# 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 you'll get a high-level view of Android internals in this chapter, and have the opportunity in later chapters to dig into parts of internals in greater detail, including tying said internals to the actual AOSP sources.

As mentioned in the Preface, this book is mainly based on 2.3.x/Gingerbread. That said, Android's internals had remained fairly stable over its lifetime up to that version of Android, and they've changed very little from that version to the current 4.2/Jelly Bean. Still, while the bulk of the internals remains relatively unchanged, 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 independently from it.[1]

## 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

1. Some speculate that this change was triggered because some app developers were doing too many fancy tricks with notification that were having 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.

working on embedding Android on a device, you might not have to actually deal directly with the idiosyncrasies of Android's app development API, but some of your colleagues might. If nothing else, you might as well share a common lingo with 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 users' lives better, but they can sometimes be challenging to deal with.

### Components

Android applications consist of loosely tied *components*. Components of one app can invoke or 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 main 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 on top of one another in the same way as a browser remembers web pages 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 have a Back button, whether it be a physical button on the device or a soft button displayed onscreen, 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 the device). 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 several others and end up creating a stack of activities all related to the original one they launched; this stack of activities is called

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 to 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. If you are familiar with Qt, you can think of an intent as similar to, though not entirely the same as, a Qt signal. The intent itself is a passive object. The effects of its dispatching will depend on its content, the mechanism used to dispatch it, the system's built-in rules, and the set of installed apps. 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 be received only by a service, not by 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. This is typically called an "implicit" intent. An intent can also be sent to a specific component in an "explicit" fashion, 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 tenet of Android is that the user shouldn't have to manage task switching. While there are a number of ways to switch among tasks, including a built-in mechanism that's typically accessed with a long press on the Home button, as well as a number of task manager apps available for Android, the user experience doesn't rely on those. Instead, the user is expected to start as many apps as he wants and "switch" among them by clicking Home to go to the home screen and clicking 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 corollary to, or consequence of, this design decision is that apps gradually use up more and more system resources as they are started, a process that can't go on forever. At some point, the system will have to start reclaiming the resources of the least recently used or nonpriority components in order to make way for newly activated components. Still, this resource recycling should be entirely transparent to the user. In other words, when a component is taken down to make way for a 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 had been waiting in memory all along.

To make this behavior possible, Android defines a standard *lifecycle* for each component type. An app developer must manage her components' lifecycle by implementing a series of callbacks for each component. These callbacks are then 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. Google uses a state diagram to explain the activity's lifecycle to app developers.

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 topmost 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 by another app, a process will be started to house that app's components. And unless the app developer does anything to override 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 contained 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 shouldn't see any adverse behavior. For all practical purposes, in fact, the user shouldn't even notice that the process housing the app's components went away and got re-created "automagically" later.

### Remote procedure calls (RPCs)

Much like many other components of the system, Android defines its own RPC/IPC (remote procedure call/inter-process communication) mechanism: *Binder*. So communication across components is not typically done using the usual socket or System V IPC. 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 marshaling/unmarshaling 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 straightforward, 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 `ViewGroups`. An activity's UI can actually be specified either 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 allow developers to store key-value pairs 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, so they are inaccessible to other apps, or they can be made readable and/or writable by other apps. App developers can also use the SQLite functionality included in Android to manage their own private databases. 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 group identifier (GID). Essentially, it's as if every app is a separate "user" in the system. And as in any multiuser Unix system, these "users" cannot access one another's resources unless permissions are explicitly 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 (universal resource identifier), 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 through Google Play. 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 *http://developer.android.com* for more information on 2D and 3D graphics, multimedia, location and maps, Bluetooth, NFC, etc.

## 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) plug-in, and the QEMU-based emulator in the SDK—allows developers to do the vast majority of development work straight from their workstations. Developers will also usually want to test their apps on real devices prior to making them available through Google Play, as there are usually runtime behavior differences between the emulator and actual devices. Some software publishers take this to the extreme and test their apps on several dozen devices before shipping a new release.

### Testing on Several Hundred Devices

Obviously, app developers can't be expected to have every possible device at their disposal for testing. A few companies have therefore sprung up to allow app developers to test their apps on several hundred devices by simply uploading their apps to these companies' websites.

These companies typically have a web interface allowing developers to submit their app for execution on their device farm. Developers are then given detailed reports about failures and sometimes fairly explicit output from the failed devices' logs. Have a look at Apkudo, Bitbar's Testdroid products, and LessPainful if you need such functionality.

Interestingly, Apkudo also provides a service to allow you to test devices prior to their release by running several hundred popular apps on the device to ensure that the AOSP it runs performs correctly.

Even if you don't plan to develop any apps for your embedded system, I highly suggest you set up the development environment 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 to extend the AOSP's API and create and distribute your own custom SDK.

To set up an app development environment, follow the instructions provided by Google for the SDK, or have a look at the book *Learning Android* by Marko Gargenta (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 natively compiled code. To this end, Google has made the Native Development Kit (NDK) available. As advertised, this is mostly aimed at game developers needing to squeeze every last bit of performance out of the device their game is running on. As such, the APIs made available in the NDK are mostly geared toward 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 parts 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, presently no API allowing you to send an intent from within C code compiled with the NDK; the SDK must be used to do it in Java instead. Again, the APIs made available through the NDK are mostly geared toward 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 the limitations and requirements that apply to Java app developers. So, as an embedded developer, remember that the NDK is useful for app developers to run native 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 I will often refer back to it, if not explicitly then implicitly. 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.

