

Android Framework

"native" == provided by the underlying Linux

Android Apps/Programs == .apk
Linux programs == ../bin, ../xbin

Ultimately, your goal is to get your embedded system to run the Android environment users and developers are accustomed to, not simply the native user-space we just covered. That includes not only the full set of system services and the packages that provide the standard APIs used by app developers, but also some less visible components, such as a set of native daemons that support the system services and the Hardware Abstraction Layer. This chapter will cover how the Android Framework operates on top of the native user-space and will discuss how to interact with and customize it.

Note that unlike the previously discussed components of Android, whose behavior can be modified in a number of ways, **most of the Android Framework has to be used as is**. You can't, for instance, pick and choose which system services to run, as they aren't started from a script or based on a configuration file. Instead, modifying the Framework typically requires diving into its sources and/or adding your own code to customize its behavior.

Such customization work therefore requires becoming intimately familiar with Android's sources and is inherently version dependent. Still, we'll try to cover enough of the essentials to enable you to start navigating Android's internals on your own. Nevertheless, expect this to be the start of a long-term endeavor, as Android's sources are fairly big, and new releases come out at a very rapid pace.

What Exactly Is the "Android Framework"?

If you refer back to **Figure 2-1**, the Android Framework includes the `android.*` packages, the System Services, the Android Runtime, and some of the native daemons. Sourcewise, the Android Framework is typically composed of all the code located in the `frameworks/` directory of the AOSP.

At a certain level, I'm using "Android Framework" here to designate practically everything "Android" that runs on top of native user-space. So my explanations here do

sometimes go beyond just *frameworks/*. Namely, I will discuss such things as Dalvik and the HAL, which are intrinsic to the Android Framework.

Kick-Starting the Framework

We closed the last chapter on the *init* command, and how it can be configured and used. I only briefly hinted, however, at how the Android Framework is started by way of the Zygote when describing the default *init.rc*. There is of course much to say on this topic, as we'll see shortly. Much of what I described in the last chapter can be easily compared to components that exist in the embedded Linux world; however, very little of what follows has any such equivalent. Indeed, the Android developers' contribution to the world of mobile is the stack they built on top of a BSD/ASL-licensed embedded Linux equivalent.

Building the AOSP Without the Framework

As odd as it may seem, there are cases where you actually may want to build the AOSP without all the fancy, Java-based system services and apps that Android is most widely known for. Whether it be to run “Android” on a “headless” system or simply because you're in the midst of a board bringup and would like a minimal build of the AOSP to get just the basic tools and environment of the native user-space, there's an AOSP build for you: Tiny Android.

To make the AOSP generate Tiny Android, you just need to go to the AOSP's source directory and type this:

```
$ BUILD_TINY_ANDROID=true make -j16
```

This will get you a set of output images with the minimal set of Android components for a functional native Android user-space to run with a kernel. Mainly, you'll get Toolbox, Bionic, *init*, *adbd*, *logcat*, *sh*, and a few other key binaries and libraries. No part of the Android Framework, such as the system services or any of the apps, will be included in those images.

It's questionable whether this is “Android” anymore, but in some cases it's exactly what you're looking for. Whether you want to refer to the end result as “Android” is really up to you. Hey, apparently beauty is in the eye of the beholder.

Core Building Blocks

The Framework's operation relies on a handful of key building blocks: the Service Manager, the Android Runtime, the Zygote, and Dalvik. Without these, none of the components that make up what we know to be Android work. We've already covered most



of these and their role in the system's startup in [Chapter 2](#). I encourage you to go back to that chapter for an in-depth discussion, but let's still recap the highlights here, especially now that we've just looked at *init* and its scripts. You may, in fact, want a finger on the pages from [Appendix D](#) about the main *init.rc* file as you read the following explanations.

One of the first services started by *init* is the *servicemanager*. As I explained earlier, this is the “Yellow Pages” or the directory of all system services running. Obviously, at the time it starts no system services have started, but it needs to be available very early on so that system services that do start can register with it and therefore become visible to the rest of the system.

If the *servicemanager* isn't running, none of the system services will be able to advertise themselves, and the Framework simply will not work. Hence, the *servicemanager* is not an optional component, and its ordering in the *init.rc* file isn't subject to customization. You must leave it exactly where it is in the main *init.rc* file with the options that are specified for it by default.

The next core component to get started is the Zygote. Here's the relevant line from *init.rc*:

```
service zygote /system/bin/app_process Xzygote /system/bin --zygote --start-sys
tem-server
```

There is a lot happening in that simple line. First, note that what's actually getting run is this *app_process* command. Here's its formal parameter list:

```
Usage: app_process [java-options] cmd-dir start-class-name [options]
```

app_process is a little-known command that packs a punch. It lets you start a new Dalvik VM for running ~~Android code~~ straight from the command line. This doesn't mean you can use it to start regular Android apps from the command line; in fact you can't use it for that purpose, but you'll soon learn about a command that does: *am*. However, some key system components and tools must be started from the command line without a reference to any existing Dalvik VM instance. The Zygote is one of these, as it's the first Dalvik process to run; *am* and *pm* are two more, which we'll cover later.

To do its magic, *app_process* relies on the Android Runtime. Packaged as a shared library, *libandroid_runtime.so*, the Android Runtime is capable of starting and managing a Dalvik VM for the purpose of running Android-type code. Among other things, it **preloads** this VM with a number of libraries that are typically used by any code that relies on the Android APIs. This includes all the native calls, which are required by any of the Android Framework's Java code. These are registered with the VM so it can find them whenever a Java-coded Android Framework package calls on one of its native functions.

The Runtime also includes functions for facilitating operations typically done for all Android-type applications running on Dalvik. You can, in fact, consider Dalvik to be a

very raw, low-level VM that doesn't assume you're running Android-type code on top of it. To run Android-type code on top of Dalvik, the Runtime starts Dalvik with parameters specifically tailored for its use to run Java code that relies on the Android Java APIs—either those publicly documented in the developer documentation and made available through the SDK, or internal APIs available only as part of building internal Android code within the AOSP.

Furthermore, the Runtime relies on many native user-space functionalities. For instance, it takes into account some of the *init*-maintained global properties in order to gate the starting of the Dalvik VM, and it uses Android's logging functions to log the progress of the Dalvik VM's initialization. In addition to setting up the parameters used to start the Dalvik VM used to run Java code, the Runtime also initializes some key aspects of the Java and Android environment before calling the code's `main()` method. Most importantly, it provides a default exception handler for all threads running on the just-instantiated VM.

Note that the Runtime doesn't preload classes: That's something the Zygote does when it sets up the system for running Android apps. And since each use of the *app_process* command results in starting a separate VM, all non-Zygote instances of Dalvik will load classes on demand, not before your code starts running.

Dalvik's Global Properties

In addition to the global properties maintained by *init* that we discussed in the last chapter, Dalvik continues to provide the property system found in Java through `java.lang.System`. As such, if you're browsing some of the system services' sources, you might notice calls to `System.getProperty()` or `System.setProperty()`. Note that those calls and the underlying set of properties are completely independent from *init*'s global properties.

The Package Manager Service, for instance, reads the `java.boot.class.path` at startup. Yet, if you use *getprop* on the command line, you won't find this property as part of the list of properties returned by *init*. Instead, such variables are maintained within each Dalvik instance for retrieval and/or use by running Java code. The specific `java.boot.class.path`, for instance, is set in *dalvik/vm/Properties.c* using the `BOOT_CLASSPATH` variable set in *init.rc*.

You can find out more about Java System Properties in Java's [official documentation](#). Note that the semantics of the variable names used by *init*'s global properties are very similar to those used by Java System Properties.

Once it's started, a Java class launched using *app_process* can start using “regular” Android APIs and talk to existing system services. If it's built as part of the AOSP, it can use many of the `android.*` packages available to it at build time. The *am* and *pm*

commands, for instance, do exactly that. It follows that you, too, could write your own command-line tool completely in Java, using the Android API, and have it start separately from the rest of the Framework. In other words, it would be started and would run independently of the Zygote and everything that the Zygote causes to start as part of its own initialization.

But this still won't let you write a regular Android app that is started by *app_process*. Android apps can be started only by the Activity Manager using intents, and the Activity Manager is itself started as part of the rest of the system services once the Zygote itself is started. Which brings the discussion back to the startup of the Zygote.

For the Zygote to start properly and have it start the System Server, you must leave its corresponding *app_process* line intact in *init.rc*, in its default location. There's nothing that you can configure about the Zygote's startup. You can, however, influence the way the Android Runtime starts any of its Dalvik VMs by modifying some of the system's global properties. Have a look at the `AndroidRuntime::startVm(JavaVM** pJavaVM, JNIEnv** pEnv)` function in *frameworks/base/core/jni/AndroidRuntime.cpp* in either 2.3/Gingerbread or 4.2/Jelly Bean to see which global properties are read by the Android Runtime as it prepares to start a new VM. Note that any use of these properties to influence the setup of Dalvik VMs is likely to be version specific.

Once the Zygote's VM is started, the `com.android.internal.os.ZygoteInit` class's `main()` function is called, and it will preload the entire set of Android packages, proceed to start the System Server, and then start looping around and listening for connections from the Activity Manager asking it to fork and start new Android apps. Again, there is nothing to be customized here unless you can see something relevant to you in the list of parameters used to start the System Server in the `startSystemServer()` function in *frameworks/base/core/java/com/android/internal/os/ZygoteInit.java*. My recommendation is to leave this as is unless you have a very strong understanding of Android's internals.

Disabling the Zygote

While you can't configure what the Zygote does at startup, you can nevertheless disable its startup entirely by adding the `disabled` option to its section in *init.rc*. Here's how this is done in 2.3/Gingerbread:

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote
--start-system-server
    socket zygote stream 666
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
disabled
```

This will effectively prevent *init* from starting the Zygote at boot time, so none of the Android Framework's parts will start, including the System Server. This may be very useful if you're in the process of debugging critical system errors or developing one of the HAL modules, and you must manually set up debugging tools, load files, or monitor system behavior **before** key system services start up.

You can then start the Zygote, and the rest of the system:

```
# start zygote
```

System Services

As we saw in the last section, the System Server is started as part of the Zygote's startup, and we'll continue delving into that part of the process in this section. However, and as was discussed in [Chapter 2](#), there are also system services started from processes other than the System Server, and we'll discuss those in this section.

Starting with 4.0/Ice-Cream Sandwich, the very first system service to get started is the **Surface Flinger**. Up to 2.3/Gingerbread, it had been started as part of the System Server, but with 4.0/Ice-Cream Sandwich, it's started right before the Zygote and runs independently from the System Server and the rest of the system services. Here's the relevant snippet that precedes the Zygote's entry in *init.rc* in 4.2/Jelly Bean:

```
service surfaceflinger /system/bin/surfaceflinger
    class main
    user system
    group graphics drmrpc
    onrestart restart zygote
```

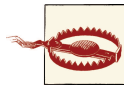


The Surface Flinger's sources are in *frameworks/base/services/surfaceflinger/* in 2.3/Gingerbread and *frameworks/native/services/surfaceflinger/* in 4.2/Jelly Bean. Its role is **to composite the drawing surfaces used by apps into the final image displayed to the user**. As such, it's one of Android's most fundamental building blocks.

In Android 4.0, because the Surface Flinger is started before the Zygote, the system's boot animation comes up much faster than in earlier versions. We'll discuss the boot animation later in this chapter.

To start the System Server, the Zygote forks and runs the `com.android.server.SystemServer` class' `main()` function. The latter loads the *libandroid_servers.so* library, which contains the JNI parts required by some of the system services and then invokes native code in *frameworks/base/cmds/system_server/library/system_init.cpp*, which starts C-coded system services that run in the `system_server` process. In 2.3/Gingerbread, this includes the Surface Flinger and the Sensor Service. In 4.2/Jelly Bean, however, the Surface Flinger is started separately, as we just saw, and the only C-coded system service started by `system_server` is the Sensor Service.

The System Server then goes back to Java and starts initializing the critical system services such as the **Power Manager, Activity Manager, and Package Manager**. It then continues to initialize all the system services it hosts and registers them with the Service Manager. This is all done in code in `frameworks/base/services/java/com/android/server/SystemServer.java`. None of this is configurable. It's all hardcoded into `SystemServer.java`, and there are no flags or parameters you can pass to tell the System Server not to start some of the system services. If you want to disable any, you'll have to go in by hand and comment out the corresponding code.



The system services are interdependent, and almost all of Android's parts, including the Android API, assume that all the system services built into the AOSP are available at all times. As I mentioned in **Chapter 2**, as a whole, the system services form an object-oriented OS built on top of Linux—and the parts of that OS weren't built with modularity in mind. So if you take one of the system services away, it's fair to assume that some of Android's parts will start breaking under your feet.

That doesn't mean it can't be done, though. As part of a presentation titled **"Headless Android"** at the 2012 Android Builders Summit, I showed how I successfully disabled the Surface Flinger, the Window Manager, and a couple of other key system services, to run the full Android stack on a headless system. As I warned in that presentation, that work was very much a proof of concept and would require a lot more effort to be production ready.¹

So, by all means, feel free to tinker around, but you've been warned that if you're going to play this deep in Android's guts, you'd better saddle up.

What's `/system/bin/system_server`?

You might notice while browsing your target's root filesystem that there's a binary called `system_server` in `/system/bin`. That binary, however, has nothing to do with the startup of the System Server or with any of the system services. It's unclear what purpose, if any, this binary has. It's very likely that this is a legacy utility from Android's early days.

This factoid is often a source of confusion, because a quick look at the list of binaries and the output of `ps` may lead you to believe that the `system_server` process is in fact started by the `system_server` command. I was in fact very skeptical of my own reading

1. Interestingly, a new `ro.config.headless` global property has been added to the official AOSP releases since 4.1/Jelly Bean. That property appears to allow the execution of the stack without a user interface.

of the sources on that matter and posted a question about it to the android-building mailing list. The ensuing **response** seems to confirm my reading of the sources, however.

In addition to the Surface Flinger and the system services started by the System Server, another set of system services stems from the starting of *mediaserver*. Here's the relevant snippet from 2.3/Gingerbread's *init.rc* (4.2/Jelly Bean's is practically identical):

