` variables is to look at existing *Android.mk* files in the AOSP. Also, *clear_vars.mk* contains the full list of variables that are cleared. So while it doesn't give you the meaning of each, it certainly lists them all.

Also, in addition to the cleaning targets that affect the AOSP globally, each module can define its own cleaning rules by providing a *CleanSpec.mk*, much like modules provide *Android.mk* files. Unlike the latter, though, the former aren't required. By default, the build system has cleaning rules for each type of module. But you can specify your own rules in a *CleanSpec.mk* in case your module's build does something the build system doesn't generate by default and, therefore, wouldn't typically know how to clean up.

Output

Now that we've looked at how the build system works and how module build templates are used by modules, let's look at the output it creates in *out/*. At a fairly high level, the

build output operates in three stages and in two modes, one for the host and one for the target:

1. *Intermediates* are generated using the module sources. These intermediates' format and location depend on the module's sources. They may be *.o* files for C/C++ code, for example, or *.jar* files for Java-based code.
2. Intermediates are used by the build system to create actual binaries and packages: taking *.o* files, for example, and linking them into an actual binary.
3. The binaries and packages are assembled together into the final output requested of the build system. Binaries, for instance, are copied into directories containing the root and */system* filesystems and images of those filesystems are generated for use on the actual device.

out/ is mainly separated into two directories, reflecting its operating modes: *host/* and *target/*. In each directory, you will find a couple of *obj/* directories that contain the various intermediates generated during the build. Most of these are stored in subdirectories named similarly to one the `BUILD_*` macros presented earlier or serve a specific complementary purpose during the build system's operation:

- *EXECUTABLES/*
- *JAVA_LIBRARIES/*
- *SHARED_LIBRARIES/*
- *STATIC_LIBRARIES/*
- *APPS/*
- *DATA/*
- *ETC/*
- *KEYCHARS/*
- *PACKAGING/*
- *NOTICE_FILES/*
- *include/*
- *lib/*

The directory you'll likely be most interested in is *out/target/product/*`PRODUCT_DEVICE`*/*. That's where the output images will be located for the `PRODUCT_DEVICE` defined in the corresponding product configuration's *.mk*. Table 4-3 explains the content of that directory.

Table 4-3. Product output

Entry	Description
android-info.txt	Contains the codename for the board for which this product is configured.
clean_steps.mk	Contains a list of steps that must be executed to clean the tree, as provided in *CleanSpec.mk* files by calling the `add-clean-step` directive.

Entry	Description
data/	The target's */data* directory.
installed-files.txt	A list of all the files installed in *data/* and *system/* directories.
obj/	The target product's intermediaries.
previous_build_config.mk	The last build target; will be used on the next *make* to check if the config has changed, thereby forcing an `installclean`.
ramdisk.img	The RAM disk image generated based on the content of the *root/* directory.
root/	The content of the target's root filesystem.
symbols/	Unstripped versions of the binaries put in the root filesystem and */system* directory.
system/	The target's */system* directory.
system.img	The */system* image, based on the content of the *system/* directory.
userdata.img	The */data* image, based on the content of the *data/* directory.

Have a look back at Chapter 2 for a refresher on the root filesytem, */system*, and */data*. Essentially, though, when the kernel boots, it will mount the RAM disk image and execute the */init* found inside. That binary, in turn, will run the */init.rc* script that will mount both the */system* and */data* images at their respective locations.

Build Recipes

With the build system's architecture and functioning in mind, let's take a look at some of the most common, and some slightly uncommon, build recipes. We'll only lightly touch on the use of the results of each recipe, often because the topic is best discussed elsewhere, but you should have enough information to get you started.

The Default droid Build

Earlier, we went through a number of plain *make* commands but never really explained the default target. When you run plain *make*, it's as if you had typed:†

```
$ make droid
```

`droid` is in fact the default target as defined in *main.mk*. You don't usually need to specify this target manually. I'm providing it here for completeness, so that you know it exists.

† This assumes you had already run *envsetup.sh* and *lunch*.