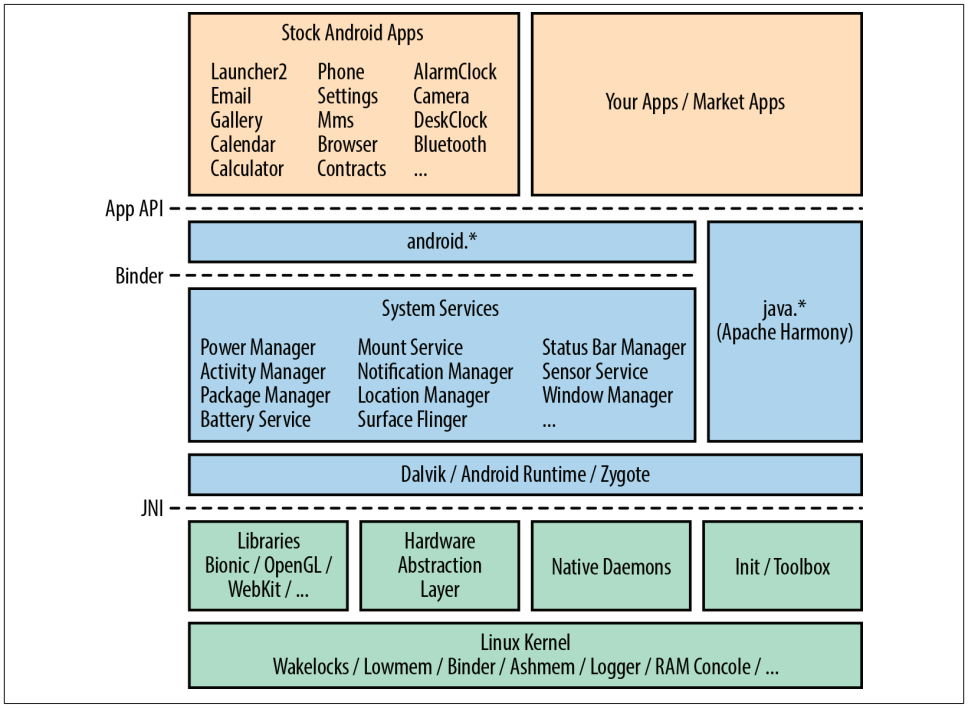



*Figure 2-1. Android's architecture*

If you are familiar with some form of Linux development, then 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, no bash shell, 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 regard 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 in Appendix A.

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 it omits key information that must be understood by embedded developers. For instance, Google's diagram and developer documentation 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 internals and significance of the System Server are rarely if at all explained 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 centerpiece 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 before distributing it to their users. Android, as such, is no different in that the Android developers patch the "vanilla" kernel to meet their needs.

Historically, Android differed from standard practice, however, in relying on several custom functionalities that were significantly different from what was 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 on the rest of the distribution's components, Android's user-space components would simply not work unless they were running on an "Androidized" kernel. As I mentioned in the previous chapter, Android kernels were, up until recently, major forks from the mainline kernel. As I also mentioned, the situation has since progressed a lot, and many of the features required to run Android are finding their way into the mainline kernel.

Hopefully things will have progressed enough by the time you read this that you can just grab a kernel straight from *http://kernel.org* and run the AOSP on top of it. However, if past is prelude and the history of embedded Linux is an indication of what's to come, then your best source for getting a proper, Android-compatible kernel to run on your hardware is likely going to be the vendor of the SoC you're using.

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.* (Addison-Wesley Professional, 2010) 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. Androidized kernels typically contain several hundred patches over the standard kernel, often to provide device-specific functionality, fixes, and enhancements. You can use *git*[2] to do an exhaustive analysis of the commit deltas between one of the kernels at *http://android.googlesource.com* and the mainline kernel it was forked from. Also, note that some of the functionality in some Androidized kernels, such as the PMEM driver, is device-specific and isn't necessarily used in all Android devices.

### Creating Your Own Androidized Kernel

If you'd like to know how to create Androidized kernels from scratch or if you're tasked with this, say because you work for an SoC vendor, have a look at the Androidization of linux kernel blog post by Linaro engineer Vishal Bhoj, published in March 2012. In this post, Vishal explains how to create an Androidized kernel using the *git rebase* command. For more information about that specific command, have a look at the corresponding online git documentation.

Incidentally, Linaro, whose role is to assist its members with platform enablement, maintains an Androidized kernel that closely follows Linus's HEAD. For more information on this work, have a look at this thread.

## Wakelocks

Of all the Androidisms, this is likely the one that was most contentious. The discussion threads covering its inclusion in the mainline kernel generated close to 2,000 emails,

2. Git is a distributed source code management tool created by Linus Torvalds to manage the kernel sources. You can find more information about it at *http://git-scm.com/*.

and even then there was no clear path for merging the wakelock functionality. It was only after the 2011 Kernel Summit, where kernel developers agreed to merge most Androidisms into the mainline, that efforts were made to try to rehabilitate the wakelock mechanism or, as was ultimately decided, to create an equivalent that was more palatable to the rest of the kernel development community.

As of the end of May 2012, equivalents to the wakelocks and their correlated *early suspend* mechanisms have been merged into the mainline kernel. The early suspend replacement is called *autosleep*, and the wakelock mechanism has been replaced by a new `epoll()` flag called `EPOLLWAKEUP`. The API is also therefore different from the original functionality added by the Android team, but the resulting functionality is effectively the same. At the time of this writing, it's expected that the new versions of the AOSP would start using the new mechanisms instead of the old ones.

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 laptops 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 for 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 explicitly 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, used to be that it became impossible to push that driver into the mainline kernel, because the mainline didn't include wakelock support. Given the recent inclusion of equivalent functionality into the mainline, this is no longer an issue.

The following LWN articles describe wakelocks in more detail and explain the various issues surrounding their inclusion in 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
- KS2011: Patch review
- Bringing Android closer to the mainline
- Autosleep and wake locks
- 3.5 merge window part 2

## Low-Memory Killer

As 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 are not high priority.

Android's low-memory killer is based on the OOM adjustments mechanism available in Linux that enables the enforcement of different OOM kill priorities for different processes. Basically, the OOM adjustments allow the 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 kicks in only 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 57), 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 for, among many other things, 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 traditionally builds on it.

At the time of this writing, Android's low-memory killer is found in the kernel's staging tree along with many of the other Android-specific drivers. Work is currently under way to rewrite this functionality within a more general framework for low-memory conditions. Have a look at the Userspace low memory killer daemon post to the Linux Kernel Mailing List (LKML) and the linux-vmevent patch for a glimpse of what's currently being worked on. Essentially, the goal is to move the decision process about what to do in low-memory conditions to a daemon in user-space.

## Android and the Linux Staging Tree

At the time of this writing, many of the drivers required to run Android have been merged into the *staging* tree. While this means they are still found in mainline kernels available at *http://kernel.org*, it also means that kernel developers believe those drivers require work before being considered mature enough to be merged alongside the "clean" set of drivers found in the rest of the kernel tree.

Specifically, many Android drivers are currently found in the *drivers/staging/android* directory of the kernel. They should remain there until they have been refactored or rewritten to suit the criteria for them to be admitted as official Linux drivers into the relevant location within the *drivers/* directory.

If you aren't familiar with the staging tree, have a look at Greg Kroah-Hartman's[3] The Linux Staging Tree, what it is and is not blog post from March 2009: "The Linux Staging tree (or just 'staging' from now on) is used to hold standalone drivers and filesystems that are not ready to be merged into the main portion of the Linux kernel tree at this point in time for various technical reasons. It is contained within the main Linux kernel tree so that users can get access to the drivers much easier than before, and to provide a common place for the development to happen, resolving the 'hundreds of different download sites' problem that most out-of-tree drivers have had in the past."