```
service media /system/bin/mediaserver
    user media
    group system audio camera graphics inet net_bt net_bt_admin net_raw
    ioprio rt 4
```

The *mediaserver*, whose sources are in *frameworks/base/media* in 2.3/Gingerbread and *frameworks/av/media* in 4.2/Jelly Bean, starts the following system services: Audio Flinger, Media Player Service, Camera Service, and Audio Policy Service. Again, none of this is configurable, and it's recommended that you leave the relevant *init.rc* portions untouched unless you fully understand the implications of your modifications. For instance, if you try to remove the startup of the *mediaplayer* service from *init.rc* or use the *disabled* option to prevent it from starting, you will notice messages such as these in *logcat*'s output:

```
...
I/ServiceManager( 56): Waiting for service media.audio_policy...
I/ServiceManager( 56): Waiting for service media.audio_policy...
I/ServiceManager( 56): Waiting for service media.audio_policy...
W/AudioSystem( 56): AudioPolicyService not published, waiting...
I/ServiceManager( 56): Waiting for service media.audio_policy...
I/ServiceManager( 56): Waiting for service media.audio_policy...
...
```

And the system will hang and continue to print out those messages until the *media-server* is started.

Note that the *mediaserver* is one of the only init services that uses the *io prio* option. Presumably—and there's unfortunately no official documentation to confirm this—this is used to make sure that media playback has an appropriate priority to avoid choppy playback.

There is finally one odd player in this game, **the Phone app, which provides the Phone** system service. Generally speaking, apps are the wrong place to put system services because apps are lifecycle managed and can therefore be stopped and restarted at will. System services, on the other hand, are supposed to live from boot to reboot and cannot therefore be stopped midstream without affecting the rest of the system. The Phone app is different, however, because its manifest file has the `android:persistent` property of the `application` XML element set to `true`. This indicates to the system that this app should not be lifecycle managed, which therefore enables it to house a system service.

It will also lead to this app being automatically started as part of the initialization of the Activity Manager.

Again, there's nothing typically configurable about the Phone app's startup. You can, however, relatively easily remove the Phone app from the list of apps built into the AOSP. The result, however, will be that any part of the system depending on that system service will fail to function correctly. Again, you might as well leave it in. If you want to remove the dialer icon from the home screen, then what you actually want to remove is the Contacts app. As counterintuitive as it may sound, the typical phone dialer Android users are accustomed to isn't part of the Phone app; it's part of the Contacts app.



Another example of an app that houses a system service is the NFC app found in `packages/apps/Nfc/`.

The Phone app way of providing a system service is very interesting, because it opens the door for us to emulate its example and to add system services as apps within our own `device/acme/coyotepad/` directory—without having to modify the sources of the default system services in `frameworks/base/services/`.

Boot Animation

As I explained when discussing the boot logo in the previous chapter, Android's LCD goes through four stages during boot. One of those is a boot animation. Here's the corresponding entry in 2.3/Gingerbread's `init.rc` (the one in 4.2/Jelly Bean is practically identical):

```
service bootanim /system/bin/bootanimation
    user graphics
    group graphics
    disabled
    oneshot
```

Notice that this service is marked as `disabled`. Hence, `init` won't actually start this right away. Instead, it must be explicitly started somewhere else. In this case, it's the Surface Flinger that actually starts the boot animation *after* it has finished its own initialization by setting the `ctl.start` global property. Here's code from the `SurfaceFlinger::readyToRun()` function in 2.3/Gingerbread's `frameworks/base/services/surfaceflinger/SurfaceFlinger.cpp`:

```
// start boot animation
property_set("ctl.start", "bootanim");
```

The code in 4.2/Jelly Bean's `frameworks/native/services/surfaceflinger/SurfaceFlinger.cpp` does effectively the same thing:

```

...
void SurfaceFlinger::startBootAnim() {
    // start boot animation
    property_set("service.bootanim.exit", "0");
    property_set("ctl.start", "bootanim");
}
...
status_t SurfaceFlinger::readyToRun()
{
    ...

    // start boot animation
    startBootAnim();

    return NO_ERROR;
}
...

```

And given that the Surface Flinger is one of the first system services started—if not the first—the boot animation ends up continuously displaying while critical parts of the system are initializing. Typically, it will stop only when the phone’s home screen finally comes to the fore. We’ll take a look at some of the things happening during the boot animation shortly.

As you can see in the previous *init.rc* snippet, the `bootanim` service corresponds to the *bootanimation* binary. The latter’s sources are in *frameworks/base/cmds/bootanimation/*, and if you dig into them you’ll notice that this utility talks directly through Binder to the Surface Flinger in order to render its animation; hence the need for the Surface Flinger to be live before the animation can start. **Figure 7-1** illustrates the default Android boot animation displayed by *bootanimation*, with the moving light reflection projected on the Android logo.



Figure 7-1. Default boot animation

bootanimation actually has two modes of operation. In one mode it creates the default Android logo boot animation using the images in *frameworks/base/core/res/assets/images/*. It's likely best not to try modifying the boot animation by touching these files. Instead, by providing either */data/local/bootanimation.zip* or */system/media/bootanimation.zip*, you will force *bootanimation* to enter its other mode of operation, where it uses the content of one of those ZIP files to render a boot animation. It's worth taking some time to see how that can be done, even though a book is not the ideal medium for illustrating a running animation.

bootanimation.zip

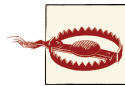
The *bootanimation.zip* is a regular, **uncompressed** ZIP file with at least a *desc.txt* file at the top-level directory inside and a bunch of directories containing PNG files. The latter are animated in sequence according to the rules in the *desc.txt* file. Note that *bootanimation* doesn't support anything but PNG files. Here are the semantics of the *desc.txt* file:

```
<width> <height> <fps>
p <count> <pause> <path>
p <count> <pause> <path>
```

Note that the file's format is very simplistic and doesn't allow for any fluff. So stick to the above semantics as is. The first line indicates the width, height, and frame rate (frames per second) for the animation. Each subsequent line is a *part* of the animation. For each part, you must provide the number of times this part is played (count), the number of frames to pause after each time the part is played (pause), and the directory where that part of the animation is located (path). Parts are played in the order they appear in the *desc.txt*.

Each animation part, and therefore the associated directory, is made of several PNG files, with filenames being a string representing the sequential number of that frame in the full sequence. Files could, for instance, be named *001.png*, *002.png*, *003.png*, etc. If the count is set to zero, the part will loop playing until the system has finished booting and the Launcher starts. Typically, initial parts are likely to have a count of 1, while the last part usually has a count of 0, so it continues playing until the boot is done.

The best way to create your own boot animation is to look at the existing *bootanimation.zip* files that have been created by others. If you look for that filename in your favorite search engine, you should find a few examples relatively easily. Have a look, for example, at some of the latest boot animations created for the [CyanogenMod](#) aftermarket Android distribution.



Again, make sure the ZIP file you created isn't compressed. Otherwise it won't work. Have a look at the *zip* command's man page—especially the `-0` flag.

Disabling the boot animation

You can also outright disable the boot animation if you don't want it. Just use the *setprop* command in *init.rc* to set the `debug.sf.nobootanimation` to 1:

```
setprop debug.sf.nobootanimation 1
```

In this case, the screen will go black at some point after the boot logo has been displayed, and stay black until the Launcher app displays the home screen.

Dex Optimization

One of the system services started during the boot animation is the Package Manager. We haven't covered its functionality in detail, but suffice it to say that the Package Manager manages all the *.apks* in the system. Among other things, it'll deal with the installation and removal of *.apks* and help the Activity Manager resolve intents.

One of the Package Manager's responsibilities is also to make sure that JIT-ready versions of any DEX byte-code are available prior to the corresponding Java code ever executing. To achieve this, the Package Manager's constructor (the Package Manager

system service is implemented as a Java class) goes through all *.apk* and *.jar* files in the system and requests that *installd* run the *dexopt* command on them.

This process should happen on the first boot only. Subsequently, the */data/dalvik-cache* directory will contain JIT-ready versions of all *.dex* files, and the boot sequence should be substantially faster. If you look into *logcat*'s output at first boot, you'll actually see entries like these:

```
D/dalvikvm( 32): DexOpt: --- BEGIN 'core.jar' (bootstrap=1) ---
D/dalvikvm( 62): Ignoring duplicate verify attempt on Ljava/lang/Object;
D/dalvikvm( 62): Ignoring duplicate verify attempt on Ljava/lang/Class;
D/dalvikvm( 62): DexOpt: load 349ms, verify+opt 4153ms
D/dalvikvm( 32): DexOpt: --- END 'core.jar' (success) ---
D/dalvikvm( 32): DEX prep '/system/framework/core.jar': unzip in 405ms, rewrite 5337ms
D/dalvikvm( 32): DexOpt: --- BEGIN 'bouncycastle.jar' (bootstrap=1) ---
D/dalvikvm( 63): DexOpt: load 54ms, verify+opt 779ms
D/dalvikvm( 32): DexOpt: --- END 'bouncycastle.jar' (success) ---
D/dalvikvm( 32): DEX prep '/system/framework/bouncycastle.jar': unzip in 48ms, rewrite 1023ms
D/dalvikvm( 32): DexOpt: --- BEGIN 'ext.jar' (bootstrap=1) ---
D/dalvikvm( 64): DexOpt: load 129ms, verify+opt 1497ms
D/dalvikvm( 32): DexOpt: --- END 'ext.jar' (success) ---
D/dalvikvm( 32): DEX prep '/system/framework/ext.jar': unzip in 91ms, rewrite 1923ms
...
D/installd( 35): DexInv: --- BEGIN '/system/framework/am.jar' ---
D/dalvikvm( 95): DexOpt: load 15ms, verify+opt 58ms
D/installd( 35): DexInv: --- END '/system/framework/am.jar' (success) ---
D/installd( 35): DexInv: --- BEGIN '/system/framework/input.jar' ---
D/dalvikvm( 96): DexOpt: load 5ms, verify+opt 28ms
D/installd( 35): DexInv: --- END '/system/framework/input.jar' (success) ---
D/installd( 35): DexInv: --- BEGIN '/system/framework/pm.jar' ---
D/dalvikvm( 97): DexOpt: load 12ms, verify+opt 64ms
D/installd( 35): DexInv: --- END '/system/framework/pm.jar' (success) ---
...
D/installd( 35): DexInv: --- BEGIN '/system/app/ApplicationsProvider.apk' ---
D/dalvikvm( 249): DexOpt: load 31ms, verify+opt 110ms
D/installd( 35): DexInv: --- END '/system/app/ApplicationsProvider.apk' (success) ---
D/installd( 35): DexInv: --- BEGIN '/system/app/UserDictionaryProvider.apk' ---
D/dalvikvm( 253): DexOpt: load 19ms, verify+opt 52ms
D/installd( 35): DexInv: --- END '/system/app/UserDictionaryProvider.apk' (success) ---
D/installd( 35): DexInv: --- BEGIN '/system/app/Settings.apk' ---
D/dalvikvm( 254): DexOpt: load 155ms, verify+opt 894ms
D/installd( 35): DexInv: --- END '/system/app/Settings.apk' (success) ---
D/installd( 35): DexInv: --- BEGIN '/system/app/Launcher2.apk' ---
D/dalvikvm( 256): DexOpt: load 178ms, verify+opt 581ms
D/installd( 35): DexInv: --- END '/system/app/Launcher2.apk' (success) ---
```

At first, the Package Manager Service isn't yet running, so we can see Dalvik running *dexopt* directly for some *.jar* files instead of being run by *install*d, as happens when the Package Manager Service requests it. Once the Package Manager is started, it then runs the rest of this optimization process in the following order:

1. *.jar* files listed in the `BOOTCLASSPATH` variable in *init.rc*
2. *.jar* files listed as libraries in */system/etc/permissions/platform.xml*
3. *.apk* and *.jar* files found in */system/framework*
4. *.apk* files found in */system/app*
5. *.apk* files found in */vendor/app*
6. *.apk* files found in */data/app*
7. *.apk* files found in */data/app-private*

Obviously this process takes some time. On my quad-core CORE i7, it takes the emulator image of a freshly compiled 2.3/Gingerbread AOSP 75 seconds for its first full boot (i.e., up to the home screen) and 24 seconds for subsequent boots. In a production system, such as a phone, boot times like this can be unacceptable.

You'll therefore be happy to hear that you can actually stop this optimization process from happening at boot time and do it at build time instead. You just need to set the `WITH_DEXPREOPT` build flag to `true` when building the AOSP:

```
$ make WITH_DEXPREOPT=true -j16
```

You can also set this variable in your device's *BoardConfig.mk* instead, and avoid having to add it to the *make* command every time. In the case of the emulator build, this wasn't done by default in 2.3/Gingerbread but is in 4.2/Jelly Bean.

The build will of course take more time, but the first boot will be significantly faster. On the same workstation mentioned previously, it takes 30 minutes to build 2.3/Gingerbread instead of 20 with the `WITH_DEXPREOPT` flag. However, the emulator image comes up in 40 seconds instead of 75 on a first boot. When the option is used, the */data/dalvik-cache* directory ends up being empty on the target after the first boot. Instead, at build time, *.odex* files are placed side by side in the same filesystem path as their corresponding *.jar* and *.apk* files.

Apps Startup

As the startup of the system services nears its end, apps start to get activated, including the home screen. As I explained in [Chapter 2](#), the Activity Manager ends its initialization by sending an intent of type `Intent.CATEGORY_HOME`, which causes the Launcher app to start and the home screen to appear. That's only part of the story, though. The startup

of the system services will in fact cause quite a few apps to start. Here's a portion of the output of the *ps* command on a freshly booted 2.3/Gingerbread emulator image:

```
# ps
...
root      32    1    60832 16240 c009b74c afd0b844 S zygote
media     33    1    17976 1056  ffffffff afd0b6fc S /system/bin/mediaserver
bluetooth 34    1    1256  220  c009b74c afd0c59c S /system/bin/dbus-daemon
root      35    1    812   232  c02181f4 afd0b45c S /system/bin/install-d
keystore  36    1    1744  212  c01b52b4 afd0c0cc S /system/bin/keystore
root      38    1    824   268  c00b8fec afd0c51c S /system/bin/qemud
shell     40    1    732   200  c0158eb0 afd0b45c S /system/bin/sh
root      41    1    3364  168  ffffffff 00008294 S /sbin/adbd
system    61   32   124096 26352 ffffffff afd0b6fc S system_server
app_19    113  32    80336 17400 ffffffff afd0c51c S com.android.inputmethod.
                                latin
radio     121  32    87112 17972 ffffffff afd0c51c S com.android.phone
system    122  32    73160 18452 ffffffff afd0c51c S com.android.systemui
app_26    132  32    76608 20812 ffffffff afd0c51c S com.android.launcher
app_1     169  32    85368 20584 ffffffff afd0c51c S android.process.acore
app_12    234  32    70752 15748 ffffffff afd0c51c S com.android.quicksearchbox
app_8     242  32    73108 16908 ffffffff afd0c51c S android.process.media
app_10    266  32    70928 16572 ffffffff afd0c51c S com.android.providers.
                                calendar
app_29    300  32    72764 17484 ffffffff afd0c51c S com.android.email
app_18    315  32    70272 15428 ffffffff afd0c51c S com.android.music
app_22    323  32    69712 15220 ffffffff afd0c51c S com.android.protips
app_3     335  32    71432 16756 ffffffff afd0c51c S com.cooliris.media
...
```

All the processes that have a Java-style process name² are actually apps that were automatically started with no user intervention whatsoever at system startup. Various system mechanisms cause these apps to start given the content of their respective manifest files. And this is a welcome change, since controlling apps' activation requires a lot less internals work than is required for controlling many other aspects of the startup, as we've seen above. Instead, it's all about creating carefully crafted apps for packaging with the AOSP. Sure, there's the case where you'll want to modify a stock app to make it behave or start differently, but at least we're into the app world, where functionality is more loosely coupled and documentation more readily accessible.

Which leads us to discussing the triggers that cause stock apps to be activated.

Input methods

One of the earliest types of apps to start are input methods. The Input Method Manager Service's constructor goes around and activates all app services that have an intent filter

2. These are dot-separated names, such as `com.android.launcher` for the Launcher app, for example.

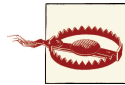
for `android.view.InputMethod`. This is how, for example, the LatinIME app, which runs as the `com.android.inputmethod.latin` process, is activated.

As you can see by reading the [Creating an Input Method](#) blog post on the Android Developers Blog, input methods are actually carefully crafted services.

Persistent apps

Apps that have the `android:persistent="true"` attribute in the `<application>` element of their manifest file will be automatically spawned at startup by the Activity Manager. In fact, should such an app ever die, it will also be automatically restarted by the Activity Manager.

As I explained earlier, unlike regular apps, apps that are marked as persistent are not lifecycle managed by the Activity Manager. Instead, they are kept alive throughout the lifetime of the system. This allows using such apps to implement special functionality. The status bar and the phone app, for example, running as the `com.android.system ui` and `com.android.phone` processes, are persistent apps.



While the app development documentation does explain the role of `android:persistent`, the use of that attribute is reserved for apps that are built within the AOSP.

Home screen

Typically there's only one home screen app, and it reacts to the `Intent.CATEGORY_HOME` intent, which is sent by the Activity Manager at the end of the system services' startup. There's a sample home app in `development/samples/Home/`, but the real home app activated is in `packages/apps/Launcher2/`. Here's the Launcher's main activity and its intent filter in 2.3/Gingerbread (4.2/Jelly Bean's is basically the same):

```
<activity
    android:name="com.android.launcher2.Launcher"
    android:launchMode="singleTask"
    android:clearTaskOnLaunch="true"
    android:stateNotNeeded="true"
    android:theme="@style/Theme"
    android:screenOrientation="nosensor"
    android:windowSoftInputMode="stateUnspecified|adjustPan">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.HOME" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.MONKEY"/>
    </intent-filter>
</activity>
```


Obviously, if you want to start a custom app to be the home screen instead of Launcher2, you'll need to remove the latter and add your own that reacts to that same intent. If more than one app reacts to that intent, users will get a dialog asking them which of the home screens they want to use.

Note that this intent isn't sent just at startup. Depending on the state of the system, the Activity Manager will send this intent whenever it needs to bring the home screen to the foreground.

BOOT_COMPLETED intent

The Activity Manager also broadcasts the `Intent.BOOT_COMPLETED` intent at startup. This is an intent commonly used by apps to be notified that the system has finished booting. A number of stock apps in the AOSP actually rely on this intent, such as Media provider, Calendar provider, Mms app, and Email app. Here's the broadcast receiver used by the Media Provider in 2.3/Gingerbread, along with its intent filter (4.2/Jelly Bean's is very similar):

```
<receiver android:name="MediaScannerReceiver">
  <intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED" />
  </intent-filter>
  <intent-filter>
    <action android:name="android.intent.action.MEDIA_MOUNTED" />
    <data android:scheme="file" />
  </intent-filter>
  <intent-filter>
    <action android:name="android.intent.action.MEDIA_SCANNER_SCAN_FILE" />
    <data android:scheme="file" />
  </intent-filter>
</receiver>
```

In order to receive this intent, apps must explicitly request permission to do so:

```
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
```

APPWIDGET_UPDATE intent

In addition to apps, the App Widget Service, which is itself a system service, registers itself to receive the `Intent.BOOT_COMPLETED`. It uses the receipt of that intent as a trigger to activate all app widgets in the system by sending `Intent.APPWIDGET_UPDATE`. Hence, if you've developed an app widget as part of your app, your code will be activated at this point. Have a look at the [App Widgets](#) section of the Android developer documentation for more information on how to write your own app widget.

Several stock AOSP apps have app widgets, such as Quick Search Box, Music, Protips, and Media. Here's the Quick Search Box's app widget declaration in its manifest file, for example:

```

<receiver android:name=".SearchWidgetProvider"
    android:label="@string/app_name">
    <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_
            UPDATE" />
    </intent-filter>
    <meta-data android:name="android.appwidget.provider" android:
        resource="@xml/search_widget_info" />
</receiver>

```

Utilities and Commands

Once the Framework and the basic set of apps is up and running, there are quite a few commands that you can use to query or interact with system services and the Framework. Much like the commands covered in [Chapter 6](#), these can be used on the command line once you shell into the device. But these commands have no meaning, and therefore no effect, unless the Framework is running. Of course you'll find many of these useful, even crucial, as you're bringing up Android on new devices and/or debugging parts of the Framework. And as with the commands in the native user-space, the tools available for interacting with the Framework vary greatly in terms of documentation and capabilities. Yet they provide the essential capabilities required to bring Android up on new hardware or to troubleshoot it on existing products. Let's take a look at the command set available to you for interacting with the Android Framework.



Many of the commands here are located in the *frameworks/base/cmds/* directory of the AOSP sources, though in 4.2/Jelly Bean, some of those commands have been moved to *frameworks/native/cmds/*. I encourage you to refer to those sources when using some of these commands, as their effects aren't always obvious just by looking at their online help, when it exists.

General-Purpose Utilities

In contrast with some utilities we'll see later, a certain number of utilities are useful for interacting with various parts of the Framework. Some of these are very powerful.

service

The *service* command allows us to interact with any system service registered with the Service Manager:

```

# service -h
Usage: service [-h|-?]
       service list
       service check SERVICE
       service call SERVICE CODE [i32 INT | s16 STR] ...

```

Options:

- i32: Write the integer INT into the send parcel.
- s16: Write the UTF-16 string STR into the send parcel.

As you can see, it can be used for querying but can also be used for invoking methods from system services. Here's how it can be used to query the list of existing system services in 2.3/Gingerbread:

```
# 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 ui mode: [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]
...
```

The interface names provided in between square brackets allow you to browse the AOSP sources to find the matching *.aidl* file that defines the interface.

You can also check if a given service exists:

```
# service check power
Service power: found
```

Most interestingly, you can use *service call* to directly invoke system services' Binder-exposed methods. In order to do that, you first need to understand that service's interface. Here's the *IStatusBarService* interface definition from 2.3/Gingerbread's *frameworks/base/core/java/com/android/internal/statusbar/IStatusBarService.aidl* (4.2/Jelly Bean's interface name is the same, though *setIcon()*'s prototype has changed):

```
...
interface IStatusBarService
{
    void expand();
    void collapse();
    void disable(int what, IBinder token, String pkg);
    void setIcon(String slot, String iconPackage, int iconId, int iconLevel);
    ...
}
```

Note that *service call* actually needs a method code, not a method's name. To find the codes matching the method names defined in the interface, you'll need to look up the code generated by the *aidl* tool based on the interface definition. Here's the relevant snippet from the *IStatusBarService.java* file generated in *out/target/common/obj/JAVA_LIBRARIES/framework_intermediates/src/core/java/com/android/internal/statusbar/*:

```
...
static final int TRANSACTION_expand = (android.os.IBinder.FIRST_CALL_
TRANSACTION + 0);
static final int TRANSACTION_collapse = (android.os.IBinder.FIRST_CALL_
TRANSACTION + 1);
static final int TRANSACTION_disable = (android.os.IBinder.FIRST_CALL_
TRANSACTION + 2);
static final int TRANSACTION_setIcon = (android.os.IBinder.FIRST_CALL_
TRANSACTION + 3);
...
```

Also, note that *frameworks/base/core/java/android/os/IBinder.java* has the following definition for *FIRST_CALL_TRANSACTION*:

```
int FIRST_CALL_TRANSACTION = 0x00000001;
```

Hence, *expand()*'s code is 1 and *collapse()*'s code is 2. Therefore, this command will cause the status bar to expand:

```
# service call statusbar 1
```

While this command will cause the status bar to collapse:

```
# service call statusbar 2
```

This is a very simple case where the action is rather obvious and the methods invoked don't take any parameters. In other cases, you'll need to look more closely at the system service's API and understand the parameters expected. In addition, keep in mind that system services' interfaces aren't necessarily exposed through *.aidl* files. In some cases, such as for the Activity Manager, the interface definition is hardcoded directly into a regular Java file instead of being autogenerated. And in the case of C-based system services, the Binder marshaling and unmarshaling is all done straight in C code. Hence, try using *grep* on the AOSP's *frameworks/* directory in addition to *out/target/common/* to find all instances of *FIRST_CALL_TRANSACTION*.


dumpsys

Another interesting thing to do is to query system services' internal state. Indeed, every system service implements a *dump()* method internally that can be queried using the *dumpsys* command:

```
dumpsys [ <service> ]
```

By default, if no system service name is provided as a parameter, *dumpsys* will first print out the list of system services and will then dump their status:

```
# dumpsys
Currently running services:
  SurfaceFlinger
  accessibility
  account
  activity
  alarm
  appwidget
  audio
  backup
  battery
  batteryinfo
  clipboard
  connectivity
  content
  cpuinfo
  device_policy
  devicestoragemonitor
  diskstats
  dropbox
  entropy
  hardware
  ...
-----
DUMP OF SERVICE SurfaceFlinger:
+ Layer 0x1e5788
  z= 21000, pos=( 0, 0), size=( 320, 480), needsBlending=0, needsDithering=0, invalidate=0, alpha=0xff, flags=0x00000000, tr=[1.00, 0.00][0.00, 1.00]
  name=com.android.internal.service.wallpaper.ImageWallpaper
  client=0x1ed2a8, identity=3
  [ head= 1, available= 2, queued= 0 ] reallocMask=00000000, identity=3, status=0
  format= 4, [320x480:320] [320x480:320], freezeLock=0x0, bypass=0, dq-q-time=2034 us
  Region transparentRegion (this=0x1e5918, count=1)
    [ 0, 0, 0, 0]
  Region transparentRegionScreen (this=0x1e57bc, count=1)
    [ 0, 0, 0, 0]
  Region visibleRegionScreen (this=0x1e5798, count=1)
    [ 0, 25, 320, 480]
+ Layer 0x268b70
  z= 21005, pos=( 0, 0), size=( 320, 480), needsBlending=1, needsDithering=1, invalidate=0, alpha=0xff, flags=0x00000000, tr=[1.00, 0.00][0.00, 1.00]
  ...
-----
DUMP OF SERVICE accessibility:
-----
DUMP OF SERVICE account:
Accounts: 0
```



```

Active Sessions: 0

RegisteredServicesCache: 1 services
  ServiceInfo: AuthenticatorDescription {type=com.android.exchange}, ComponentInfo{com.android.email/com.android.email.service.EasAuthenticatorService}, uid 10029
-----
DUMP OF SERVICE activity:
Providers in Current Activity Manager State:
  Published content providers (by class):
    * ContentProviderRecord{4060d0e0 com.android.deskclock.AlarmProvider}
...

```

Obviously the output is very verbose and, most importantly, requires understanding the corresponding system service's internals. If you're implementing your own system service, however, being able to query its state at runtime can be crucial. Of course, if you're not interested in dumping the state of all system services, you just need to provide the name of the specific service you'd like to get information about as a parameter to *dumpsys*:

```

# dumpsys power
Power Manager State:
  mIsPowered=true mPowerState=1 mScreenOffTime=46793204 ms
  mPartialCount=1
  mWakeLockState=SCREEN_ON_BIT
  mUserState=
  mPowerState=SCREEN_ON_BIT
  mLocks.gather=SCREEN_ON_BIT
  mNextTimeout=94351 now=46880555 -46786s from now
  mDimScreen=true mStayOnConditions=1
  mScreenOffReason=0 mUserState=0
  mBroadcastQueue={-1,-1,-1}
  mBroadcastWhy={0,0,0}
  mPokey=0 mPokeAwakeonSet=false
...

```

dumpstate

In some cases, what you're trying to do is get a snapshot of the entire system, not just the system services. This is what *dumpstate* takes care of. In fact, you might recall our discussion of this command when we covered *adb's bugreport* in [Chapter 6](#), since *dumpstate* is what provides *bugreport* with its information. Here's *dumpstate's* detailed help in 2.3/Gingerbread:

```

# dumpstate -h
usage: dumpstate [-d] [-o file] [-s] [-z]
  -d: append date to filename (requires -o)
  -o: write to file (instead of stdout)
  -s: write output to control socket (for init)
  -z: gzip output (requires -o)

```

In 4.2/Jelly Bean, *dumpstate*'s capabilities have expanded:

```
root@android:/ # dumpstate -h
usage: dumpstate [-b soundfile] [-e soundfile] [-o file [-d] [-p] [-z]] [-s] [-q]
  -o: write to file (instead of stdout)
  -d: append date to filename (requires -o)
  -z: gzip output (requires -o)
  -p: capture screenshot to filename.png (requires -o)
  -s: write output to control socket (for init)
  -b: play sound file instead of vibrate, at beginning of job
  -e: play sound file instead of vibrate, at end of job
  -q: disable vibrate
```

If you invoke it without any parameters, it goes ahead and queries several parts of the system to provide you with a complete snapshot of the system's status:

```
# dumpstate
=====
== dumpstate: 2012-10-10 03:15:26
=====

Build: generic-eng 2.3.4 GINGERBREAD eng.karim.20120913.141233 test-keys
Bootloader: unknown
Radio: unknown
Network: Android
Kernel: Linux version 2.6.29-00261-g0097074-dirty (digit@digit.mtv.corp.google.com) (gcc version 4.4.0 (GCC) ) #20 Wed Mar 31 09:54:02 PDT 2010
Command line: qemu=1 console=ttyS0 android.checkjni=1 android.qemud=ttyS1 android.ndns=1

----- MEMORY INFO (/proc/meminfo) -----
MemTotal:          94096 kB
MemFree:            1296 kB
Buffers:             0 kB
Cached:            32424 kB
...
----- CPU INFO (top -n 1 -d 1 -m 30 -t) -----

User 2%, System 11%, IOW 33%, IRQ 0%
User 3 + Nice 0 + Sys 15 + Idle 67 + IOW 42 + IRQ 0 + SIRQ 0 = 127

  PID   TID CPU% S    VSS    RSS PCY UID      Thread      Proc
  339   339  13% R   976K   440K  fg  shell    top         top
  121   121   0% S  86100K 18484K  fg  radio    m.android.phone com.android.phone
    3     3   0% S     0K    0K  fg  root    ksoftirqd/0
    4     4   0% S     0K    0K  fg  root    events/0
...
----- PROCRAK (procrank) -----
  PID     Vss     Rss     Pss     Uss  cmdline
   61  25676K  25076K  10581K  8552K  system_server
  124  21412K  21412K   6851K  4908K  com.android.launcher
```

```

122 19268K 19268K 5698K 4388K com.android.systemui
121 18484K 18484K 4744K 3568K com.android.phone
295 18176K 18176K 4337K 3132K com.android.email
115 17836K 17836K 4118K 2960K com.android.inputmethod.latin
...
----- VIRTUAL MEMORY STATS (/proc/vmstat) -----
nr_free_pages 553
nr_inactive_anon 6708
nr_active_anon 6068
nr_inactive_file 3449
nr_active_file 2062
...
----- VMALLOC INFO (/proc/vmallocinfo) -----
0xc684c000-0xc684e000 8192 __arm_ioremap_pfn+0x68/0x2fc ioremap
0xc6850000-0xc6852000 8192 __arm_ioremap_pfn+0x68/0x2fc ioremap
0xc6854000-0xc6856000 8192 __arm_ioremap_pfn+0x68/0x2fc ioremap
0xc6880000-0xc68a1000 135168 binder_mmap+0xb4/0x200 ioremap
...
----- SLAB INFO (/proc/slabinfo) -----
slabinfo - version: 2.1
# name <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab>
: tunables <limit> <batchcount> <sharedfactor> : slabdata <active_slabs> <num_s
labs> <sharedavail>
rpc_buffers 8 8 2048 2 1 : tunables 24 12 0 : sla
bdata 4 4 0
rpc_tasks 8 24 160 24 1 : tunables 120 60 0 : sla
bdata 1 1 0
rpc_inode_cache 0 0 416 9 1 : tunables 54 27 0 : sla
bdata 0 0 0
bridge_fdb_cache 0 0 64 59 1 : tunables 120 60 0 : sla
bdata 0 0 0
...
----- ZONEINFO (/proc/zoneinfo) -----
Node 0, zone Normal
pages free 550
min 312
low 390
high 468
scanned 0 (aa: 0 ia: 0 af: 26 if: 0)
...
----- SYSTEM LOG (logcat -v time -d *:v) -----
10-10 01:38:02.762 I/DEBUG ( 30): debuggerd: Feb 26 2012 21:06:53
10-10 01:38:02.882 I/Netd ( 29): Netd 1.0 starting
10-10 01:38:02.932 D/qemud ( 38): entering main loop
10-10 01:38:02.972 I/Vold ( 28): Vold 2.1 (the revenge) firing up
10-10 01:38:02.972 D/Vold ( 28): USB mass storage support is not enabled in
the kernel
...
----- VM TRACES JUST NOW (/data/anr/traces.txt.bugreport: 2012-10-10 03:15:26)
-----

```



```

----- pid 61 at 2012-10-10 03:15:26 -----
Cmd line: system_server

DALVIK THREADS:
(mutexes: tll=0 tsl=0 tscl=0 ghl=0 hwl=0 hwl=0)
"main" prio=5 tid=1 NATIVE
  | group="main" sCount=1 dsCount=0 obj=0x4001f1a8 self=0xce48
  | sysTid=61 nice=0 sched=0/0 cgrp=default handle=-1345006528
  | schedstat=( 1116789165 392598071 782 )
  at com.android.server.SystemServer.init1(Native Method)
  at com.android.server.SystemServer.main(SystemServer.java:625)
...
----- EVENT LOG (logcat -b events -v time -d *:v) -----
10-10 01:38:03.642 I/boot_progress_start( 32): 5126
10-10 01:38:04.221 I/boot_progress_preload_start( 32): 5706
10-10 01:38:04.251 I/dvm_gc_info( 32): [8825198673194415294,-90644969689662529
97,-4012584086963399109,0]
10-10 01:38:04.281 I/dvm_gc_info( 32): [8825198673194406507,-92148046065296736
57,-4012584086963329465,0]
10-10 01:38:04.331 I/dvm_gc_info( 32): [8825198673194406993,-91348657131437777
12,-4012584086963259824,0]
10-10 01:38:04.371 I/dvm_gc_info( 32): [8825198673194415172,-91399322627244589
19,-4012584086963149223,0]
...
----- RADIO LOG (logcat -b radio -v time -d *:v) -----
10-10 01:58:04.988 D/AT ( 31): AT< +CSQ: 7,99
10-10 01:58:04.988 D/AT ( 31): AT< OK
10-10 01:58:04.988 D/RILJ ( 121): [0114]< SIGNAL_STRENGTH {7, 99, 0, 0, 0
, 0, 0}
10-10 01:58:24.998 D/RILJ ( 121): [0115]> SIGNAL_STRENGTH
10-10 01:58:25.008 D/RIL ( 31): onRequest: SIGNAL_STRENGTH
...
----- NETWORK INTERFACES (netcfg) -----
*** exec(netcfg): Permission denied
*** netcfg: Exit code 255
[netcfg: 0.1s elapsed]

----- NETWORK ROUTES (/proc/net/route) -----
Iface Destination Gateway Flags RefCnt Use Metric Mask
MTU Window IRTT
eth0 0002000A 00000000 0001 0 0 0 00FFFFFF
0 0 0
eth0 00000000 0202000A 0003 0 0 0 00000000
0 0 0

----- ARP CACHE (/proc/net/arp) -----
IP address HW type Flags HW address Mask Device
10.0.2.2 0x1 0x2 52:54:00:12:35:02 * eth0

----- SYSTEM PROPERTIES -----

```

```

[dalvik.vm.heapsize]: [16m]
[dalvik.vm.stack-trace-file]: [/data/anr/traces.txt]
[dev.bootcomplete]: [1]
[gsm.current.phone-type]: [1]
[gsm.defaultpdcontext.active]: [true]
...
----- KERNEL LOG (dmesg) -----
Initializing cgroup subsys cpu
Linux version 2.6.29-00261-g0097074-dirty (digit@digit.mtv.corp.google.com) (gcc
  version 4.4.0 (GCC) ) #20 Wed Mar 31 09:54:02 PDT 2010
CPU: ARM926EJ-S [41069265] revision 5 (ARMv5TEJ), cr=00093177
CPU: VIPT data cache, VIPT instruction cache
Machine: Goldfish
Memory policy: ECC disabled, Data cache writeback
On node 0 totalpages: 24576
...
----- KERNEL WAKELOCKS (/proc/wakelocks) -----

```

name	count	expire_count	wake_count	active_since	total_time
sleep_time		max_time	last_change		
"alarm"	106	0	0	1632946980	0
5822030632794					41697763
"KeyEvents"	27	0	0	0	123592046
27084159991					0
"event0-61"	26	0	0	0	48780811
27083608920					0
"radio-interface"	3	0	0	0	3472899963
1459986280		25362482435			0

```

...
----- KERNEL CPUFREQ (/sys/devices/system/cpu/cpu0/cpufreq/stats/time_in_state)
-----
*** /sys/devices/system/cpu/cpu0/cpufreq/stats/time_in_state: No such file or di
rectory

----- VOLD DUMP (vdc dump) -----
000 Dumping loop status
000 Dumping DM status
000 Dumping mounted filesystems
000 rootfs / rootfs ro 0 0
...
----- SECURE CONTAINERS (vdc asec list) -----
200 asec operation succeeded
[vdc: 0.1s elapsed]

----- PROCESSES (ps -P) -----

```

USER	PID	PPID	VSIZE	RSS	PCY	WCHAN	PC	NAME
root	1	0	268	180	fg	c009b74c	0000875c	S /init
root	2	0	0	0	fg	c004e72c	00000000	S kthreadd
root	3	2	0	0	fg	c003fdc8	00000000	S ksoftirqd/0
root	4	2	0	0	fg	c004b2c4	00000000	S events/0
root	5	2	0	0	fg	c004b2c4	00000000	S khelper
root	6	2	0	0	fg	c004b2c4	00000000	S suspend

```

...

```

```

----- PROCESSES AND THREADS (ps -t -p -P) -----
USER      PID  PPID  VSIZE  RSS   PRIO  NICE  RTPRI  SCHED  PCY  WCHAN    PC
NAME
root      1    0    268    180   20    0    0    0    fg  c009b74c 0000875c
S /init
root      2    0    0      0   15   -5    0    0    fg  c004e72c 00000000
S kthreadd
root      3    2    0      0   15   -5    0    0    fg  c003fdc8 00000000
S ksoftirqd/0
root      4    2    0      0   15   -5    0    0    fg  c004b2c4 00000000
S events/0
root      5    2    0      0   15   -5    0    0    fg  c004b2c4 00000000
S khelper
root      6    2    0      0   15   -5    0    0    fg  c004b2c4 00000000
S suspend
...
----- LIBRANK (librank) -----
RSStot    VSS      RSS      PSS      USS  Name/PID
16658K
          6980K   6980K   3218K   2896K   /dev/ashmem/dalvik-heap
          5208K   5208K   1371K   1048K   system_server [61]
          5272K   5272K   1343K   1012K   com.android.launcher [124]
          5272K   5272K   1343K   1012K   com.android.phone [121]
...
----- BINDER FAILED TRANSACTION LOG (/sys/kernel/debug/binder/failed_transaction_log) -----
*** /sys/kernel/debug/binder/failed_transaction_log: No such file or directory

----- BINDER TRANSACTION LOG (/sys/kernel/debug/binder/transaction_log) -----
*** /sys/kernel/debug/binder/transaction_log: No such file or directory

----- BINDER TRANSACTIONS (/sys/kernel/debug/binder/transactions) -----
*** /sys/kernel/debug/binder/transactions: No such file or directory

----- BINDER STATS (/sys/kernel/debug/binder/stats) -----
*** /sys/kernel/debug/binder/stats: No such file or directory

----- BINDER STATE (/sys/kernel/debug/binder/state) -----
*** /sys/kernel/debug/binder/state: No such file or directory

----- FILESYSTEMS & FREE SPACE (df) -----
Filesystem      1K-blocks      Used Available Use% Mounted on
tmpfs            47048           32    47016    0% /dev
tmpfs            47048           0    47048    0% /mnt/asec
tmpfs            47048           0    47048    0% /mnt/obb
/dev/block/mtdblock0 65536      65536         0 100% /system
/dev/block/mtdblock1 65536     25292    40244   39% /data
/dev/block/mtdblock2 65536      1156    64380    2% /cache
[df: 0.1s elapsed]

----- PACKAGE SETTINGS (/data/system/packages.xml: 2012-10-10 01:38:16) -----
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<packages>

```

```

<last-platform-version internal="10" external="0" />
...
----- PACKAGE UID ERRORS (/data/system/uiderrors.txt: 2012-09-24 21:06:14) ----
--
2012-09-24 21:06: No settings file; creating initial state

----- LAST KMSG (/proc/last_kmsg) -----
*** /proc/last_kmsg: No such file or directory

----- LAST RADIO LOG (parse_radio_log /proc/last_radio_log) -----
*** exec(parse_radio_log): Permission denied
*** parse_radio_log: Exit code 255
[parse_radio_log: 0.1s elapsed]

----- LAST PANIC CONSOLE (/data/dontpanic/apanic_console) -----
*** /data/dontpanic/apanic_console: No such file or directory

----- LAST PANIC THREADS (/data/dontpanic/apanic_threads) -----
*** /data/dontpanic/apanic_threads: No such file or directory


----- BLOCKED PROCESS WAIT-CHANNELS -----
----- BACKLIGHTS -----
LCD brightness=*** /sys/class/leds/lcd-backlight/brightness: No such file or dir
ectory
Button brightness=*** /sys/class/leds/button-backlight/brightness: No such file
or directory
Keyboard brightness=*** /sys/class/leds/keyboard-backlight/brightness: No such f
ile or directory
ALS mode=*** /sys/class/leds/lcd-backlight/als: No such file or directory
LCD driver registers:
*** /sys/class/leds/lcd-backlight/registers: No such file or directory