Seeing the Build Commands

When you build the AOSP, you'll notice that it doesn't actually show you the commands it's running. Instead, it only prints out a summary of each step it's at. If you want to see everything it does, like the *gcc* command lines for example, add the `showcommands` target to the command line:

```
$ make showcommands
....
host Java: apicheck (out/host/common/obj/JAVA_LIBRARIES/apicheck_intermediates/classes)
for f in ; do if [ ! -f $f ]; then echo Missing file $f; exit 1; fi; unzip -qo $f -d
out/host/common/obj/JAVA_LIBRARIES/apicheck_intermediates/classes; (cd  out/host/common
/obj/JAVA_LIBRARIES/apicheck_intermediates/classes && rm -rf META-INF); done
javac -J-Xmx512M -target 1.5 -Xmaxerrs 9999999 -encoding ascii -g    -extdirs "" -d
out/host/common/obj/JAVA_LIBRARIES/apicheck_intermediates/classes \@out/host/common/obj
/JAVA_LIBRARIES/apicheck_intermediates/java-source-list-uniq || ( rm -rf out/host/common
/obj/JAVA_LIBRARIES/apicheck_intermediates/classes ; exit 41 )
rm -f out/host/common/obj/JAVA_LIBRARIES/apicheck_intermediates/java-source-list
rm -f out/host/common/obj/JAVA_LIBRARIES/apicheck_intermediates/java-source-list-uniq
jar -cfm out/host/common/obj/JAVA_LIBRARIES/apicheck_intermediates/javalib.jar build
/tools/apicheck/src/MANIFEST.mf  -C out/host/common/obj/JAVA_LIBRARIES/apicheck_intermediates
/classes .
Header: out/host/linux-x86/obj/include/libexpat/expat.h
cp -f external/expat/lib/expat.h out/host/linux-x86/obj/include/libexpat/expat.h
Header: out/host/linux-x86/obj/include/libexpat/expat_external.h
cp -f external/expat/lib/expat_external.h out/host/linux-x86/obj/include/libexpat/expat_external.h
Header: out/target/product/generic/obj/include/libexpat/expat.h
cp -f external/expat/lib/expat.h out/target/product/generic/obj/include/libexpat/expat.h
...
```

Which, to illustrate what I just explained in the previous section, is also the same as:

```
$ make droid showcommands
```

As you'll rapidly notice when using this, it generates a lot of output and is therefore hard to follow. You may, however, want to save the standard output and standard error into files if you'd like to analyze the actual commands used to build the AOSP:

```
$ make showcommands > aosp-build-stdout 2> aosp-build-stderr
```

Building the SDK for Linux and MacOS

The official Android SDK is available at *http://developer.android.com*. You can, however, build your own SDK using the AOSP if, for instance, you extended the core APIs to expose new functionality and would like to distribute the result to developers so they can benefit from your new APIs. To do so, you'll need to select a special combo:

```
$ . build/envsetup.sh
$ lunch sdk-eng
$ make sdk
```

Once this is done, the SDK will be in *out/host/linux-x86/sdk/* when built on Linux and *out/host/darwin-x86/sdk/* when built on a Mac. There will be two copies, one a zip file, much like the one distributed at *http://developer.android.com*, and one uncompressed and ready to use.

Assuming you had already configured Eclipse for Android development using the instructions at *http://developer.android.com*, you'll need to carry out two additional steps to use your newly-built SDK. First, you'll need to tell Eclipse the location of the new SDK. To do so, go to Window→Preferences→Android, enter the path to the new SDK in the "SDK Location" box, and click OK. Also, for reasons that aren't entirely clear to the author at the time of this writing, you also need to go to Window→"Android SDK and AVD Manager"→"Installed Packages" and click on "Update All..." That will display a wizard. Reject all the items selected except the first one, "Android SDK Tools, revision *api_level*", and click on "Install." Once that is done, you'll be able to create new projects using the new SDK and access any new APIs you expose in it. If you don't do that second step, you'll be able to create new Android projects, but none of them will resolve Java libraries properly and will, therefore, never build.

Building the SDK for Windows

The instructions for building the SDK for Windows are slightly different from Linux and MacOS:

```
$ . build/envsetup.sh
$ lunch sdk-eng
$ make win_sdk
```

The resulting output will be in *out/host/windows/sdk/*.