3. Greg is one of the top kernel developers and maintainers.

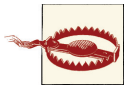
## Binder

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 standalone project, a few key developers who had worked on it, such as Dianne Hackborn and Arve Hjønnevåg, eventually ended up working on 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 underpinnings and its design philosophy, and so is Dianne Hackborn's explanation on the LKML of how the Binder is used in Android.

In essence, Binder attempts to provide remote object invocation capabilities on top of a classic OS. In other words, instead of reengineering 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 cornerstone 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 26 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 privilege sandbox associated with the app they are part of. Services running within the System Server, on the other hand, typically run with system privileges and live from boot to reboot. The only things these two types of services share are: a) their name, and 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 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.

Since the 3.3 release of the Linux kernel, the Binder driver has been merged into the staging tree. There is currently no project under way to clean this driver up or to rewrite it to make it applicable and/or useful for more general-purpose use in standard Linux desktop and server systems. It's therefore likely to remain in *drivers/staging/android/* for the foreseeable future.

## 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 of the System V IPC mechanisms. If you look at the *bionic/libc/docs/SYSV-IPC.TXT* 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 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 uses 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. "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—through the `IMemory` interface, rather than directly.

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

At the time of this writing, the ashmem driver is included in the mainline's *drivers/staging/android/* directory and is slated for rewriting.

## 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; for more information, see the `setitimer()`'s man page. The system call allows for a handful of types of timers, one of which, `ITIMER_REAL`, uses the kernel's 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()` to, among other things, 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 of 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 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 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.

At the time of this writing, Android's alarm driver is in the kernel's staging tree with upstreaming work pending.

## Logger

Logging is another essential component of any Linux system, embedded ones included. Being able to analyze a system's logs for errors or warnings either postmortem 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 are found in Android. Instead, Android defines its own logging mechanisms based on the Android logger driver added to the kernel. The classic 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 in RAM where it logs every incoming event and returns immediately back to the caller.

There are numerous benefits to avoiding file-writes in the settings in which Android is used. For example, unlike in a desktop or server environment, it isn't necessarily desirable to have a log that grows indefinitely in an embedded system. It's also desirable to have a system that enables logging even though the filesystem types used may be read-only. Furthermore, most Android devices rely on solid-state storage devices, which have a limited number of erase cycles. Avoiding superfluous writes is crucial in those cases.

Because of its lightweight, efficient, and embedded-system-friendly design, Android's logger can actually be used by user-space components at runtime 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, likely would have 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 functions 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.

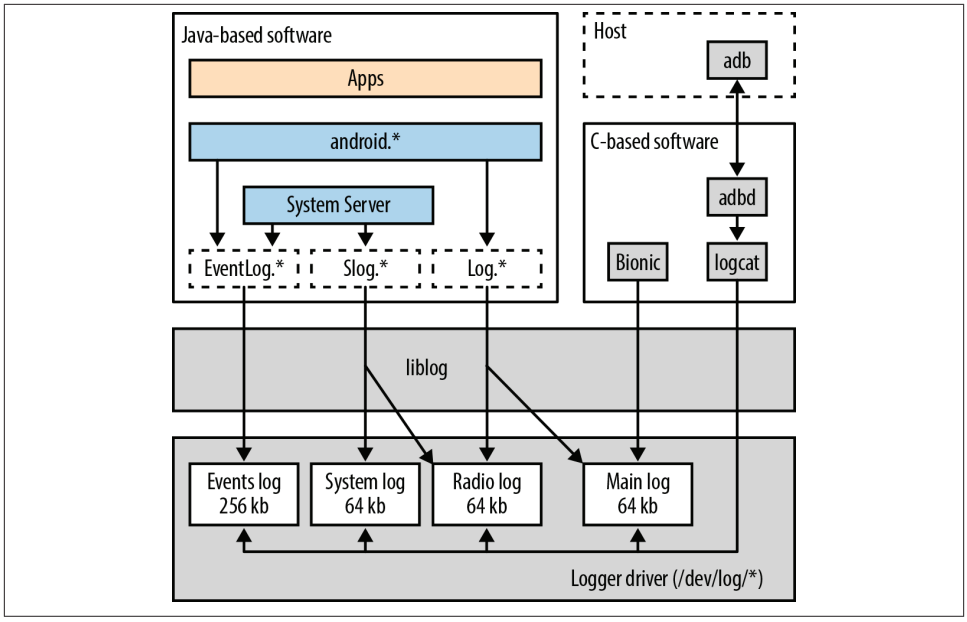



*Figure 2-2. Android's logging framework*

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 as mainly for 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 either `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 development 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.

At the time of this writing, the logger driver has been merged into the kernel's *drivers/staging/android/* directory. Have a look at the Mainline Android logger project for more information regarding the state of this driver's mainlining.

## Other Notable Androidisms

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

  At the time of this writing, this functionality hasn't been merged into the mainline, and the path for its inclusion is unclear. You could run an AOSP on a kernel that doesn't have this functionality, but Android's permission system, especially with regard to socket creation, would be broken.

*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 postmortem 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*.

  At the time of this writing, the RAM console's functionality seems to have been merged into mainline within the pstore filesystem in the kernel's *fs/pstore/* directory.

*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 was

used in very few devices. In fact, according to Brian Swetland, one of the Android kernel development team members, it was written to specifically address the MSM7201A's limitations, the MSM7201A being the SoC in the G1.

At the time of this writing, this driver is considered obsolete and has been dropped. It isn't found in the mainline kernel, and there are no plans to revive it. It appears that the ION memory allocator is poised to replace whatever uses pmem had.

# Hardware Support

Android's hardware support approach is significantly different from the classic approach typically found in the Linux kernel and Linux-based distributions. Specifically, the way hardware support is implemented, the abstractions built on that hardware support, and the mind-set 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 Memory Technology Device (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 has allowed various software stacks to be built on top of */dev* nodes either to interact with the hardware directly or to 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 this 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 for getting 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.

## 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. The ALSA libraries typically used in Linux desktop environments to interface with ALSA drivers, for example, aren't used in the official AOSP tree. Instead, recent Android releases include a BSD-licensed *tinyalsa* library as a replacement.

Figure 2-3 presents the typical way in which hardware is abstracted and supported in Android, along with 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. Generally speaking, you can consider the HAL layer as being the hardware library loader shown in the diagram, along with the header files defining the various hardware types, with those same header files being used as the API definitions for the hardware library *.so* files.