=====
== Android Framework Services
=====
----- DUMPSYS (dumpsys) -----
Currently running services:
    SurfaceFlinger
...

```

In most cases, as you can see, *dumpstate* is in fact invoking other commands such as *logcat*, *dumpsys*, and *ps* to retrieve its information. As you can also see, the command is very verbose.

rawbu

In some cases, you may want to back up and later restore the contents of */data*. You can use the *rawbu* command to do that:

```

# rawbu help
Usage: rawbu COMMAND [options] [backup-file-path]

```

```
commands are:
  help          Show this help text.
  backup        Perform a backup of /data.
  restore       Perform a restore of /data.
options include:
  -h            Show this help text.
  -a            Backup all files.
```

The `rawbu` command allows you to perform low-level backup and restore of the `/data` partition. This is where all user data is kept, allowing for a fairly complete restore of a device's state. Note that because this is low-level, it will only work across builds of the same (or very similar) device software.

Here's how it can be used to create a backup:

```
# rawbu backup /sdcard/backup.dat
Stopping system...
Backing up /data to /sdcard/backup.dat...
Saving dir /data/local...
Saving dir /data/local/tmp...
Saving dir /data/app-private...
Saving dir /data/app...
Saving dir /data/property...
Saving file /data/property/persist.sys.localevar...
Saving file /data/property/persist.sys.country...
Saving file /data/property/persist.sys.language...
Saving file /data/property/persist.sys.timezone...
...
Backup complete! Restarting system...
```

The first thing the command does is stop the *Zygote*, thereby stopping all system services. It then proceeds to copy everything from */data* and finishes by restarting the *Zygote*. Once data is backed up, you can restore it later:

```
# rawbu restore /sdcard/backup.dat
Stopping system...
Wiping contents of /data...
warning -- rmdir() error on '/data/system': Directory not empty
warning -- rmdir() error on '/data/system': Directory not empty
Restoring from /sdcard/backup.dat to /data...
Restoring dir /data/local...
Restoring dir /data/local/tmp...
Restoring dir /data/app-private...
Restoring dir /data/app...
...
Restore complete! Restarting system, cross your fingers...
```

Obviously, as the command's output implies, this is a fragile operation and you should be aware that results will vary.

Service-Specific Utilities

As we saw earlier, there are dozens of system services. Typically, using these system services requires writing code that interacts with their Binder-exposed API in some way, shape, or form. In some cases, however, the AOSP includes command-line utilities for directly interacting with certain system services. Some of these utilities are very powerful and allow us to tap into Android's functionality straight from the command line. This opens the door for using many of the following utilities as part of scripts either in production or during development.

Circumventing Android's Permission System

The system services' APIs are typically protected by Android's permission system, which requires apps' manifest files to declare upfront which permissions they require. Generally, a system service will check whether its caller has the appropriate permissions before going ahead and servicing the caller's request. Part of this checking will require checking the caller's PID and using the Package Manager's services to verify the originating *.apk*'s rights.

There is one case, however, that circumvents all safeguards: when the caller is running as root. Indeed, if you look at the permission-checking code of the Activity Manager, which is used by the other system services to check for permissions, you will see this snippet in *frameworks/base/services/java/com/android/server/am/ActivityManagerService.java* in 2.3/Gingerbread:

```
int checkComponentPermission(String permission, int pid, int uid,
...
    // Root, system server and our own process get to do everything.
    if (uid == 0 || uid == Process.SYSTEM_UID || pid == MY_PID ||
        !Process.supportsProcesses()) {
        return PackageManager.PERMISSION_GRANTED;
    }
...

```

In 4.2/Jelly Bean, you'll find this instead:

```
int checkComponentPermission(String permission, int pid, int uid,
...
    if (pid == MY_PID) {
        return PackageManager.PERMISSION_GRANTED;
    }

    return ActivityManager.checkComponentPermission(permission, uid,
        owningUid, exported);
}

```

With *ActivityManager.checkComponentPermission()* being defined as the following in *frameworks/base/core/java/android/app/ActivityManager.java*:



```

        public static int checkComponentPermission(String permission, int uid,
            int owningUid, boolean exported) {
            // Root, system server get to do everything.
            if (uid == 0 || uid == Process.SYSTEM_UID) {
                return PackageManager.PERMISSION_GRANTED;
            }
        }
    }
    ...

```

Hence, in both versions of the AOSP, any of the commands you see here that talk to a system service will typically be granted a green light on anything they ask for from a system service. You must, therefore, **be very careful** when talking to system services while running as root. The same applies if you write a command-line utility that mimics the way many of the commands we cover in this section interact with system services.

am

As I mentioned earlier, one of the most important system services is the Activity Manager. It should come as no surprise, therefore, that there's a command that allows us to directly invoke its functionality. Here's its online help in 2.3/Gingerbread:

```

# am
usage: am [subcommand] [options]

start an Activity: am start [-D] [-W] <INTENT>
    -D: enable debugging
    -W: wait for launch to complete

start a Service: am startservice <INTENT>

send a broadcast Intent: am broadcast <INTENT>

start an Instrumentation: am instrument [flags] <COMPONENT>
    -r: print raw results (otherwise decode REPORT_KEY_STREAMRESULT)
    -e <NAME> <VALUE>: set argument <NAME> to <VALUE>
    -p <FILE>: write profiling data to <FILE>
    -w: wait for instrumentation to finish before returning

start profiling: am profile <PROCESS> start <FILE>
stop profiling: am profile <PROCESS> stop

start monitoring: am monitor [--gdb <port>]
    --gdb: start gdbserve on the given port at crash/ANR

<INTENT> specifications include these flags:
    [-a <ACTION>] [-d <DATA_URI>] [-t <MIME_TYPE>]
    [-c <CATEGORY>] [-c <CATEGORY>] ...]
    [-e|--es <EXTRA_KEY> <EXTRA_STRING_VALUE> ...]
    [--esn <EXTRA_KEY> ...]
    [--ez <EXTRA_KEY> <EXTRA_BOOLEAN_VALUE> ...]
    [-e|--ei <EXTRA_KEY> <EXTRA_INT_VALUE> ...]
    [-n <COMPONENT>] [-f <FLAGS>]

```

```

[--grant-read-uri-permission] [--grant-write-uri-permission]
[--debug-log-resolution]
[--activity-brought-to-front] [--activity-clear-top]
[--activity-clear-when-task-reset] [--activity-exclude-from-recents]
[--activity-launched-from-history] [--activity-multiple-task]
[--activity-no-animation] [--activity-no-history]
[--activity-no-user-action] [--activity-previous-is-top]
[--activity-reorder-to-front] [--activity-reset-task-if-needed]
[--activity-single-top]
[--receiver-registered-only] [--receiver-replace-pending]
[<URI>]

```



In 4.2/Jelly Bean, *am*'s capabilities have expanded, and so, too, has its online help. Since the latter now covers three pages, it's impractical to print it in its entirety in this book. The previous snippet is sufficient for the present discussion; still, I encourage you to read the *am* command's online help in 4.2/Jelly Bean.

As we saw in [Chapter 2](#), there are four types of components available to app developers: activities, services, broadcast receivers, and content providers. The first three types of components are activated through intents, and one of *am*'s major features is its ability to send intents straight from the command line.

Here's how you can use *am* to get the browser to navigate to a given website along with the relevant log excerpts:

```

# am start -a android.intent.action.VIEW -d http://source.android.com
Starting: Intent { act=android.intent.action.VIEW dat=http://source.android.com }

# logcat
...
D/AndroidRuntime( 786):
D/AndroidRuntime( 786): >>>>> AndroidRuntime START com.android.internal.os.Run
timeInit <<<<<<
D/AndroidRuntime( 786): CheckJNI is ON
D/AndroidRuntime( 786): Calling main entry com.android.commands.am.Am
I/ActivityManager( 62): Starting: Intent { act=android.intent.action.VIEW dat=
http://source.android.com flg=0x10000000 cmp=com.android.browser/.BrowserActivit
y } from pid 786
I/ActivityManager( 62): Start proc com.android.browser for activity com.androi
d.browser/.BrowserActivity: pid=794 uid=10015 gids={3003, 1015}
D/AndroidRuntime( 786): Shutting down VM
D/dalvikvm( 786): GC_CONCURRENT freed 100K, 69% free 317K/1024K, external 0K/0K
, paused 1ms+1ms
D/jdwp ( 786): adbd disconnected
I/ActivityThread( 794): Pub browser: com.android.browser.BrowserProvider
I/BrowserSettings( 794): Selected search engine: ActivitySearchEngine{android.a
pp.SearchableInfo@40593270}
D/dalvikvm( 794): GC_CONCURRENT freed 447K, 51% free 2909K/5831K, external 934K

```



```

/1038K, paused 5ms+14ms
I/ActivityManager( 62): Displayed com.android.browser/.BrowserActivity: +1s924
ms
D/dalvikvm( 794): GC_EXTERNAL_ALLOC freed 51K, 50% free 2953K/5831K, external 9
51K/1038K, paused 62ms
...

```

That's a rather straightforward example. Let's look at something a little more customized. Here's a broadcast receiver declaration from a custom application:

```

<receiver android:name="FastBirdApproaching">
    <intent-filter >
        <action android:name="com.acme.coyotebirdmonitor.FAST_BIRD"/>
    </intent-filter>
</receiver>

```

And here's the corresponding code:

```

public class FastBirdApproaching extends BroadcastReceiver {
    private static final String TAG = "FastBirdApproaching";

    @Override
    public void onReceive(Context context, Intent intent) {
        // TODO Auto-generated method stub
        Log.i(TAG, "*****");
        Log.i(TAG, "Meep Meep!");
        Log.i(TAG, "*****");
    }
}

```

Here's how you can use *am* to trigger this broadcast receiver and the resulting output in the logs:

```

# am broadcast -a com.acme.coyotebirdmonitor.FAST_BIRD
Broadcasting: Intent { act=com.acme.coyotebirdmonitor.FAST_BIRD }
Broadcast completed: result=0

# logcat
...
I/ActivityManager( 62): Start proc com.acme.coyotebirdmonitor for broadcast co
m.acme.coyotebirdmonitor/.FastBirdApproaching: pid=466 uid=10029 gids={}
I/FastBirdApproaching( 466): *****
I/FastBirdApproaching( 466): Meep Meep!
I/FastBirdApproaching( 466): *****
...

```

As you can see from *am*'s online help, you can specify a lot of details regarding the intent to be sent. Whereas the previous two examples used implicit intents, you can also send explicit intents to activate designated components:

```

# am start -n com.android.settings/.Settings

```

In this case, this will start the *Settings* activity of the settings app in the system. Interestingly, *am* can start components in ways you can't replicate using the officially

published app development API. That's because it's built as part of the AOSP and has therefore access to hidden calls available only to code building within the AOSP.

am is in fact a shell script, as you can see in *frameworks/base/cmds/am/am/*:

```
# Script to start "am" on the device, which has a very rudimentary
# shell.
#
base=/system
export CLASSPATH=$base/framework/am.jar
exec app_process $base/bin com.android.commands.am.Am "$@"
```

The script uses *app_process* to start Java code that implements *am*'s functionality. All parameters passed on the command line are actually passed on to the Java code as is.

You can also use *am* for instrumentation, profiling, and monitoring. Have a look at the [Testing Fundamentals](#) and [Testing from Other IDEs](#) sections of the Android developer manual for more information on Android testing and the use of the *am instrument* command.

The *am profile* commands allow us to generate data that can then be visualized on the host using the *traceview* command. You can find more information about *traceview* in the relevant section of the [Android developer manual](#). Note that the documentation says there are two ways to create trace files, and the use of the *am* command on the command line isn't listed as one of them.

Finally, the *am monitor* command allows us to monitor apps run by the Activity Manager. Here's a session where I start the command and then start several apps:

```
# am monitor
Monitoring activity manager... available commands:
(q)uit: finish monitoring
** Activity starting: com.android.browser
** Activity resuming: com.android.launcher
** Activity starting: com.android.settings
** Activity resuming: com.android.launcher
** Activity starting: com.android.browser
** Activity starting: com.android.launcher
...
```

Note that when you start an app and click Back, the command reports that the Launcher is resuming, whereas if you click the Home button, the Launcher is reported as starting. This monitoring capability will also allow you to catch ANRs (Application Not Responding) and enable you to attach *gdb* to a crashing process.



Don't let this brief coverage of *am* mislead you: This is an extremely powerful and useful command that you should keep well in mind. If you ever need to script the starting of apps from the command line, you will find it to be very useful.

pm



Another very important system service is the Package Manager and, much like the Activity Manager, it's got its own command-line tool. Here's its online help from 2.3/Gingerbread:

```
# pm
usage: pm [list|path|install|uninstall]
       pm list packages [-f] [-d] [-e] [-u] [FILTER]
       pm list permission-groups
       pm list permissions [-g] [-f] [-d] [-u] [GROUP]
       pm list instrumentation [-f] [TARGET-PACKAGE]
       pm list features
       pm list libraries
       pm path PACKAGE
       pm install [-l] [-r] [-t] [-i INSTALLER_PACKAGE_NAME] [-s] [-f] PATH
       pm uninstall [-k] PACKAGE
       pm clear PACKAGE
       pm enable PACKAGE_OR_COMPONENT
       pm disable PACKAGE_OR_COMPONENT
       pm setInstallLocation [0/auto] [1/internal] [2/external]
```

The list packages command prints all packages, optionally only those whose package name contains the text in FILTER. Options:

- f: see their associated file.
- d: filter to include disabled packages.
- e: filter to include enabled packages.
- u: also include uninstalled packages.

The list permission-groups command prints all known permission groups.

The list permissions command prints all known permissions, optionally only those in GROUP. Options:

- g: organize by group.
- f: print all information.
- s: short summary.
- d: only list dangerous permissions.
- u: list only the permissions users will see.

The list instrumentation command prints all instrumentations, or only those that target a specified package. Options:

- f: see their associated file.

The list features command prints all features of the system.

The path command prints the path to the .apk of a package.

The install command installs a package to the system. Options:

- l: install the package with FORWARD_LOCK.
- r: reinstall an existing app, keeping its data.
- t: allow test .apks to be installed.

- i: specify the installer package name.
- s: install package on sdcard.
- f: install package on internal flash.

The `uninstall` command removes a package from the system. Options:

- k: keep the data and cache directories around.
after the package removal.

The `clear` command deletes all data associated with a package.

The `enable` and `disable` commands change the enabled state of a given package or component (written as "package/class").

The `getInstallLocation` command gets the current install location

- 0 [auto]: Let system decide the best location
- 1 [internal]: Install on internal device storage
- 2 [external]: Install on external media

The `setInstallLocation` command changes the default install location

- 0 [auto]: Let system decide the best location
- 1 [internal]: Install on internal device storage
- 2 [external]: Install on external media



Much like *am*, *pm*'s capabilities have grown through the versions, and the online help in 4.2/Jelly Bean for this tool is now much larger than can reasonably fit in this book. I still encourage you to take a look at it.

Fortunately, this command is actually pretty well documented, as you can see from the output above. Listing the installed packages, for example, is as simple as:

```
# pm list packages
package:android
package:android.tts
package:com.android.bluetooth
package:com.android.browser
package:com.android.calculator2
package:com.android.calendar
package:com.android.camera
package:com.android.certinstaller
package:com.android.contacts
package:com.android.defcontainer
...
```

Installing an app (the command used by the `adb install` command covered in the last chapter):

```
# pm install FastBirds.apk
pkg: FastBirds.apk
Success
```

Note that removing the app requires knowing its package name, not the original *.apk*'s name:

```
# pm uninstall com.acme.fastbirds
Success
```

pm is also a shell script that starts Java code:

```
# Script to start "pm" on the device, which has a very rudimentary
# shell.
#
base=/system
export CLASSPATH=$base/framework/pm.jar
exec app_process $base/bin com.android.commands.pm.Pm "$@"
```



As with *am*, there's much more to *pm* than I can cover in this book. I encourage you to explore its many uses, as it can be very helpful for scripts, either during development and/or in production.

SVC

Unlike the two previous commands, *svc* is something of a Swiss Army knife in attempting to provide you with the ability to control several system services. Here's the online help for 2.3/Gingerbread:

```
# svc
Available commands:
  help    Show information about the subcommands
  power   Control the power manager
  data    Control mobile data connectivity
  wifi    Control the Wi-Fi manager
```

The online help for 4.2/Jelly Bean shows that it can now also deal with USB:

```
root@android:/ # svc
Available commands:
  help    Show information about the subcommands
  power   Control the power manager
  data    Control mobile data connectivity
  wifi    Control the Wi-Fi manager
  usb     Control Usb state
```

Note how *svc*'s capabilities are limited to enabling and disabling the behavior of the designated system services:

```
# svc help power
Control the power manager

usage: svc power stayon [true|false|usb|ac]
       Set the 'keep awake while plugged in' setting.
```



```
# svc help data
Control mobile data connectivity

usage: svc data [enable|disable]
    Turn mobile data on or off.

    svc data prefer
    Set mobile as the preferred data network

# svc help wifi
Control the Wi-Fi manager

usage: svc wifi [enable|disable]
    Turn Wi-Fi on or off.

    svc wifi prefer
    Set Wi-Fi as the preferred data network
```

Overall, you should be aware of *svc*, but it's unlikely that you'll make regular use of it. Like *am* and *pm*, *svc* is also a script that uses *app_process* to start Java code.

ime

The *ime* command lets you communicate with the Input Method system service to control the system's use of available input methods, and it's the same in 2.3/Gingerbread and 4.2/Jelly Bean:

```
# ime
usage: ime list [-a] [-s]
    ime enable ID
    ime disable ID
    ime set ID
```

The *list* command prints all enabled input methods. Use the *-a* option to see all input methods. Use the *-s* option to see only a single summary line of each.

The *enable* command allows the given input method ID to be used.

The *disable* command disallows the given input method ID from use.

The *set* command switches to the given input method ID.

Here's the list of input methods available on the 2.3/Gingerbread emulator, for example:

```
# ime list
com.android.inputmethod.latin/.LatinIME:
  mId=com.android.inputmethod.latin/.LatinIME mSettingsActivityName=com.android.
inputmethod.latin.LatinIMESettings
  mIsDefaultResId=0x7f080001
  Service:
    priority=0 preferredOrder=0 match=0x108000 specificIndex=-1 isDefault=false
  ServiceInfo:
```



```

name=com.android.inputmethod.latin.LatinIME
packageName=com.android.inputmethod.latin
labelRes=0x7f0c001f nonLocalizedLabel=null icon=0x0
enabled=true exported=true processName=com.android.inputmethod.latin
permission=android.permission.BIND_INPUT_METHOD

```

Again, *ime* uses *app_process* from within a script to start Java code. Like *svc*, *ime* is a command worth keeping in mind, but you're unlikely to use it very often.

input

input connects to the Window Manager system service and injects text or key events into the system. Here's how it operates on 2.3/Gingerbread:

```

# input
usage: input [text|keyevent]
       input text <string>
       input keyevent <event_code>

```

Here's how it works on 4.2/Jelly Bean:

```

root@android:/ # input
usage: input ...
       input text <string>
       input keyevent <key code number or name>
       input [touchscreen|touchpad] tap <x> <y>
       input [touchscreen|touchpad] swipe <x1> <y1> <x2> <y2>
       input trackball press
       input trackball roll <dx> <dy>

```

input's functionality is very simple, however. It doesn't, for instance, know anything about what's receiving the events, just that the events are sent to whatever presently has focus. It's therefore up to you to make sure that whatever needs to receive your input actually has focus. Evidently this is difficult when you're not in front of the screen and are, instead, trying to script such behavior. Still, *input* gives you a tool to provide raw input from the command line. And, in some cases, the meaning of the input you send doesn't require focus. Here's how to click the Home button from the command line, for example:

```
# input keyevent 3
```

You're probably wondering how I know that 3 is the Home key. Have a look at *frameworks/base/core/java/android/view/KeyEvent.java* and *frameworks/base/native/include/android/keycodes.h* in 2.3/Gingerbread or *frameworks/native/include/android/keycodes.h* in 4.2/Jelly Bean for the full list of key codes recognized by Android. The former, for example, contains code such as this:

```

...
public static final int KEYCODE_HOME           = 3;
/** Key code constant: Back key. */
public static final int KEYCODE_BACK           = 4;

```

```

    /** Key code constant: Call key. */
    public static final int KEYCODE_CALL          = 5;
    /** Key code constant: End Call key. */
    public static final int KEYCODE_ENDCALL        = 6;
    /** Key code constant: '0' key. */
    public static final int KEYCODE_0             = 7;
    ...

```

Like all other commands, *input* is a script that relies on *app_process*.

monkey

There's another tool that allows you to provide input to Android. It's called *monkey*, and there's an entire section about it in the app developer documentation entitled **UI/Application Exerciser Monkey**. As the documentation says, *monkey* can be used to provide random yet repeatable input to your application. This command, for instance, will send 50 pseudo-random inputs to the browser app:

```
# monkey -p com.android.browser -v 50
```

monkey can, however, do much more, as you can see from this output on 2.3/Gingerbread (4.2/Jelly Bean's is fairly similar):

```

# monkey
usage: monkey [-p ALLOWED_PACKAGE [-p ALLOWED_PACKAGE] ...]
              [-c MAIN_CATEGORY [-c MAIN_CATEGORY] ...]
              [--ignore-crashes] [--ignore-timeouts]
              [--ignore-security-exceptions]
              [--monitor-native-crashes] [--ignore-native-crashes]
              [--kill-process-after-error] [--hprof]
              [--pct-touch PERCENT] [--pct-motion PERCENT]
              [--pct-trackball PERCENT] [--pct-syskeys PERCENT]
              [--pct-nav PERCENT] [--pct-majornav PERCENT]
              [--pct-appswitch PERCENT] [--pct-flip PERCENT]
              [--pct-anyevent PERCENT]
              [--pkg-blacklist-file PACKAGE_BLACKLIST_FILE]
              [--pkg-whitelist-file PACKAGE_WHITELIST_FILE]
              [--wait-dbg] [--dbg-no-events]
              [--setup scriptfile] [-f scriptfile] ...]
              [--port port]
              [-s SEED] [-v [-v] ...]
              [--throttle MILLISEC] [--randomize-throttle]
              [--profile-wait MILLISEC]
              [--device-sleep-time MILLISEC]
              [--randomize-script]
              [--script-log]
              [--bugreport]
              COUNT

```

Most interestingly, you can provide a script to *monkey* for running a predefined set of input instead of providing random input. This is a very useful feature for development, testing, and in-the-field diagnostics. Unfortunately, there's virtually no documentation



whatsoever on this very powerful feature of *monkey*. So, for reference, here's a sample script file:

```
# This is a sample test script
# Lines starting with '#' are comments

# This part is the "header"
# monkey doesn't actually look for 'type', but does require 'count', 'speed' and
# 'start data >>'
type= custom
count= 100
speed= 1.0
start data >>

# These are the actual instructions to carry out
LaunchActivity(com.android.contacts,com.android.contacts.TwelveKeyDialer)
# Use this instead in 4.2./Jelly Bean (line-wrap is for book, remove to run)
# LaunchActivity(com.android.contacts,com.android.contacts.activities.Dialtact
# sActivity)
UserWait(2500)
DispatchPress(KEYCODE_1)
UserWait(200)
DispatchPress(KEYCODE_8)
UserWait(200)
DispatchPress(KEYCODE_0)
UserWait(200)
DispatchPress(KEYCODE_0)
UserWait(200)
DispatchPress(KEYCODE_8)
UserWait(200)
DispatchPress(KEYCODE_8)
UserWait(200)
DispatchPress(KEYCODE_9)
UserWait(200)
DispatchPress(KEYCODE_8)
UserWait(200)
DispatchPress(KEYCODE_9)
UserWait(200)
DispatchPress(KEYCODE_6)
UserWait(200)
DispatchPress(KEYCODE_9)
UserWait(200)
DispatchPress(KEYCODE_ENTER)
UserWait(10000)
DispatchPress(KEYCODE_ENDCALL)
UserWait(200)
RunCmd(input keyevent 3)
UserWait(1000)
RunCmd(service call statusbar 1)
UserWait(2000)
RunCmd(service call statusbar 2)
```

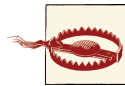
To run this script, use this command line:

```
# monkey -f myscript 1
```

This script will essentially start the standard dialer, dial 1-800-889-8969,³ wait 10 seconds, hang up, return to the home screen, and then expand and collapse the status bar. Notice that the last part uses the `RunCmd` instruction to make the script run commands straight from the command line; incidentally these are commands we saw earlier. Of course this script is rather short and simple. You can create much longer scripts; you can possibly even integrate the invocation of such scripts into much more complicated shell scripts.

For a detailed understanding of the scripting language understood by *monkey*, along with the parameters each command can take, I invite you to take a look at *monkey*'s script interpreting code in `development/cmds/monkey/src/com/android/commands/monkey/MonkeySourceScript.java` and look for `EVENT_KEYWORD_`. You should then find event keywords such as `DispatchPress`, `UserWait`, and many others.

To do its magic, *monkey* communicates with the Activity Manager, the Window Manager, and the Package Manager. It too is a shell script that relies on *app_process* to start the Java code that implements the utility.



If you look into the tool's sources in `development/cmds/monkey/`, you will find a file called `example_script.txt` that appears to contain some scripted instructions. It's unclear why this file is in the sources, as the semantics in that file do not correspond to the actual semantics expected by the *monkey* utility.

bmgr

Since 2.2/Froyo, Android has included a backup capability, allowing users to have their data backed up into the cloud so it can be restored later should they lose or change their device. Google itself provides some of this capability by acting as one of the possible *transports*,⁴ but others could provide alternative transports. The API provided within Android and to app developers is transport-independent. This remains, however, a functionality that is very specific to the use of Android for phones and tablets and may not be required in an embedded environment. There's a tool that allows you to control the behavior of the Backup Manager system service from the command line:⁵

```
# bmgr
usage: bmgr [backup|restore|list|transport|run]
```

3. The publisher's phone number, if you're wondering.
4. A "transport" in the context of *bmgr* is the required engine to interface with a given cloud service.
5. This is the output on 2.3/Gingerbread. 4.2/Jelly Bean's is fairly similar.