Building the CTS

If you want to build the CTS, you don't need to use *envsetup.sh* or *lunch*. You can go right ahead and type:

```
$ make cts
...
Generating test description for package android.sax
Generating test description for package android.performance
Generating test description for package android.graphics
Generating test description for package android.database
Generating test description for package android.text
```

```
Generating test description for package android.webkit
Generating test description for package android.gesture
Generating test plan CTS
Generating test plan Android
Generating test plan Java
Generating test plan VM
Generating test plan Signature
Generating test plan RefApp
Generating test plan Performance
Generating test plan AppSecurity
Package CTS: out/host/linux-x86/cts/android-cts.zip
Install: out/host/linux-x86/bin/adb
```

The *cts* commands includes its own online help:

```
$ cd out/host/linux-x86/bin/
$ ./cts
Listening for transport dt_socket at address: 1337
Android CTS version 2.3_r3
$ cts_host > help
Usage: command options
Avaiable commands and options:
  Host:
    help: show this message
    exit: exit cts command line
  Plan:
    ls --plan: list available plans
    ls --plan plan_name: list contents of the plan with specified name
    add --plan plan_name: add a new plan with specified name
    add --derivedplan plan_name -s/--session session_id -r/--result result_type: derive
    a plan from the given session
    rm --plan plan_name/all: remove a plan or all plans from repository
    start --plan test_plan_name: run a test plan
    start --plan test_plan_name -d/--device device_ID: run a test plan using the specified device
    start --plan test_plan_name -t/--test test_name: run a specific test
...
$ cts_host > ls --plan
List of plans (8 in total):
Signature
RefApp
VM
Performance
AppSecurity
Android
Java
CTS
```

Once you have a target up and running, such as the emulator for instance, you can launch the test suite and it will use *adb* to run tests on the target:

```
$ ./cts start --plan CTS
Listening for transport dt_socket at address: 1337
```

```
Android CTS version 2.3_r3
Device(emulator-5554) connected
cts_host > start test plan CTS

CTS_INFO >>> Checking API...

CTS_INFO >>> This might take several minutes, please be patient...
...
```

Building the NDK

As I had mentioned earlier, the NDK has its own separate build system, with its own setup and help system, which you can invoke like this:

```
$ cd ndk/build/tools
$ export ANDROID_NDK_ROOT=aosp-root/ndk
$ ./make-release --help
Usage: make-release.sh [options]

Valid options (defaults are in brackets):

  --help                        Print this help.
  --verbose                     Enable verbose mode.
  --release=name               Specify release name [20110921]
  --prefix=name                Specify package prefix [android-ndk]
  --development=path           Path to development/ndk directory [/home/karim/opersys-dev/
android/aosp-2.3.4/development/ndk]
  --out-dir=path               Path to output directory [/tmp/ndk-release]
  --force                       Force build (do not ask initial question) [no]
  --incremental                 Enable incremental packaging (debug only). [no]
  --darwin-ssh=hostname        Specify Darwin hostname to ssh to for the build.
  --systems=list               List of host systems to build for [linux-x86]
  --toolchain-src-dir=path     Use toolchain sources from path
```

When you are ready to build the NDK, you can invoke *make-release* as follows, and witness its rather emphatic warning:

```
$ ./make-release
IMPORTANT WARNING !!

This script is used to generate an NDK release package from scratch
for the following host platforms: linux-x86

This process is EXTREMELY LONG and may take SEVERAL HOURS on a dual-core
machine. If you plan to do that often, please read docs/DEVELOPMENT.TXT
that provides instructions on how to do that more easily.

Are you sure you want to do that [y/N]
y
Downloading toolchain sources...
Using git clone prefix: git://android.git.kernel.org/toolchain
```

```
downloading sources for toolchain/binutils
...
```

Updating the API

The build systems has safeguards in case you modify the AOSP's core API. If you do, the build will fail by default with a warning such as this:

```
******************************
You have tried to change the API from what has been previously approved.

To make these errors go away, you have two choices:
   1) You can add "@hide" javadoc comments to the methods, etc. listed in the
      errors above.

   2) You can update current.xml by executing the following command:
         make update-api

      To submit the revised current.xml to the main Android repository,
      you will need approval.
******************************


make: *** [out/target/common/obj/PACKAGING/checkapi-current-timestamp] Error 38
make: *** Waiting for unfinished jobs....
```