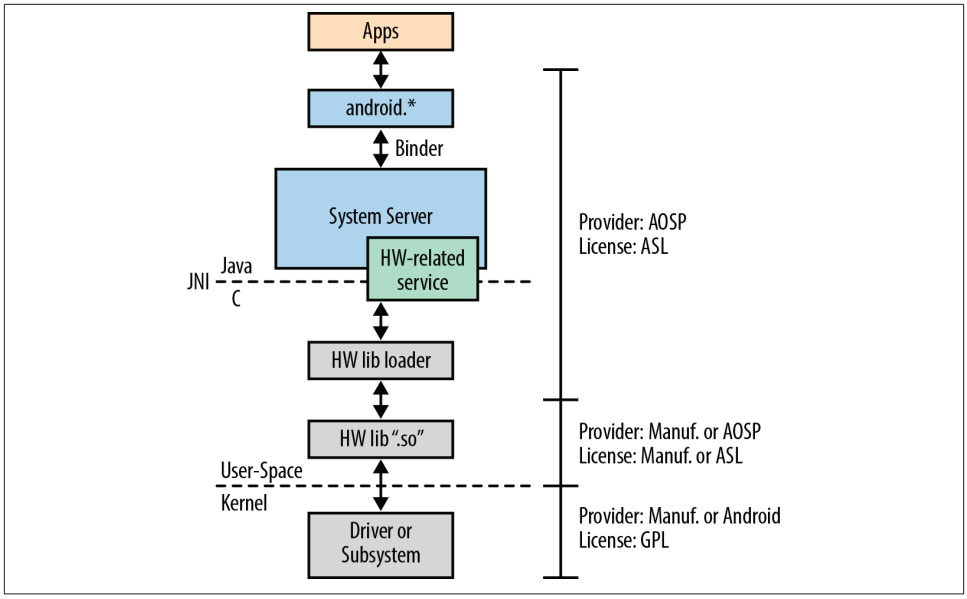



*Figure 2-3. Android's "Hardware Abstraction Layer"*

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 the HAL. 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 library. That system service will be responsible for

interacting and coordinating with the other system services to make the hardware behave 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/C++ whose main job is to interact with the HAL, the hardware-supporting shared library and other low-level functions.

## Loading and Interfacing Methods

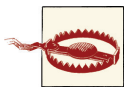
As I mentioned earlier, there are various ways in which system services and Android in general interact with the shared libraries implementing hardware support and hardware devices in general. It's difficult to fully understand why there is such a variety of methods, but I suspect that some of them evolved organically. Luckily, there seems to be a movement toward 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 foreseeable 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. Also applies to Audio and Camera starting from 4.0/Ice-Cream Sandwich.*

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.



HAL "modules" shouldn't be confused with loadable kernel modules, which are a completely different and unrelated software construct, even though they share some similar properties.

*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. Use of sysfs entries instead of */dev* nodes makes sense, for instance, when defaults need to be set during system initialization when no part of the framework is yet running.

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

  Arguably, 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 communication is likely hidden from Android itself because it interacts with a shared library instead, but in some corner 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 freedesktop.org's D-Bus page for more information.

Given that BlueZ has been removed from the AOSP starting with 4.2/Jelly Bean, it's unclear what uses D-Bus will have, if any, in future Android releases.

## 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 that were used by Google to develop new Android releases and therefore served as lead devices. Sometimes the sources for hardware support are quite extensive, as was the case for the Samsung Nexus S (a.k.a. "Crespo," its code name) in Gingerbread, and the Galaxy Nexus (a.k.a. "Maguro") and the Nexus 7 (a.k.a. "Grouper") in Jelly Bean.

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 Hardware Support | Interface to Hardware |
|---|---|---|---|
| Audio | Audio Flinger | Linker-loaded[a] *libaudio.so* | Up to hardware manufacturer, though ALSA is typical |
| Bluetooth | Bluetooth Service | Socket/D-Bus to BlueZ[b] | BlueZ stack |
| Camera | Camera Service | Linker-loaded[c] *libcamera.so* | Up to hardware manufacturer, sometimes Video4Linux |
| Display | Surface Flinger | HAL-loaded *gralloc* module[d] | Up to hardware manufacturer, */dev/fb0* or */dev/graphics/fb0* |
| GPS | Location Manager | HAL-loaded *gps* module | Up to hardware manufacturer |
| Input | Input Manager | Native *libui.so* library[e] | Entries in */dev/input/* |
| Lights | Lights Service | HAL-loaded *lights* module | Up to hardware manufacturer |
| Media | N/A, StageFright framework within Media Service | dlopen on *libstagefrighthw.so* | Up to hardware manufacturer |
| Network interfaces[f] | Network Management Service | Socket to *netd* | `ioctl()` on interfaces |
| Power management | Power Manager Service | Linker-loaded *libhardware_legacy.so* | Entries in */sys/power/* or, in older days, */sys/android_power/* |

| Hardware | System Service | Interface to User-Space Hardware Support | Interface to Hardware |
|---|---|---|---|
| Radio (Phone) | Phone Service | Socket to *rild*, which itself does a `dlopen()` on manufacturer-provided *.so* | Up to hardware manufacturer |
| Storage | Mount Service | Socket to *vold* | System calls and */sys* entries |
| Sensors | Sensor Service | HAL-loaded *sensors* module | Up to hardware manufacturer |
| Vibrator | Vibrator Service | Linker-loaded *libhardware_legacy.so* | Up to hardware manufacturer |
| WiFi | Wifi Service | Linker-loaded *libhardware_legacy.so* | Classic *wpa_supplicant*[g] in most cases |

[a] This is HAL-loaded starting with 4.0/Ice-Cream Sandwich.

[b] BlueZ has been removed starting with 4.2/Jelly Bean. A Broadcom-supplied Bluetooth stack called *bluedroid* has replaced it. The new Bluetooth stack relies on HAL-loading like most other hardware types.

[c] This is HAL-loaded starting with 4.0/Ice-Cream Sandwich.

[d] The module used by the Surface Flinger is *hwcomposer* starting with 4.0/Ice-Cream Sandwich

[e] This has been replaced by the *libinput.so* library starting with 4.0/Ice-Cream Sandwich.