```
bmgr backup PACKAGE
bmgr enable BOOL
bmgr enabled
bmgr list transports
bmgr list sets
bmgr transport WHICH
bmgr restore TOKEN
bmgr restore PACKAGE
bmgr run
bmgr wipe PACKAGE
```

The 'backup' command schedules a backup pass for the named package. Note that the backup pass will effectively be a no-op if the package does not actually have changed data to store.

The 'enable' command enables or disables the entire backup mechanism. If the argument is 'true' it will be enabled, otherwise it will be disabled. When disabled, neither backup or restore operations will be performed.

The 'enabled' command reports the current enabled/disabled state of the backup mechanism.

The 'list transports' command reports the names of the backup transports currently available on the device. These names can be passed as arguments to the 'transport' command. The currently selected transport is indicated with a '*' character.

The 'list sets' command reports the token and name of each restore set available to the device via the current transport.

The 'transport' command designates the named transport as the currently active one. This setting is persistent across reboots.

The 'restore' command when given a restore token initiates a full-system restore operation from the currently active transport. It will deliver the restore set designated by the TOKEN argument to each application that had contributed data to that restore set.

The 'restore' command when given a package name initiates a restore of just that one package according to the restore set selection algorithm used by the `RestoreSession.restorePackage()` method.

The 'run' command causes any scheduled backup operation to be initiated immediately, without the usual waiting period for batching together data changes.

The 'wipe' command causes all backed-up data for the given package to be erased from the current transport's storage. The next backup operation that the given application performs will rewrite its entire data set.

If this is relevant to your use of Android, have a look at the [Data Backup](#) section of the app developer manual, along with the information [provided by Google](#) regarding its own backup transport. Much like many of the other commands we saw, *app_process* is used to start the actual Java code that interfaces with the Backup Manager service.

stagefright

One of Android's key features is its rich media layer, and the AOSP includes tools that enable you to interact with it. More specifically, the *stagefright* command interacts with the Media Player service to allow you to do media playback. Here's its online help in 2.3/Gingerbread (4.2/Jelly Bean's is slightly expanded):

```
# stagefright -h
usage: stagefright
      -h(elp)
      -a(udio)
      -n repetitions
      -l(ist) components
      -m max-number-of-frames-to-decode in each pass
      -b bug to reproduce
      -p(rofiles) dump decoder profiles supported
      -t(humbnail) extract video thumbnail or album art
      -s(oftware) prefer software codec
      -o playback audio
      -w(rite) filename (write to .mp4 file)
      -k seek test
```

Here's how you can play an .mp3 file, for example:

```
# stagefright -a -o /sdcard/trainwhistle.mp3
```

You might also want to investigate the *record* and *audioloop* utilities found alongside *stagefright*'s sources in *frameworks/base/cmds/stagefright/* in 2.3/Gingerbread and *frameworks/av/cmds/stagefright/* in 4.2/Jelly Bean. Their documentation is severely lacking, though, and few examples of their uses can be found online or elsewhere. Interestingly, though, all three utilities are coded in C, unlike the majority of the system service-specific utilities we've seen thus far, which were mostly written in Java and activated through a script using *app_process*. Also, while *stagefright* directly communicates with the Media Player service, the *record* and *audioloop* commands use an *OMXClient*, which conveniently wraps the communication to the same service.

Dalvik Utilities

We've already seen how we can send intents with the *am* command and therefore trigger the starting of new apps, each of which comes with its own Zygote-forked Dalvik instances. We've also seen how the *app_process* command can be used to start Java-coded command-line tools using the Android Runtime. There are some cases, however, where

you may want to forgo all the Android-specific layers and dabble directly with Dalvik. Here are the commands that allow you to do just that.

dalvikvm

If you haven't yet already asked yourself if there's a way to actually start just a Dalvik VM without any Android-specific functionality, here's the command you've been looking for:⁶

dalvikvm -help

```
dalvikvm: [options] class [argument ...]
dalvikvm: [options] -jar file.jar [argument ...]
```

The following standard options are recognized:

- classpath classpath
- Dproperty=value
- verbose:tag ('gc', 'jni', or 'class')
- ea[:<package name>... |:<class name>]
- da[:<package name>... |:<class name>]
- (-enableassertions, -disableassertions)
- esa
- dsa
- (-enablesystemassertions, -disablesystemassertions)
- showversion
- help

The following extended options are recognized:

- Xrunjwp:<options>
- Xbootclasspath:bootclasspath
- Xcheck:tag (e.g. 'jni')
- XmsN (min heap, must be multiple of 1K, >= 1MB)
- XmxN (max heap, must be multiple of 1K, >= 2MB)
- XssN (stack size, >= 1KB, <= 256KB)
- Xverify:{none,remote,all}
- Xrs
- Xint (extended to accept ':portable', ':fast' and ':jit')

These are unique to Dalvik:

- Xzygote
- Xdexopt:{none,verified,all}
- Xnoquithandler
- Xjnimreflimit:N (must be multiple of 100, >= 200)
- Xjnipts:{warnonly,forcecopy}
- Xjnitrac:substring (eg NativeClass or nativeMethod)
- Xdeadlockpredict:{off,warn,err,abort}
- Xstacktracefile:<filename>
- Xgc:[no]precise

6. This is the output from 2.3/Gingerbread. 4.2/Jelly Bean's output is fairly similar.

```

-Xgc:[no]preverify
-Xgc:[no]postverify
-Xgc:[no]concurrent
-Xgc:[no]verifycardtable
-Xgenregmap
-Xcheckdexsum
-Xincludeselectdop
-Xjitop:hexopvalue[-endvalue][,hexopvalue[-endvalue]]*
-Xincludeselectmethod
-Xjitthreshold:decimalvalue
-Xjitblocking
-Xjitmethod:signature[,signature]* (eg Ljava/lang/String\;replace)
-Xjitcheckcg
-Xjitverbose
-Xjitprofile
-Xjitdisableopt

```

Configured with: debugger profiler hprof jit(armv5te) show_exception=1

Dalvik VM init failed (check log file)

dalvikvm is actually a raw Dalvik VM without any connection to “Android” whatsoever. It doesn’t rely on the Zygote, nor does it include the Android Runtime. It simply starts a VM to run whatever class or JAR file you provide it. It’s actually not used very often in the AOSP itself, probably because there isn’t much in the AOSP that doesn’t run in the context of “Android.” The “preload” Java library in 2.3/Gingerbread, for example, uses it in *frameworks/base/tools/preload/MemoryUsage.java* in conjunction with *adb* to check the amount of memory used by a class on the target.

dvz

Yet another way to start a Dalvik VM is the *dvz* command:

```

# dvz --help
Usage: dvz [--help] [-classpath <classpath>]
[additional zygote args] fully.qualified.java.ClassName [args]

```

Requests a new Dalvik VM instance to be spawned from the zygote process. stdin, stdout, and stderr are hooked up. This process remains while the spawned VM instance is alive and forwards some signals. The exit code of the spawned VM instance is dropped.

As the description implies, *dvz* actually acts in a similar fashion to the Activity Manager by requesting the Zygote to fork and start a new process. The only difference here is that the resulting process isn’t managed by the Activity Manager. Instead, it’s very much standalone.

It’s unclear whether this utility is meant to be heavily used, as the only instances of its use within 2.3/Gingerbread are in test code, specifically in *dalvik/tests/etc/push-and-*

run-test-jar, and it's not even included in the default builds in 4.2/Jelly Bean. Nevertheless, there might be instances where having this in your arsenal could be useful.

The Many Ways to Start Dalvik

Up to now, we've seen four different ways to start a Dalvik VM. It's worth taking a moment to put them all in perspective. **Table 7-1** describes each way to get a working Dalvik VM, along with what's included in the VM and how it's started.

Table 7-1. Ways to start Dalvik

Command	Dalvik VM	Android Runtime	Zygote	Activity Manager	Mechanism
<i>dalvikvm</i>	X				Uses <i>libdvm.so</i>
<i>app_process</i>	X	X			Uses <i>libandroid_runtime.so</i>
<i>dvz</i>	X	X	X		Uses <i>libcutils</i> ^a
<i>am</i>	X	X	X	X	Talks to Activity Manager service

^aSee *system/core/libcutils/zygote.c*, which contains a *zygote_run_wait()* and a *zygote_run_one_shot()*.

am is the only command that provides us with a Dalvik VM instance that's actually controlled by the Activity Manager. In all other cases, the VM is independent and does not have its lifecycle managed. *am* is also the only command that allows us to automatically trigger the execution of code contained in an *.apk*. All other commands require us to provide a specific class or JAR file.

dexdump

If you'd like to reverse-engineer Android apps or JAR files, you can do so with *dexdump*:

```
# dexdump
dexdump: no file specified
Copyright (C) 2007 The Android Open Source Project

dexdump: [-c] [-d] [-f] [-h] [-i] [-l layout] [-m] [-t tempfile] dexfile...

-c : verify checksum and exit
-d : disassemble code sections
-f : display summary information from file header
-h : display file header details
-i : ignore checksum failures
-l : output layout, either 'plain' or 'xml'
-m : dump register maps (and nothing else)
-t : temp file name (defaults to /sdcard/dex-temp-*)
```

Here's how it can be used on a JAR file:

```
# dexdump /system/framework/services.jar
Processing '/system/framework/services.jar'...
Opened '/system/framework/services.jar', DEX version '035'
Class #0
  Class descriptor : 'Lcom/android/server/AccessibilityManagerService$1;'
  Access flags     : 0x0000 ()
  Superclass      : 'Landroid/os/Handler;'
  Interfaces      : -
  Static fields   : -
  Instance fields : -
    #0            : (in Lcom/android/server/AccessibilityManagerService$1;)
      name        : 'this$0'
      type        : 'Lcom/android/server/AccessibilityManagerService;'
      access      : 0x1010 (FINAL SYNTHETIC)
  Direct methods  : -
    #0            : (in Lcom/android/server/AccessibilityManagerService$1;)
      name        : '<init>'
      type        : '(Lcom/android/server/AccessibilityManagerService;)V'
      access      : 0x10000 (CONSTRUCTOR)
      code        : -
      registers   : 2
      ins         : 2
      outs        : 1
      insns size   : 6 16-bit code units
      catches      : (none)
      positions    :
        0x0000 line=113
      locals      :
        0x0000 - 0x0006 reg=0 this Lcom/android/server/AccessibilityManagerServi
ce$1;
  Virtual methods : -
    #0            : (in Lcom/android/server/AccessibilityManagerService$1;)
      name        : 'handleMessage'
  ...
```

You can also ask it to disassemble code:

```
# dexdump -d /system/app/Launcher2.apk
...
00ea5c:                                [[00ea5c] com.android.common.Array
yListCursor.<init>:(Ljava/lang/String;Ljava/util/ArrayList;)V
00ea6c: 1206                                |0000: const/4 v6, #int 0 // #0
00ea6e: 1a07 e804                          |0001: const-string v7, "_id" //
string@04e8
00ea72: 7010 b400 0800                    |0003: invoke-direct {v8}, Landro
id/database/AbstractCursor.<init>:()V // method@00b4
00ea78: 2190                                |0006: array-length v0, v9
00ea7a: 1201                                |0007: const/4 v1, #int 0 // #0
00ea7c: 1202                                |0008: const/4 v2, #int 0 // #0
00ea7e: 3502 0f00                          |0009: if-ge v2, v0, 0018 // +000
f
```



```

00ea82: 4604 0902                |000b: aget-object v4, v9, v2
00ea86: 1a05 e804                |000d: const-string v5, "_id" //
string@04e8
00ea8a: 6e20 dd07 7400           |000f: invoke-virtual {v4, v7}, L
java/lang/String;.compareToIgnoreCase:(Ljava/lang/String;)I // method@07dd
00ea90: 0a04                    |0012: move-result v4
00ea92: 3904 3e00               |0013: if-nez v4, 0051 // +003e
00ea96: 5b89 3600               |0015: iput-object v9, v8, Lcom/a
ndroid/common/ArrayListCursor;.mColumnNames:[Ljava/lang/String; // field@0036
00ea9a: 1211                    |0017: const/4 v1, #int 1 // #1
00ea9c: 3901 1400               |0018: if-nez v1, 002c // +0014
00eaa0: d804 0001               |001a: add-int/lit8 v4, v0, #int
1 // #01
00eaa4: 2344 d901               |001c: new-array v4, v4, [Ljava/l
ang/String; // class@01d9
00eaa8: 5b84 3600               |001e: iput-object v4, v8, Lcom/a
ndroid/common/ArrayListCursor;.mColumnNames:[Ljava/lang/String; // field@0036
00eaac: 5484 3600               |0020: iget-object v4, v8, Lcom/a
ndroid/common/ArrayListCursor;.mColumnNames:[Ljava/lang/String; // field@0036
...

```

Obviously the topic of reverse-engineering Android goes way beyond the scope of this book, but if this topic is of general interest, I recommend taking a look at your favorite online bookstore for books that specialize in Android security and forensics.

Support Daemons

While the bulk of Android's intelligence is implemented in system services, there are a number of cases where a system service acts partly as intermediary to a native daemon that actually does the key operations required. There are likely two main reasons why this approach has been favored instead of conducting the actual operations directly as part of a system server: security and reliability.

As I explained in [Chapter 1](#), Android's permission model requires app developers who need to call on privileged operations to request specific permissions at build time. Typically, these permissions will resemble something like this in an app's manifest file:

```

...
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
...

```

In this case, these permissions ask for the ability to open sockets and grab wakelocks. There are obviously a whole lot more permissions than this. Have a look at the app developer documentation on the full list of [permissions available](#). Without these permissions, an app can't conduct some of the most critical Android operations. And the main reason is that apps run as unprivileged users that can't, for instance, invoke any system call that requires root privileges or access most of the key devices in */dev*. Instead,



apps must ask system services to act on their behalf and, in turn, system services check apps' permissions before following through with any requests they get.

System services don't, however, themselves run as root. Instead, the *system_server* process runs as *system*; the *mediaserver* process runs as *media*; and the Phone app runs as *radio*. And if you check in */dev*, you'll see that some entries belong exclusively to some of these users. You'll also see quite a few entries that belong to the root user. Hence, much like apps, system services can't typically use system calls that require root privileges nor access key devices in */dev*.

Instead, many key operations require system services to communicate through Unix domain sockets in */dev/socket/* with native daemons running as either root or as a specific user to conduct privileged operations. Many of those daemons are Android-specific, though some, such as *bluetoothd* prior to 4.2/Jelly Bean, we've already covered in [Chapter 6](#) as being legacy Linux daemons.

In some specific cases, such as *rild*, for example, which takes care of the communication with the Baseband Processor, it seems that the choice to run as a separate process might likely have more to do with reliability. Indeed, the phone functionality of a smartphone is so critical that it's worth ensuring that its operation is independent of any potential issues that could affect the system services housed in the *system_server* process.

Let's take a look at the main support daemons used by system services, their configuration, and related command-line tools. Note that we won't cover the daemons we covered earlier, such as the Zygote; or those that aren't tied to system services, such as *ueventd* and *dumppsys*; or those, such as *bluetoothd* or *wpa_supplicant*, that are not Android specific.

installd

While the Package Manager service's job is to deal with the management of *.apk* files, it doesn't have the proper privileges to carry out many of the manipulations and/or operations required to set up an app to run. Instead, it relies on *installd*, which runs as root in 2.3/Gingerbread and as **the install user in 4.2/Jelly Bean, for key filesystem operations and commands**. Running *dexopt* on an *.apk* to generate JIT-optimized *.dex* files for Dalvik, for instance, is done by *installd* on the Package Manager's behalf at install time.

installd is started by this section of *init.rc* in 2.3/Gingerbread (4.2/Jelly Bean does something fairly similar):