As the error message suggests, to get the build to continue, you'll need to do something like this:

```
$ make update-api
...
Install: out/host/linux-x86/framework/apicheck.jar
Install: out/host/linux-x86/framework/clearsilver.jar
Install: out/host/linux-x86/framework/droiddoc.jar
Install: out/host/linux-x86/lib/libneo_util.so
Install: out/host/linux-x86/lib/libneo_cs.so
Install: out/host/linux-x86/lib/libneo_cgi.so
Install: out/host/linux-x86/lib/libclearsilver-jni.so
Copying: out/target/common/obj/JAVA_LIBRARIES/core_intermediates/emma_out/lib
           /classes-jarjar.jar
Install: out/host/linux-x86/framework/dx.jar
Install: out/host/linux-x86/bin/dx
Install: out/host/linux-x86/bin/aapt
Copying: out/target/common/obj/JAVA_LIBRARIES/bouncycastle_intermediates
           /emma_out/lib/classes-jarjar.jar
Copying: out/target/common/obj/JAVA_LIBRARIES/ext_intermediates/emma_out/lib
           /classes-jarjar.jar
Install: out/host/linux-x86/bin/aidl
Copying: out/target/common/obj/JAVA_LIBRARIES/core-junit_intermediates/emma_out
           /lib/classes-jarjar.jar
Copying: out/target/common/obj/JAVA_LIBRARIES/framework_intermediates/emma_out
```

```
           /lib/classes-jarjar.jar
Copying current.xml
```

The next time you start *make*, you won't get any more errors regarding API changes.

Building a Single Module

Up to now, we've looked at building the entire tree. You can also build individual modules. Here's how you can ask the build system to build the Launcher2 module (i.e., the Home screen):

```
$ make Launcher2
```

You can also clean modules individually:

```
$ make clean-Launcher2
```

If you'd like to force the build system to regenerate the system image to include your updated module, you can add the `snod` target to the command line:

```
$ make Launcher2 snod
============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=generic
...
target Package: Launcher2 (out/target/product/generic/obj/APPS/Launcher2_intermediates/package.apk)
 'out/target/common/obj/APPS/Launcher2_intermediates//classes.dex' as 'classes.dex'...
Install: out/target/product/generic/system/app/Launcher2.apk
Install: out/host/linux-x86/bin/mkyaffs2image
make snod: ignoring dependencies
Target system fs image: out/target/product/generic/system.img
```

Building Out of Tree

If ever you'd like to build code against the AOSP and its Bionic library but don't want to incorporate that into the AOSP, you can use a makefile such as the following to get the job done:‡

```
# Paths and settings
TARGET_PRODUCT = generic
ANDROID_ROOT   = /home/karim/android/aosp-2.3.x
```

‡ This makefile is inspired by a blog post by Row Boat developer Amit Pundir and is based on the example makefile provided in Chapter 4 of *Building Embedded Linux Systems, 2nd ed.* (O'Reilly).

```
BIONIC_LIBC    = $(ANDROID_ROOT)/bionic/libc
PRODUCT_OUT    = $(ANDROID_ROOT)/out/target/product/$(TARGET_PRODUCT)
CROSS_COMPILE  = \
    $(ANDROID_ROOT)/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin/arm-eabi-

# Tool names
AS            = $(CROSS_COMPILE)as
AR            = $(CROSS_COMPILE)ar
CC            = $(CROSS_COMPILE)gcc
CPP           = $(CC) -E
LD            = $(CROSS_COMPILE)ld
NM            = $(CROSS_COMPILE)nm
OBJCOPY       = $(CROSS_COMPILE)objcopy
OBJDUMP       = $(CROSS_COMPILE)objdump
RANLIB        = $(CROSS_COMPILE)ranlib
READELF       = $(CROSS_COMPILE)readelf
SIZE          = $(CROSS_COMPILE)size
STRINGS       = $(CROSS_COMPILE)strings
STRIP         = $(CROSS_COMPILE)strip

export AS AR CC CPP LD NM OBJCOPY OBJDUMP RANLIB READELF \
         SIZE STRINGS STRIP

# Build settings
CFLAGS        = -O2 -Wall -fno-short-enums
HEADER_OPS    = -I$(BIONIC_LIBC)/arch-arm/include \
                -I$(BIONIC_LIBC)/kernel/common \
                -I$(BIONIC_LIBC)/kernel/arch-arm
LDFLAGS       = -nostdlib -Wl,-dynamic-linker,/system/bin/linker \
                $(PRODUCT_OUT)/obj/lib/crtbegin_dynamic.o \
                $(PRODUCT_OUT)/obj/lib/crtend_android.o \
                -L$(PRODUCT_OUT)/obj/lib -lc -ldl

# Installation variables
EXEC_NAME     = example-app
INSTALL       = install
INSTALL_DIR   = $(PRODUCT_OUT)/system/bin

# Files needed for the build
OBJS          = example-app.o

# Make rules
all: example-app

.c.o:
        $(CC) $(CFLAGS) $(HEADER_OPS) -c $<

example-app: ${OBJS}
        $(CC) -o $(EXEC_NAME) ${OBJS} $(LDFLAGS)

install: example-app
        test -d $(INSTALL_DIR) || $(INSTALL) -d -m 755 $(INSTALL_DIR)
        $(INSTALL) -m 755 $(EXEC_NAME) $(INSTALL_DIR)

clean:
```

```
	rm -f *.o $(EXEC_NAME) core

distclean:
	rm -f *~
	rm -f *.o $(EXEC_NAME) core
```