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

[g] *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 be invoked from the command line once a developer shells into the device. Such binaries usually have direct access to the root filesystem and the native libraries included in the system. Their capabilities are restricted only by the filesystem rights granted to them and their effective UID and GID. They aren't subject to any of the restrictions imposed on a typical Android app by the Android Framework because they are running outside 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 as much as possible to explain where Android's user-space is different from 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).[4] 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, or use some form of "*chroot* jail" to isolate it and its supporting packages from the rest of the root filesystem (see *chroot*'s man page for details).

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 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 that it might be possible to host Android side by side with a common Linux distribution on the same root filesystem. Have a look at Appendix A for more information on how to create such a hybrid.

*/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 read-only 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, though in read-write mode.

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 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*.

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

Interestingly, Android doesn't include any */bin* or */lib* directories. These directories are typically crucial in a Linux system, containing, respectively, essential binaries and essential libraries. This is yet another artifact 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 I haven't covered here. We will revisit the Android root filesystem and its makeup in more detail in Chapter 6.

## Libraries

Android relies on about 100 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 |
|---|---|---|---|
| *audio.so*, *liba2dp*, *input.so*, *libbluetooth* and *libbluetoothd* | BlueZ[a] | *http://www.bluez.org* | GPL |
| *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*[b] | 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, in 2.3/Gingerbread, *libskiagl.so* | skia 2D graphics library | *http://code.google.com/p/skia/* | ASL |
| *libsonivox* | Sonic Network's Audio Synthesis library | N/A | ASL |

| Library(ies) | External Project | Original Location | License |
|---|---|---|---|
| *libsqlite.so* | SQLite database | *http://www.sqlite.org* | Public domain |
| *libSR_AudioIn.so* and, in 2.3/Gingerbread, *libsrec_jni.so* | Nuance Communications' Speech Recognition engine | N/A | ASL |
| *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.so* | Library used by legacy HAL to talk to wpa_supplicant daemon | *http://hostap.epitest.fi/wpa_supplicant/* | GPL and BSD |
| *libz.so* | zlib compression library | *http://zlib.net* | Custom, BSD-like |

[a] BlueZ has been replaced by an ASL-licensed, Broadcom-supplied Bluetooth stack called *bluedroid* that is also found in *external/*. The libraries generated by bluedroid are different from those listed here.

[b] Note that Android's *libexif.so*'s API is very different from that library's API as available in traditional Linux distributions.

*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* | C++ support library[a] |
| | *libthread_db.so* | Thread debugging library |
| Core[b] | *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*, *libsensorservice.so*, and *libcameraservice.so* | System-services-related libraries |
| | *libcamera_client.so* and, in 2.3/Gingerbread, *libsurfaceflinger_client.so*[c] | 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* | Library for functions related to sensors and, starting with 4.0/Ice-Cream Sandwich, client communication with the Surface Flinger |

| Category | Library(ies) | Description |
|---|---|---|
| | *liblog.so* | The logging library |
| | *libandroid_runtime.so* | The Android Runtime library |
| | *libandroid.so* | C interface to lifecycle management, input events, window management, assets, and Storage Manager |
| Dalvik | *libdvm.so* | The Dalvik VM library |
| | *libnativehelper.so* | JNI-related helper functions |
| Hardware | *libhardware.so* | The HAL library that provides `hw_get_module()` and uses `dlopen()` to load hardware support modules (i.e., shared libraries that provide hardware support to the HAL) on demand |
| | *libhardware_legacy.so* | Legacy HAL library providing hardware support for WiFi, power-management, and vibrator |
| | Various hardware-supporting shared libraries | Libraries that provide support for various hardware components; some are loaded 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 (Digital Rights Management) framework libraries |
| OpenGL | *libEGL.so*, *libETC1.so*, *libGLESv1_CM.so*, *libGLESv2.so*, and *egl/ligGLES_android.so* | Android's OpenGL implementation |

[a] Some say that this library is similar in its role to the *libsupc++.a* found in standard Linux systems, while Android's *libstlport.so* is closer to traditional Linux systems' *libstdc++.so*.

[b] I'm using this category as catchall for many core Android functionalities.

[c] Starting with 4.0/Ice-Cream Sandwich, the functionality corresponding to *libsurfaceflinger_client.so* has been merged into *libgui.so*.

Since 2.3/Gingerbread, many libraries have been added to that AOSP. Tables 2-4 and 2-5 list some of the most notable additions you'll find in 4.1/Jelly Bean.

*Table 2-4. Important libraries from external projects found in 4.1/Jelly Bean*

| Library(ies) | External Project | Original Location | License |
|---|---|---|---|
| *libtinyalsa.so* | tinyalsa | *http://github.com/tinyalsa* | ASL |
| *libmtp.so* | libmtp | *http://libmtp.sourceforge.net/* | LGPL |
| *libchromium_net.so* | WebKit | *http://webkit.org/* | LGPL and BSD |
| *libmdnssd.so* | mDNSResponder | *http://www.opensource.apple.com/tarballs/mDNSResponder/* | ASL |

*Table 2-5. Important Android-specific libraries found in 4.1/Jelly Bean*

| Category | Library(ies) | Description |
|---|---|---|
| Core | *libjnigraphics.so* | C interface to the 2D graphics system |
| | *libcorkscrew.so* | Debugging library |
| | *libRS.so* | Interface to RenderScript |
| Media | *libOpenMAXAL.so* | Native multimedia library, based on Khronos OpenMAX AL |
| | *libOpenSLES.so* | Khronos OpenSL EL compatible audio system |
| | *libaudioutils.so* | Echo cancellation and other audio tools |

## Init

One thing Android doesn't change is the kernel's boot process. Hence, whatever you know about the kernel's startup continues to apply just the same to Android's use of Linux. What changes in Android is what happens once the kernel finishes booting. Indeed, after it's finished initializing itself and the drivers it contains, the kernel starts just one user-space 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. Traditionally, Linux distributions have relied on SystemV init for the init process, 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 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*, where *<device_name>* is the name of the device. In some cases, such as the emulator, for example, there's also a device-specific script stored as */system/etc/init.<device_name>.sh*. 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—a key system component that we'll cover in greater detail later in this chapter and in Chapter 7—from starting up automatically and then starting it manually yourself after having used *adb* to shell into the device.

We'll discuss the init's configuration language in depth in Chapter 6.

### 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. Some of the 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 100 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.

We'll discuss the init's global properties, the files used to provide its default values, and the relevant commands in greater detail in Chapter 6.

### udev events

As I explained earlier, access to devices in Linux is done through nodes within the */dev* directory. In the old days, Linux distributions would ship with thousands of entries in that directory to accommodate 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.

We'll discuss the *ueventd* and its configuration files in detail in Chapter 6.