```
service installd /system/bin/installd
    socket installd stream 600 system system
```

It then opens */dev/socket/installd* and listens for a connection, and thereafter listens for commands from the Package Manager. It doesn't have a configuration file, nor does it



take any command-line parameters. Neither is there any command-line tool to communicate with it independently of the Package Manager. Hence, the only way to activate *installd* from the command line is to use the *pm* command, which will communicate with the Package Manager, which will, in turn, communicate with *installd* if required.

installd's sources are in *frameworks/base/cmds/installd/*, and you may want to take a look at *install.c* and *commands.c*. The former contains the list of commands recognized by *installd*, and the latter contains the actual implementation of those commands. For reference, here's the snippet from 2.3/Gingerbread's *install.c* that lists the commands recognized by *installd* (4.2/Jelly Bean adds a few more commands to that list):

```
struct cmdinfo cmds[] = {
    { "ping",                0, do_ping },
    { "install",             4, do_install },
    { "dexopt",              3, do_dexopt },
    { "movedex",             2, do_move_dex },
    { "rmdex",               1, do_rm_dex },
    { "remove",              2, do_remove },
    { "rename",              3, do_rename },
    { "freecache",           1, do_free_cache },
    { "rmcache",             2, do_rm_cache },
    { "protect",             2, do_protect },
    { "getsize",             4, do_get_size },
    { "rmuserdata",         2, do_rm_user_data },
    { "movefiles",          0, do_movefiles },
    { "linklib",            2, do_linklib },
    { "unlinklib",          1, do_unlinklib },
};
```

Note that, much like many of the other daemons we'll see below, the wire protocol between *installd* and the Package Manager is string based. Hence, the above snippet contains three entries per command: the command's string as sent "on the wire," the number of parameters expected, and the function within *install.c* to call when the command is received.

vold

vold takes care of many of the key operations required by the Mount Service, such as mounting and formatting volumes. Unlike *installd*, *vold* runs as root in both 2.3/Gingerbread and 4.2/Jelly Bean, while the Mount Service is part of the System Server. *vold* is started by this section of 2.3/Gingerbread's *init.rc* (the snippet in 4.2/Jelly Bean is similar):

```
service vold /system/bin/vold
    socket vold stream 0660 root mount
    ioprio be 2
```

Unlike the rest of the support daemons covered here, *vold* actually has a configuration file, */etc/vold.fstab*. Here's a snippet from the default *vold.fstab* found in *system/core/rootdir/etc/* describing the file's semantics:

```
#####
## Regular device mount
##
## Format: dev_mount <label> <mount_point> <part> <sysfs_path1...>
## label      - Label for the volume
## mount_point - Where the volume will be mounted
## part       - Partition # (1 based), or 'auto' for first usable partition.
## <sysfs_path> - List of sysfs paths to source devices
#####
```

Here's the section that relates to the SD card in the emulator, for example:

```
dev_mount sdcard /mnt/sdcard auto /devices/platform/goldfish_mmc.0 /devices/platform/msm_sdcc.2/mmc_host/mmc1
```

When *vold* starts, it parses this file and then opens */dev/socket/vold* to listen for connections and commands. Unlike *installd*, there's a command-line tool to communicate directly with *vold*:

```
Usage: vdc <monitor>|<cmd> [arg1] [arg2...]
```

The actual parameters expected by *vdc* on the command line are the same as those expected by *vold* from the Mount Service when it connects through the designated socket. There is, unfortunately, no document or online help that describes the complete command set. Instead, you must look at the *CommandListener.cpp* file in *system/vold/* to see the implementation of *vold*'s command set.

You can, for instance, dump *vold*'s internal status:

```
# vdc dump
000 Dumping loop status
000 Dumping DM status
000 Dumping mounted filesystems
000 rootfs / rootfs ro 0 0
000 tmpfs /dev tmpfs rw,mode=755 0 0
000 devpts /dev/pts devpts rw,mode=600 0 0
000 proc /proc proc rw 0 0
000 sysfs /sys sysfs rw 0 0
000 none /acct cgroup rw,cpuacct 0 0
000 tmpfs /mnt/asec tmpfs rw,mode=755,gid=1000 0 0
000 tmpfs /mnt/obb tmpfs rw,mode=755,gid=1000 0 0
000 none /dev/cpuctl cgroup rw,cpu 0 0
000 /dev/block/mtdblock0 /system yaffs2 ro 0 0
000 /dev/block/mtdblock1 /data yaffs2 rw,nosuid,nodev 0 0
000 /dev/block/mtdblock2 /cache yaffs2 rw,nosuid,nodev 0 0
200 dump complete
```

In some cases, *vdc* actually offers online help:

```
# vdc volume format
500 Usage: volume format <path>
```

To customize the list of storage devices for your device in 4.2/Jelly Bean, have a look at *frameworks/base/core/res/res/xml/storage_list.xml*. You may want to create an overlay version of that file in your *device/acme/coyotepad/overlay/* to customize it for your device.

netd

The Network Management Service relies on *netd* for critical network configuration operations such as configuring network interfaces, setting up tethering, and running *pppd*. In this case, too, *netd* runs as root, while the Network Management Service is part of the System Server. *netd* is started by the following section of *init.rc* in 2.3/Gingerbread:

```
service netd /system/bin/netd
    socket netd stream 0660 root system
```

In 4.2/Jelly Bean, however, the declaration has changed:

```
service netd /system/bin/netd
    class main
    socket netd stream 0660 root system
    socket dnspoxyd stream 0660 root inet
    socket mdns stream 0660 root system
```

netd opens */dev/socket/netd* and listens for connections and commands. It doesn't take any command-line parameters, nor does it rely on any configuration file. Like *vold*, however, it has a command-line tool to communicate with it. Here's the online help for that command in 2.3/Gingerbread:

```
# ndc
Usage: ndc <monitor>|<cmd> [arg1] [arg2...]
```

Here's the same help on 4.2/Jelly Bean:

```
root@android:/ # ndc
Usage: ndc [sockname] <monitor>|<cmd> [arg1] [arg2...]
```

Like *vdc*, the command-line parameters expected by *ndc* are the same as those expected by *netd* on its socket. And as with *vold*, you need to look at *netd*'s *CommandListener.cpp* in *system/netd/* to understand its command semantics.

As with *vdc*, you can request *netd* status info with *ndc*:

```
# ndc interface list
110 lo
110 eth0
110 tunl0
110 gre0
200 Interface list completed
```



The Command Sets of *vold* and *netd*

Both *vold* and *netd* are constructed using the same C++ mechanism provided by *libsysutils* and rely on a *CommandListener.cpp* to parse and dispatch commands sent to them. To understand the specific commands accepted by each, have a look at the constructors in *CommandListener.cpp*:

```
CommandListener::CommandListener() :  
    FrameworkListener("...") {  
    ...
```

Each will contain calls to `registerCmd()`, which register objects defined farther below in the same file. Here's an excerpt from *vold* for the *dump* command in 2.3/Gingerbread:

```
CommandListener::CommandListener() :  
    FrameworkListener("vold") {  
    registerCmd(new DumpCmd());  
    registerCmd(new VolumeCmd());  
    ...  
CommandListener::DumpCmd::DumpCmd() :  
    VoldCommand("dump") {  
}  
  
int CommandListener::DumpCmd::runCommand(SocketClient *cli,  
                                         int argc, char **argv) {  
    cli->sendMsg(0, "Dumping loop status", false);  
    if (Loop::dumpState(cli)) {  
        cli->sendMsg(ResponseCode::CommandOkay, "Loop dump failed", true);  
    }  
    ...
```

Every command accepted by *vold* or *netd* has a corresponding `runCommand()` that parses the parameters passed to that command. By running *vdump* on the command line as we did earlier, for instance, we're invoking the `runCommand()` in the snippet above. Conversely, typing *vdump volume list* will invoke the following function and pass *list* as one part of the arguments:

```
int CommandListener::VolumeCmd::runCommand(SocketClient *cli,  
                                           int argc, char **argv) {  
    ...
```

rild

The Phone system service, which is hosted in the Phone app, uses *rild* to communicate with the Baseband Processor. *rild* itself uses `dlopen()` to load a baseband-specific *.so* to interface to the actual baseband hardware. As I mentioned before, *rild* likely exists to ensure that the phone side of the system remains active even if a problem occurs with the rest of the stack.



In the case of the emulator, *rild* is started by this portion of the *init.rc* file in 2.3/Gingerbread (4.2/Jelly Bean's version is practically identical):

```
service ril-daemon /system/bin/rild
    socket rild stream 660 root radio
    socket rild-debug stream 660 radio system
    user root
    group radio cache inet misc audio sdcard_rw
```

While it doesn't have a configuration file, *rild* itself can take a few command-line parameters:

```
Usage: rild -l <ril impl library> [-- <args for impl library>]
```

If no RIL implementation library is provided on the command line, *rild* will attempt to locate the library using the `rild.libpath` global property. If that isn't specified either, it'll assume there's no radio on the system loop around calls to `sleep()`. In the case of the emulator, the system relies on `/system/lib/libreference-ril.so`, which, as its name implies, is a reference implementation for manufacturers that need to implement real RIL libraries.

There are two Unix domain sockets used by *rild*: `/dev/socket/rild`, which is used by the Phone system service, and `/dev/socket/rild-debug`, which can be used by the *radiooptions* command to interact. Indeed, the latter is a command-line tool to communicate with *rild*:

```
Usage: radiooptions [option] [extra_socket_args]
    0 - RADIO_RESET,
    1 - RADIO_OFF,
    2 - UNSOL_NETWORK_STATE_CHANGE,
    3 - QXDM_ENABLE,
    4 - QXDM_DISABLE,
    5 - RADIO_ON,
    6 apn- SETUP_PDP apn,
    7 - DEACTIVE_PDP,
    8 number - DIAL_CALL number,
    9 - ANSWER_CALL,
    10 - END_CALL
```

If you'd like to know more about *rild* and *radiooptions*, have a look at their sources in *hardware/ril/rild*. The reference RIL implementation is itself in *hardware/ril/reference-ril*.

keystore

Unlike the rest of the daemons I've presented thus far, *keystore* doesn't actually service any of the system services. Instead, it's used by a variety of different pieces of the system for the storage and retrieval of key-value pairs. The values it maintains are mainly security keys for connecting to networks or network infrastructure such as access points



and VPNs, and the means to secure the values is a user-defined password. Clearly, the goal of having a separate daemon for the storage of this information is to increase the system's overall security.

keystore is started by this portion of the *init.rc* file in 2.3/Gingerbread (4.2/Jelly Bean does substantially the same):

```
service keystore /system/bin/keystore /data/misc/keystore
    user keystore
    group keystore
    socket keystore stream 666
```

keystore doesn't have a configuration file, but it does expect to be provided with a directory to store each key-pair value. Typically, this is */data/misc/keystore*, as you can see before. *keystore* then listens in to */dev/socket/keystore* for connections and commands. Several native daemons connect to *keystore* to retrieve keys, such as *wpa_supplicant*, *mtpd*, and *racoon*. But the Settings app also connects to *keystore* to list and insert new keys.

There's also a command-line utility for communicating with *keystore*:

```
Usage: keystore_cli action [parameter ...]
```

You'll find both the sources of *keystore* and *keystore_cli* in *frameworks/base/cmds/keystore/* in 2.3/Gingerbread and in *system/security/keystore/* in 4.2/Jelly Bean.

Other Support Daemons

There are a few additional daemons that play a more minor role, which we won't cover here, such as *mtpd* and *racoon*. The former is used for VPNs and is found in *external/mtpd/*, and the latter is for IPsec and is found in *external/ipsec-tools/*.

There are possibly, of course, other daemons that may be running on your system for specific purposes, and/or you may want to add your own custom daemons. Have a look back at [Chapter 4](#) for instructions on how to add your own custom binaries to the AOSP's build system. Remember that if you want a daemon to be started at startup by *init*, you need to add a service declaration for it in either the main *init.rc* or in the board-specific *init.<device_name>.rc*.

Hardware Abstraction Layer

As I explained in [Chapter 2](#), Android relies on a Hardware Abstraction Layer (HAL) to interface with hardware. Indeed, system services almost never interact with devices through */dev* entries directly. Instead, they go through HAL modules, typically shared libraries, to talk to hardware, as is detailed in [Table 2-1](#).



Android's HAL implementation is found in *hardware/*. Most importantly, you'll find the definitions of the interfaces between the Framework and the HAL modules in header files in *hardware/libhardware/include/hardware/* and *hardware/libhardware_legacy/include/hardware_legacy/*. The header files therein provide the exact API required for each type of hardware to be supported under Android. You'll also find example implementations of some of those HAL modules in the sources for the lead devices in *device/*.

Ideally, you want to avoid having to implement your own HAL modules for existing system services. Instead, you should query your SoC or board vendor for such modules. HAL module writing requires intricate knowledge of the internals of the system server that the module has to interact with and the specific Linux device driver required to interact with the hardware. Learning how to do this right can be a very time-intensive process, especially since the HAL interface tends to evolve with every new version of Android. I therefore strongly recommend that you use components/boards for which most HAL modules have already been made by the manufacturer or the SoC vendor.

Generally, given Android's market success, component and SoC vendors make a big effort to ensure that Android runs well with their products. This means they either provide you with fully functional AOSPs and Android-ready kernels for eval boards, and/or HAL modules and Linux drivers for their components. So, at the risk of sounding redundant, implement your own HAL modules for hardware types already recognized by Android only as a last resort. Instead, talk to your SoC or component vendor to get your hands on the HAL modules and drivers (or kernel) required to run Android on your hardware.



All major SoC vendors provide—in one way or another—access to ready-to-use AOSPs and kernels for running on the eval boards. Such is the case for TI, Qualcomm, Freescale, Samsung, and many others. If you're building your own custom board based on one of their designs, I recommend that you grab those reference AOSP trees and customize them for your own use. Attempting to start from scratch to port Android to your hardware using the AOSP trees provided directly from Google is not likely to be a good use of your time or fit your time-to-market requirements.

If you absolutely must implement your own HAL modules for existing system services, then refer to the header files I alluded to previously, which define the APIs required by each HAL module type, and take as much inspiration as possible from the reference HAL implementations provided for the lead devices in the *device/* directory. For 2.3/Gingerbread, for example, have a look at the various *lib** directories in *device/samsung/crespo/*. In the case of 4.2/Jelly Bean, have a look at *device/asus/grouper/* and *device/samsung/tuna/*.