In this case, you don't need to care about either *envsetup.sh* or *lunch*. You can just go ahead and type the magic incatation:

```
$ make
```

Obviously this won't add your binary to any of the images generated by the AOSP. Even the `install` target here will be of value only if you're mounting the target's filesystem off NFS; and that's valuable only during debugging, which is what this makefile is assumed to be useful for. To an extent, it could also be argued that using such a makefile is actually counter-productive since it's far more complicated than the equivalent *Android.mk* had this code been added as a module part of the AOSP.

Still, this kind of hack can have its uses. Under certain circumstances, for instance, it might make sense to modify the conventional build system used by a rather large code base to build that project against the AOSP yet outside of it; the alternative being to copy the project into the AOSP and create *Android.mk* files to reproduce the mechanics of its original conventional build system, which might turn out to be a substantial endeavour in and of itself.

Basic AOSP Hacks

You most likely bought this book with one thing in mind: to hack the AOSP to fit your needs. Over the next few pages, we'll start looking into some of the most obvious hacks you'll likely want to try. Of course we're only setting the stage here with the parts that pertain to the build system, which is where you'll likely want to start anyway. The next chapters will allow to push what we see here much further.

Adding an App

If you would like to add a default app in addition to the stock ones, you'll need to start by creating a directory for it in *packages/apps/*. As a starter, try creating a "HelloWorld!" app with Eclipse and the default SDK; by default all new Android projects in Eclipse are a "HelloWorld!". Then copy that app from the Eclipse workspace to its destination:

```
$ cp -a ~/workspace/HelloWorld ~/android/aosp-2.3.x/packages/apps/
```

You'll then have to create an *Android.mk* in *aosp-rootpackages/apps/HelloWorld* to build that app:

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE_TAGS := optional
LOCAL_SRC_FILES := $(call all-java-files-under, src)
LOCAL_PACKAGE_NAME := HelloWorld

include $(BUILD_PACKAGE)
```

Given that we're tagging this module as `optional`, it won't be included by default in the AOSP build. To get it to be included, you'll need to add it to the default `PRODUCT_PACKAGES` listed in *aosp-root/build/target/product/core.mk*.

Note that the commands I've shown so far means you're adding the app globally to **all** products. That might not be what you're looking for, though. If you want to add the app to just your product, which you likely should if it's going to be available only on your device, you should add the app into your product's entry in *device/* instead of *packages/apps/*. We'll cover how to add your own device shortly.

Adding a Native Tool or Daemon

There are a number of locations in the tree where native tools and daemons are located. Here are the most important ones:

system/core/ and system/
: Custom Android binaries that are meant to be used outside the Android framework or are stand-alone pieces.

frameworks/base/cmds/
: Binaries that are tightly coupled to the Android framework. This is where the Service Manager and *installd* are found, for example.

external/
: Binaries that are generated by an external project that is imported into the AOSP. *strace*, for instance, is here.

Now that you most likely know where the code generating the binary should go, you'll also need to provide an *Android.mk* in the directory containing the code to build that module:

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE := hello-world
LOCAL_MODULE_TAGS := optional
LOCAL_SRC_FILES := hello-world.cpp
LOCAL_SHARED_LIBRARIES := liblog
```

```
include $(BUILD_EXECUTABLE)
```

Again, you'll also need to make sure `hello-world` is also part of the default `PRODUCT_PACKAGES` listed in *aosp-root/build/target/product/core.mk*. And again, what you'd be doing here is adding that binary to all products. So, much like a custom app, the best location for your binary may actually be in your product-specific directory in *device/*.