## Toolbox

Much like the root filesystem's directory hierarchy, there are essential binaries on most Linux systems, 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 projects available on the Internet. In an embedded system, it doesn't make sense to have to deal with so many packages, nor necessarily to 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 to 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-rich 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 be 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 is. 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 from the final production release. I'll cover this in more detail in Appendix A.

## 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 daemons, such as *adbd*, are started on demand, depending on build options and changes to global properties. Table 2-6 provides a list of some of the more prominent daemons that Android runs. Many of these are discussed in much greater detail in Chapters 6 and 7.

*Table 2-6. Android daemons*

| Daemon | Description |
|---|---|
| *ueventd* | Android's replacement for *udev*. |
| *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. |
| *rild* | The RIL daemon. Mediates all communication between the Phone Service and the Baseband Processor. |
| 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. |

| Daemon | Description |
|---|---|
| *mediaserver* | The Media server. Hosts most media-related services. We'll 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. No longer in the AOSP as of 4.2/Jelly Bean, since the BlueZ stack has been removed. |
| *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-value pair store for cryptographic keys, SSL certs for instance. |
| *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 throughout 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 Chapters 6 and 7.

## 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 I can delve into that explanation, I 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, two aspects are important to keep in mind with regard to Java: its differences from 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 you write in Java gets compiled by a Java compiler into architecture-independent byte-code that is executed

at runtime by a byte-code interpreter, also commonly referred to as a "virtual machine." This modus operandi, along with Java's semantics, enables 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 keep track of objects you allocate. In fact, it requires you to lose track of all unused objects, since it has an integrated garbage collector that will ensure that 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 at build time, 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.

None of the JDK components are found in the images generated by the build of the AOSP. Hence, none of the JDK's components would be distributed by you when using Android for your embedded system.

## Java Lingo

Java has its own specialized terminology. The following explanations should help you make sense of some of the terms being used in the text, if you aren't already familiar with them:

*virtual machine*
: This term was less ambiguous when Java came out, because "virtual machine" software products 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.

*reflection*
: The ability to ask an object whether it implements a certain method.

*anonymous classes*
: 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 see the code handling an event at the same location in the source code where she invokes the callback registration method.

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

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 little 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.

For 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."

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 the inside track on the benefits and trade-offs between stack-based VMs and 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, p. 153–163.

A feature of Dalvik very much worth highlighting, though, is that since 2.2/Froyo it has included a Just-in-Time (JIT) compiler for ARM, with x86 and MIPS having been added since. Historically, JIT has been a defining feature for many VMs, helping them close the gap with noninterpreted languages. Indeed, having a JIT 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 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 is available for a limited number of architectures only, namely ARM, x86, and MIPS.

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 with 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 bridge to other languages such as C and C++. It's an equivalent to *P/Invoke* 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. Internally, though, the AOSP relies massively 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.

Appendix B shows an example use of JNI to interface Java and C code. For the moment, 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 ensure that you use the appropriate call semantics and function parameters.

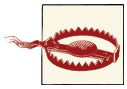
Unfortunately, JNI seems to be a dark art reserved for the initiated. In other words, it's rather difficult to find good documentation on it. There is one authoritative book on the topic, *The Java Native Interface Programmer's Guide and Specification* by Sheng Liang (Addison-Wesley, 1999).

# 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 to 70 system services. These services cooperate 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 one another 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 same 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. Another set of system services is 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 multimedia framework and audio effects. Finally, the Phone application houses the Phone service separately from the rest. Since 4.0/Ice-Cream Sandwich, note that the Surface Flinger has been forked off into a separate standalone process.



The terminology here isn't my choosing, and it's unfortunately confusing. The "System Server" process houses **several** system services within the same process. So does the "Media Service." Both "System Server" and "Media Service" are spelled out as **singular** regardless of the number of system services they comprise. When this book refers to "system services," plural, it refers to all system services available in the system regardless of the process they run under. So, in short, neither "System Server" nor "Media Service" are part of the "system services." Instead, they are processes used to run the latter.

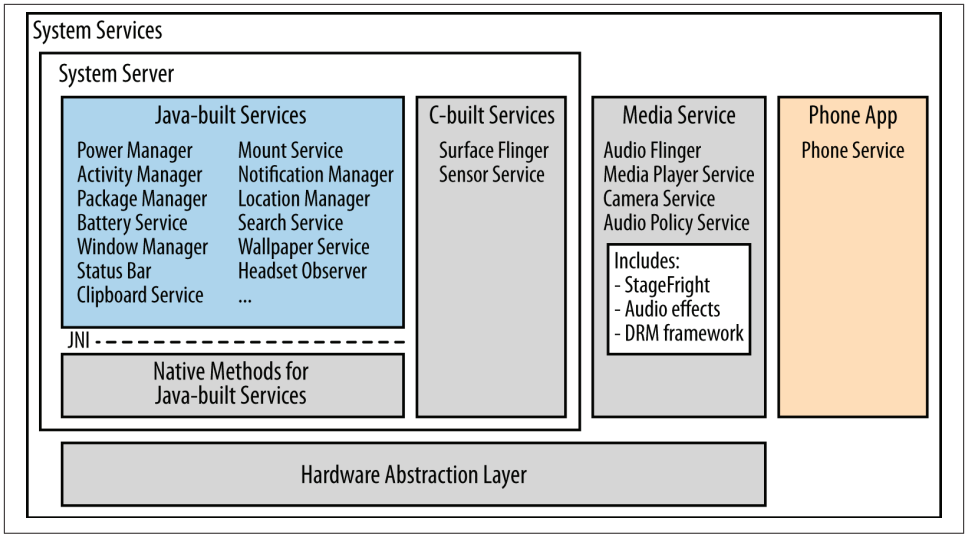



*Figure 2-4. System services*

Note that despite there being only a handful of 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 an Android 2.3/Gingerbread 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]
```

Here's the same output on a 4.2/Jelly Bean emulator:

```
root@android:/ # service list
Found 68 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 dreams: [android.service.dreams.IDreamManager]
5 commontime_management: []
6 samplingprofiler: []
7 diskstats: []
8 appwidget: [com.android.internal.appwidget.IAppWidgetService]
9 backup: [android.app.backup.IBackupManager]
10 uimode: [android.app.IUiModeManager]
11 serial: [android.hardware.ISerialManager]
12 usb: [android.hardware.usb.IUsbManager]
13 audio: [android.media.IAudioService]
14 wallpaper: [android.app.IWallpaperManager]
15 dropbox: [com.android.internal.os.IDropBoxManagerService]
16 search: [android.app.ISearchManager]
17 country_detector: [android.location.ICountryDetector]
18 location: [android.location.ILocationManager]
19 devicestoragemonitor: []
```

```
notification: [android.app.INotificationManager]
updatelock: [android.os.IUpdateLock]
throttle: [android.net.IThrottleManager]
servicediscovery: [android.net.nsd.INsdManager]
connectivity: [android.net.IConnectivityManager]
wifi: [android.net.wifi.IWifiManager]
wifip2p: [android.net.wifi.p2p.IWifiP2pManager]
netpolicy: [android.net.INetworkPolicyManager]
netstats: [android.net.INetworkStatsService]
textservices: [com.android.internal.textservice.ITextServicesManager]
network_management: [android.os.INetworkManagementService]
clipboard: [android.content.IClipboard]
statusbar: [com.android.internal.statusbar.IStatusBarService]
device_policy: [android.app.admin.IDevicePolicyManager]
lock_settings: [com.android.internal.widget.ILockSettings]
mount: [IMountService]
accessibility: [android.view.accessibility.IAccessibilityManager]
input_method: [com.android.internal.view.IInputMethodManager]
input: [android.hardware.input.IInputManager]
window: [android.view.IWindowManager]
alarm: [android.app.IAlarmManager]
vibrator: [android.os.IVibratorService]
battery: []
hardware: [android.os.IHardwareService]
content: [android.content.IContentService]
account: [android.accounts.IAccountManager]
user: [android.os.IUserManager]
permission: [android.os.IPermissionController]
cpuinfo: []
dbinfo: []
gfxinfo: []
meminfo: []
activity: [android.app.IActivityManager]
package: [android.content.pm.IPackageManager]
scheduling_policy: [android.os.ISchedulingPolicyService]
telephony.registry: [com.android.internal.telephony.ITelephonyRegistry]
display: [android.hardware.display.IDisplayManager]
usagestats: [com.android.internal.app.IUsageStats]
batteryinfo: [com.android.internal.app.IBatteryStats]
power: [android.os.IPowerManager]
entropy: []
sensorservice: [android.gui.SensorServer]
media.audio_policy: [android.media.IAudioPolicyService]
media.camera: [android.hardware.ICameraService]
media.player: [android.media.IMediaPlayerService]
media.audio_flinger: [android.media.IAudioFlinger]
drm.drmManager: [drm.IDrmManagerService]
SurfaceFlinger: [android.ui.ISurfaceComposer]
```

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. In 2.3/Gingerbread there were about 85,000 lines of Java code in the System Server alone, spread across 100 different files. And that didn'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 nonexistent. 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 into that project. This won't compile in any way, but it'll allow you to benefit from Eclipse's Java browsing capabilities in trying to understand the code. For instance, you can open a single Java file, right-click 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 toward locating bugs, not reverse-engineering the code being analyzed. Still, some have reported that they've found Ctags and the open source AndroidXRef projects helpful in their efforts.

## Service Manager and Binder Interaction

As I explained earlier, the Binder mechanism used as system services' underlying fabric enables object-oriented RPC/IPC. 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.

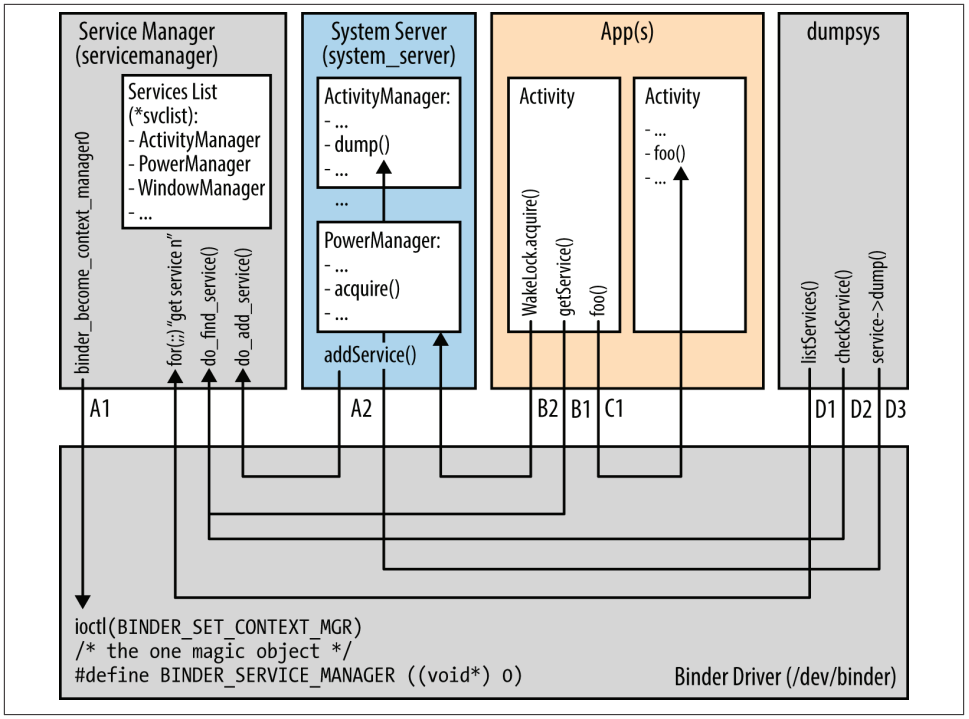



*Figure 2-5. Service Manager and Binder interaction*

Think of the Service Manager as a Yellow Pages book of all services available in the system. If a system service isn't registered with the Service Manager, then 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 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 not looked up through the Service Manager.

The Service Manager is also used in a special way by a number of command-line utilities such as the *dumpsys* utility, which allows you to dump the status of a single or all system services. To get the list of all services, *dumpsys* 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, *dumpsys* just asks the Service Manager to locate that specific one (D2). With a service handle in hand, *dumpsys* 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 regular app developers. 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. Appendix B shows you how to add a system service and make it available through `getSystemService()`.

## A Service Example: the Activity Manager

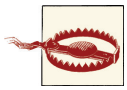
Although covering each and every system service is outside the scope of this book, let's have a quick look at the Activity Manager, one of the key system services. In 2.3/Gingerbread, the Activity Manager's sources actually span over 30 files and 20,000 lines of code. If there's a core to Android's internals, 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 an icon to start an app from his home screen, the first thing that happens is the Launcher's `onClick()` callback is called (the Launcher being the default app packaged with the AOSP that takes care of the main interface with the user, the home screen). 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 Maps, YouTube, and Gmail aren't part of the AOSP. Let's take a look at some of the most notable packages included by default; as we'll see below, the AOSP includes many more packages. Table 2-7 lists the most important stock apps included in the 2.3/Gingerbread AOSP; Table 2-8 lists that AOSP's main content providers; and Table 2-9 lists the corresponding IMEs (input method editors).