Adding a Native Library

Like binaries, libraries are typically found in a number of locations in the tree. Unlike binaries, though, a lot of libraries are used within a single module but nowhere else. Hence, these libraries will typically be placed within that module's code and not in the typical locations where libraries used system-wide are found. The latter are typically in:

system/core
: Libraries used by many parts of the system, including some outside the Android framework. This is where liblog is, for instance.

frameworks/base/libs/
: Libraries intimately tied to the framework. This is where libbinder is.

external/
: Libraries generated by external projects imported in the AOSP. OpenSSL's libssl is here.

Whether your library best belongs in one of these locations, within another module, or in your product's *device/* entry, you'll need an *Android.mk* to build it:

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE := libmylib
LOCAL_MODULE_TAGS := optional
LOCAL_PRELINK_MODULE := false
LOCAL_SRC_FILES := $(call all-c-files-under,.)

include $(BUILD_SHARED_LIBRARY)
```

Library Prelinking

To reduce the time it takes to load libraries, Android *prelinks* most of its libraries. This is done by specifying ahead of time the address location where the library will be loaded instead of letting it be figured out at run time. The file where the addresses are specifies is *build/core/prelink-linux-arm.map* and the tool that does the mapping is called *apriori*. It contains entries such as these:

```
# core system libraries
libdl.so                0xAFF00000 # [<64K]
```

```
libc.so                 0xAFD00000 # [~2M]
libstdc++.so            0xAFC00000 # [<64K]
libm.so                 0xAFB00000 # [~1M]
liblog.so               0xAFA00000 # [<64K]
libcutils.so            0xAF900000 # [~1M]
libthread_db.so         0xAF800000 # [<64K]
libz.so                 0xAF700000 # [~1M]
libevent.so             0xAF600000 # [???]
libssl.so               0xAF400000 # [~2M]
...
# assorted system libraries
libsqlite.so            0xA8B00000 # [~2M]
libexpat.so             0xA8A00000 # [~1M]
libwebcore.so           0xA8300000 # [~7M]
libbinder.so            0xA8200000 # [~1M]
libutils.so             0xA8100000 # [~1M]
libcameraservice.so     0xA8000000 # [~1M]
libhardware.so          0xA7F00000 # [<64K]
libhardware_legacy.so   0xA7E00000 # [~1M]
...
```

If you want to add a custom native library, you need either to add it to the list of libraries in *prelink-linux-arm.map* or to set the `LOCAL_PRELINK_MODULE` to `false`. The build will fail if you forget to do one of these.

To use this library, you must add it to the list of libraries listed by the *Android.mk* file of whichever binaries depend on it:

```
LOCAL_SHARED_LIBRARIES := libmylib
```

You'll also likely need to add relevant headers to an *include/* directory located in about the same location as you put your library so that the code that need to link against your library can find said headers, such as *system/core/include/* or *frameworks/base/include/*

Adding a Device

Adding a custom device is most likely one of the topmost items (if not the topmost) on your list if you're reading this book. You'll likely therefore want to bookmark this section, as I'm about to show you how to just that. Of course I'm actually only showing you the build aspects of the work. There are a lot more steps involved in porting Android to new hardware as we'll see through the rest of the book. Still, adding the new device to the build system will definitely be one of the first things you do. Fortunately, doing that is relatively straight-forward.

For the benefit of the current exercise, assume you work for a company called ACME and that you're tasked with delivering its latest gizmo: the CoyotePad, intended to be the best platform for playing all bird games. Let's get started by creating an entry for our new device in *device/*:

```
$ cd ~/android/aosp-2.3.x
$ . build/envsetup.sh
$ mkdir -p device/acme/coyotepad
$ cd device/acme/coyotepad
```

The first thing we'll need in here is an *AndroidProducts.mk* file to describe the various AOSP products that could be built for the CoyotePad:

```
PRODUCT_MAKEFILES := \
    $(LOCAL_DIR)/full_coyotepad.mk
```

While we could've had several products,§ the typical case is to just have one as in this case, and it's described in *full_coyotepad.mk*:

```
$(call inherit-product, $(SRC_TARGET_DIR)/product/languages_full.mk)
$(call inherit-product, $(SRC_TARGET_DIR)/product/full.mk)

DEVICE_PACKAGE_OVERLAYS :=

PRODUCT_PACKAGES +=
PRODUCT_COPY_FILES +=

PRODUCT_NAME := full_coyotepad
PRODUCT_DEVICE := coyotepad
PRODUCT_MODEL := Full Android on CoyotePad, meep-meep
```

It's worth taking a closer look at this makefile. First, we're using the `inherit-product` directive to tell the build system to pull in other product descriptions as the basis of ours. This allows us to build on other people's work and not have to specify from scratch every bit and piece of the AOSP that we'd like to include. *languages_full.mk* will pull in a vast number of locales and *full.mk* will make sure we get the same set of modules as if we had built using the `full-eng` combo.

With regard to the other variables:

`DEVICE_PACKAGE_OVERLAYS`
: Allows us to specify a directory which will form the basis of an overlay that will be applied onto the AOSP's sources, thereby allowing us to substitute default package resources with device-specific resources. You'll find this useful if you'd like to set custom layouts or colors for Launcher2 or other apps, for instance. We'll look at how to use this in the next section.

§ See *build/target/product/AndroidProducts.mk* for an example

PRODUCT_PACKAGES[‖]
: Allows us to specify packages we'd like to have this product include in addition to those specified in the products we're already inheriting from. If you have custom apps, binaries, or libraries located within *device/acme/coyotepad/*, for instance, you'll want to add them here so that they get included in the final images generated.

PRODUCT_COPY_FILES
: Allows us to list specific files we'd like to see copied to the target's filesystem and the location where they need to be copied. Each source/destination pair is colon-separated and pairs are space-separated amongst themselves. This is useful for configuration files and prebuilt binaries such as firmware images or kernel modules.

PRODUCT_NAME
: The TARGET_PRODUCT, which you can set either by selecting a *lunch* combo or passing it as a part of the combo parameter to *lunch* as in:

```
$ lunch full_coyotepad-eng
```

PRODUCT_DEVICE
: The name of the actual finished product shipped to the customer. TARGET_DEVICE derives from this variable. PRODUCT_DEVICE has to match an entry in *device/acme/*, since that's where the build looks for the corresponding *BoardConfig.mk*. In this case, the variable is the same as the name of the directory we're already in.

PRODUCT_MODEL
: The name of this product as provided in the "Model number" in the "About the phone" section of the settings. This variable actually gets stored as the `ro.product.model` global property accessible on the device.

Now that we've described the product, we must also provide some information regarding the board the device is using through a *BoardConfig.mk* file:

```
TARGET_NO_KERNEL := true
TARGET_NO_BOOTLOADER := true
TARGET_CPU_ABI := armeabi
BOARD_USES_GENERIC_AUDIO := true

USE_CAMERA_STUB := true
```

This is a very skinny *BoardConfig.mk* and only ensures that we actually build successfully. For a real-life version of that file, have a look at *device/samsung/crespo/BoardConfigCommon.mk*.

‖ Notice the use of the += sign. It allows us to append to the existing values in the variable instead of subtituting its content.

You'll also need to provide a conventional *Android.mk* in order to build all the modules that you might have included in this device's directory:

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

ifneq ($(filter coyotepad,$(TARGET_DEVICE)),)
include $(call all-makefiles-under,$(LOCAL_PATH))
endif
```

It's in fact the preferred modus operandi to put all device-specific apps, binaries, and libraries within the device's directory instead of globally within the rest of the AOSP as was shown earlier. If you do add modules here, don't forget to also add them to `PRODUCT_PACKAGES` as I explained earlier. If you just put them here and provide them valid *Android.mk* files, they'll build, but they won't be in the final images.

Lastly, let's close the loop by making the device we just added visible to *envsetup.sh* and *lunch*. To do so, you'll need to add a *vendorsetup.sh* in your device's directory:

```
add_lunch_combo full_coyotepad-eng
```

You also need to make sure that it's executable if it's to be operational:

```
$ chmod 755 vendorsetup.sh
```

We can now go back to the AOSP's root and take our brand new ACME CoyotePad for a ~~*run*~~chase:

```
$ croot
$ . build/envsetup.sh
$ lunch

You're building on Linux

Lunch menu... pick a combo:
     1. generic-eng
     2. simulator
     3. full_coyotepad-eng
     4. full_passion-userdebug
     5. full_crespo4g-userdebug
     6. full_crespo-userdebug

Which would you like? [generic-eng] 3

============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=full_coyotepad
TARGET_BUILD_VARIANT=eng
```

```
TARGET_SIMULATOR=false
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=GINGERBREAD
============================================

$ make -j16
```