While stock apps 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 2-7. Stock AOSP apps*

| App in AOSP | Name Displayed in Launcher | Description |
|---|---|---|
| AccountsAndSyncSettings | N/A | Accounts management app |
| Bluetooth | N/A | Bluetooth manager |
| Browser | Browser | Default Android browser, includes bookmark widget |
| Calculator | Calculator | Calculator app |
| Calendar | Calendar | Calendar 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 |
| DownloadProviderUi | Downloads | UI for DownloadProvider |
| Development | Dev Tools | Miscellaneous dev tools |
| Email | Email | Default Android email app |
| 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 |

| App in AOSP | Name Displayed in Launcher | Description |
|---|---|---|
| Nfc | N/A | NFC configuration UI and NFC system service |
| PackageInstaller | N/A | App install/uninstall UI |
| Phone | Phone | Default phone dialer/UI and phone system service |
| 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 |
| SoundRecorder | N/A | Sound recording app; activated when recording intent is sent, not by user |
| SpeechRecorder | Speech Recorder | Speech recording app |
| SystemUI | N/A | Status bar |

*Table 2-8. Stock AOSP providers*

| Provider | Description |
|---|---|
| ApplicationsProvider | Provider to 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 |
| UserDictionaryProvider | Storage and provider for user-defined words dictionary |

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

*Table 2-9. Stock AOSP input methods*

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

The AOSP contains a lot more packages than those listed in the above tables. Indeed, if you search the sources, you'll find that a 4.2/Jelly Bean release can generate about 500 apps. A large number of those are either tests or samples and aren't worth focusing on in the current discussion. Roughly a quarter of these apps are worth putting into a final product, and they are mostly found in the following directories of the AOSP:

- *packages/apps/*
- *packages/inputmethods/*
- *packages/providers/*
- *packages/screensavers/* (new to 4.2/Jelly Bean)
- *packages/wallpapers/*
- *frameworks/base/packages/*
- *development/apps/*

You'll probably want to look at the content of those directories in conjunction with the above tables to determine which packages are worth further investigation in the context of your project. Like many other things in the AOSP, of course, the packages it contains change over time, as do their locations. Here's a summary of some of the location changes that have occurred between 2.3.4/Gingerbread and 4.2/Jelly Bean:

- AccountAndSyncSettings and Gallery3D have been removed from *packages/apps/*, and the following packages have been added: CellBroadcastReceiver, SmartCardService, BasicSmsReceiver, Exchange, Gallery2, KeyChain, MusicFX, SpareParts, VideoEditor, and LegacyCamera.
- TtsService and VpnServices have been removed from *frameworks/base/packages/*, and the following packages have been added: BackupRestoreConfirmation, SharedStorageBackup, VpnDialogs, WAPPushManager, FakeOemFeatures, FusedLocation, and InputDevices.

## System Startup

The best way to bring together everything we've 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 hardcoded 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 SoC devices, which include a CPU and a slew of peripherals 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 onboard flash chips because they boot straight from an SD card.

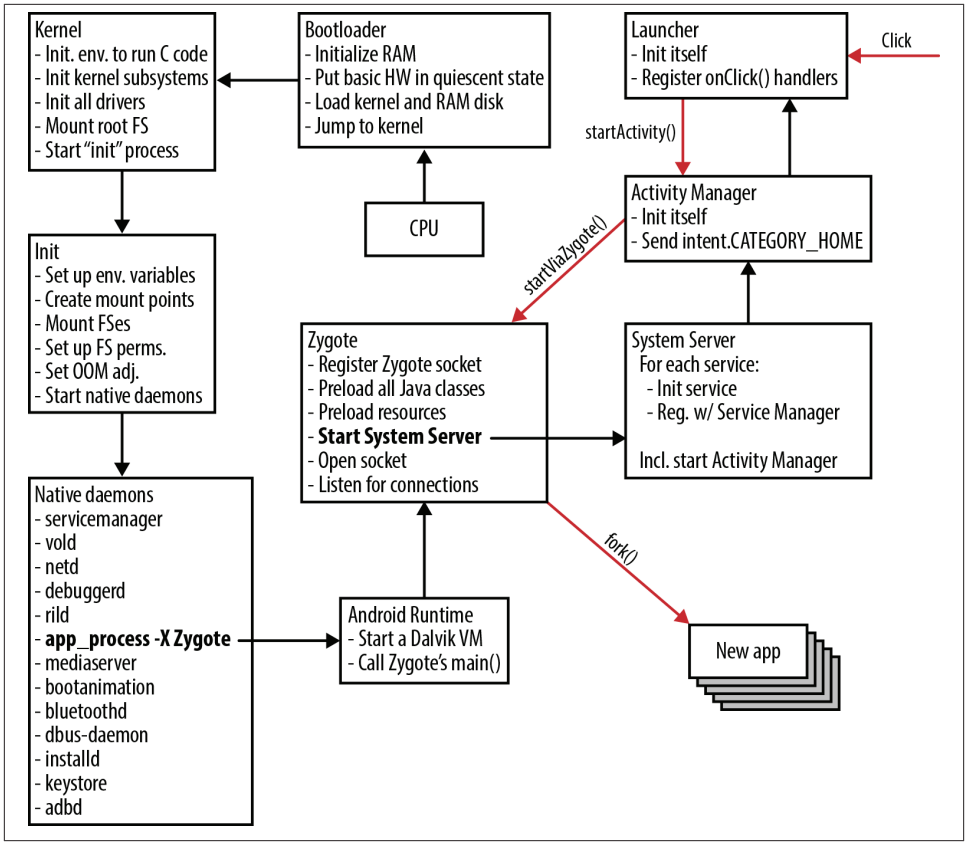



*Figure 2-6. Android's boot sequence*

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 its various subsystems, and proceeds 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. The 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 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. 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 to fork 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 an icon on the home screen, the process I described in "A Service Example: the Activity Manager" on page 70 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 a 2.3/Gingerbread Android emulator, so it contains some emulator-specific artifacts such as the *qemud* daemon. Notice that the apps running all bear 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's 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.

Most importantly, notice that the Zygote's process identifier (PID) is 32 and the the parent PID (PPID) of all apps is 32. This illustrates the earlier explanations that the Zygote is the parent of all apps in the system.