As you can see, the AOSP now recognizes our new device and prints the information correspondingly. When the build is done, we'll also have the same type of output provided in any other AOSP build, except that it will be a product-specific directory:

```
$ ls -al out/target/product/coyotepad/
total 89356
drwxr-xr-x  7 karim karim     4096 2011-09-21 19:20 .
drwxr-xr-x  4 karim karim     4096 2011-09-21 19:08 ..
-rw-r--r--  1 karim karim        7 2011-09-21 19:10 android-info.txt
-rw-r--r--  1 karim karim     4021 2011-09-21 19:41 clean_steps.mk
drwxr-xr-x  3 karim karim     4096 2011-09-21 19:11 data
-rw-r--r--  1 karim karim    20366 2011-09-21 19:20 installed-files.txt
drwxr-xr-x 14 karim karim     4096 2011-09-21 19:20 obj
-rw-r--r--  1 karim karim      327 2011-09-21 19:41 previous_build_config.mk
-rw-r--r--  1 karim karim  2649750 2011-09-21 19:43 ramdisk.img
drwxr-xr-x 11 karim karim     4096 2011-09-21 19:43 root
drwxr-xr-x  5 karim karim     4096 2011-09-21 19:19 symbols
drwxr-xr-x 12 karim karim     4096 2011-09-21 19:19 system
-rw-------  1 karim karim 87280512 2011-09-21 19:20 system.img
-rw-------  1 karim karim  1505856 2011-09-21 19:14 userdata.img
```

Also, have a look at the *build.prop* file in *system/*. It contains various global properties that will be available at runtime on the target and that relate to our configuration and build:

```
# begin build properties
# autogenerated by buildinfo.sh
ro.build.id=GINGERBREAD
ro.build.display.id=full_coyotepad-eng 2.3.4 GINGERBREAD eng.karim.20110921.190849 test-keys
ro.build.version.incremental=eng.karim.20110921.190849
ro.build.version.sdk=10
ro.build.version.codename=REL
ro.build.version.release=2.3.4
ro.build.date=Wed Sep 21 19:10:04 EDT 2011
ro.build.date.utc=1316646604
ro.build.type=eng
ro.build.user=karim
ro.build.host=w520
ro.build.tags=test-keys
```

```
ro.product.model=Full Android on CoyotePad, meep-meep
ro.product.brand=generic
ro.product.name=full_coyotepad
ro.product.device=coyotepad
ro.product.board=
ro.product.cpu.abi=armeabi
ro.product.manufacturer=unknown
ro.product.locale.language=en
ro.product.locale.region=US
ro.wifi.channels=
ro.board.platform=
# ro.build.product is obsolete; use ro.product.device
ro.build.product=coyotepad
# Do not try to parse ro.build.description or .fingerprint
ro.build.description=full_coyotepad-eng 2.3.4 GINGERBREAD eng.karim.20110921.190849 test-keys
ro.build.fingerprint=generic/full_coyotepad/coyotepad:2.3.4/GINGERBREAD
                              /eng.karim.20110921.190849:eng/test-keys
# end build properties
...
```

As you can imagine, there's a lot more to be done here to make sure the AOSP runs on our hardware. But the preceding steps give us the starting point.

Adding an App Overlay

Overlays are a mechanism included in the AOSP to allow device manufacturers to change the resources provided (such as for apps), without actually modifying the original resources included in the AOSP. To use this capability you must create an overlay tree and tell the build system about it. The easiest location for an overlay is within a device-specific directory such as the one we just created in the previous section:

```
$ cd device/acme/coyotepad/
$ mkdir overlay
```

To tell the build system to take this overlay into account, we need to modify our *full_coyotepad.mk* such that:

```
DEVICE_PACKAGE_OVERLAYS := device/acme/coyotepad/overlay
```

At this point, though, our overlay isn't doing much. Let's say we want to modify the Launcher2's default strings. We could then do something like this:

```
$ mkdir -p overlay/packages/apps/Launcher2/res/values
$ cp aosp-root/packages/apps/Launcher2/res/values/strings.xml \
> overlay/packages/apps/Launcher2/res/values/
```

You are then free to modify your copy of *strings.xml* to suite your needs. Your device will have a Launcher2 that has your custom strings, but the default Launcher2 will still have its original strings. So if someone relies on the same AOSP sources you're using to build for another product, they'll still get the original strings. You can, of course, replace most resources like this, including images and XML files. So long as you put the files in the same hierarchy as they are found in the AOSP but within *device/acme/coyotepad/overlay/*, they'll be taken into account by the build system.