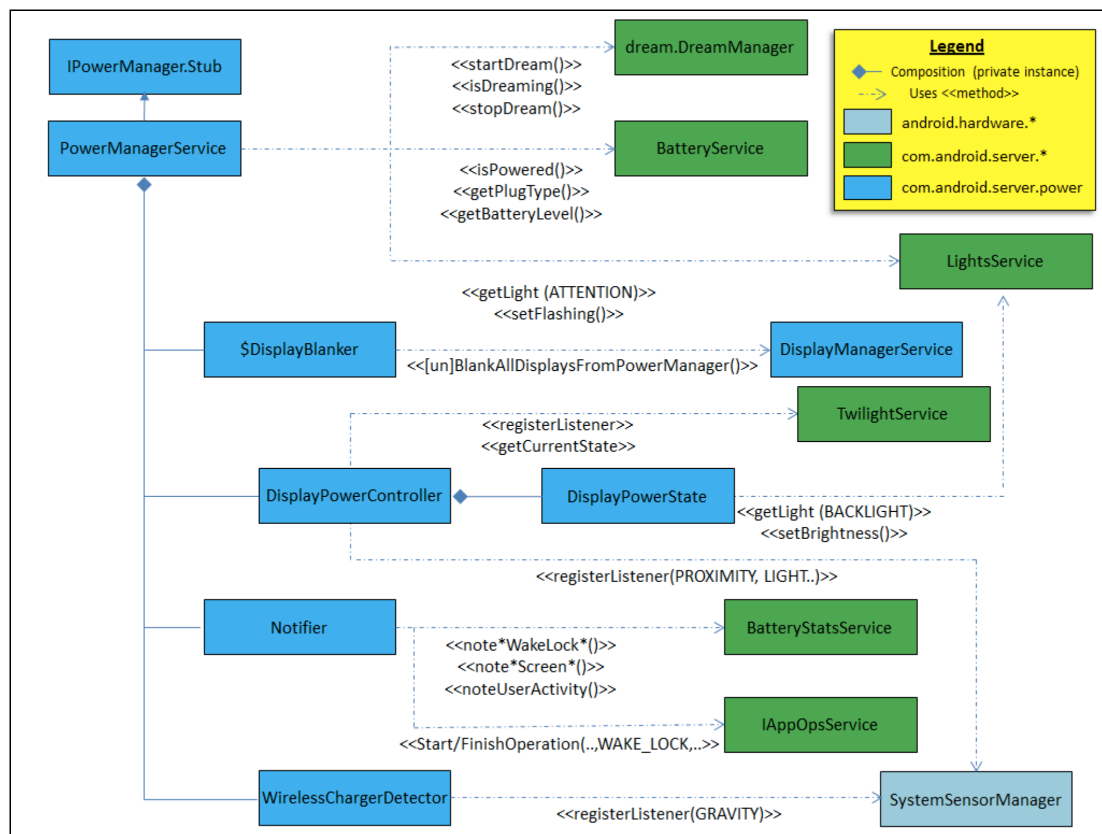
The `PowerManagerService` and Friends

Android provides an aggressive power management mechanism, which is meant to address one of the chief weaknesses of mobile devices - their very finite (and often insufficient) battery life. The usual approach of throwing ever increasing-in-size batteries does extend battery life, but is offset by large displays and more sensors which guzzle battery. Additionally, errant (or intentionally misbehaving) processes can crunch CPU and quickly drain the battery. And with no battery, the device - be it a cheap clone or high end phone - is about as useful as a paperweight.

It should come as no surprise, then, that Power Management is one of the areas which marks notable improvements in Android L. Google announced "Project Volta", which is meant to further conserve battery power by scheduling CPU (and therefore power) hungry jobs when the device is plugged in (among other conditions, as discussed earlier in this chapter). Android L also offers extensive battery statistics down to per-app usage, and a Battery Historian tool to visualize consumption (as we discuss later).

The `PowerManagerService` is entrusted with maintaining the system power state, but does not do so on its own - It requires quite a few services to work together, in order to constantly monitor the battery status, respond to power events, and save on power consumption whenever possible. A high level diagram of the architecture is presented in figure s2-pma:

Figure s2-pma: The (slightly simplified) Android Power Management architecture



We next turn to explore the key components of the architecture, beginning with the `PowerManagerService` itself.

PowerManagerService

PowerManagerService	
Interface:	android.os.IPowerManager
Dependencies:	Lights Content Providers Battery
Permissions:	`WAKE_LOCK`, `DEVICE_POWER`
Thread:	`PowerManagerService`
JNI:	Yes

The `PowerManagerService` offers its clients control over the device sleep policy and some control over power consumption. Applications and services can instantiate the `PowerManager` object as a proxy, and may then use it to acquire a **wakelock**, which can prevent the device from going to sleep or turning off the framebuffer (for example, when watching a movie). Clients may also instruct the device to nap, sleep, wake up or even reboot (or crash). The `PowerManager` object provides a rich set of methods, only some of which are detailed in the documentation. Table s2-pm shows these methods, including those which are are hidden, but nonetheless public:

Table s2-pm: The `PowerManager` object methods

Method
void userActivity(long when, boolean noChangeLights)
boolean isInteractive()
void goToSleep(long time)
void wakeUp(long time)
void reboot(String reason)
WakeLock newWakeLock(int levelAndFlags, String tag)
int getMinimumScreenBrightnessSetting() int getMaximumScreenBrightnessSetting() int getMaximumScreenBrightnessDefault()
bool useScreenAutoBrightnessAdjustmentFeature()
bool useTwilightAdjustmentFeature()
validateWakeLockParameters (int *levelAndFlags*, String *tag*)
void nap()
setBacklightBrightness(int brightness)
bool isWakeLockLevelSupported(int level)

All other methods exposed by `PowerManager` are direct proxies to the service, and have the same prototypes as those defined in `IPowerManager.aidl`. Note the grayed methods are **not** exported via the proxy:

Listing s2-ipm: The `PowerManagerService` methods defined in `IPowerManager.aidl`

```
interface IPowerManager
{
  // WARNING: The first four methods must remain the first three methods because their
  // transaction numbers must not change unless IPowerManager.cpp is also updated.

 /*  1 */ void acquireWakeLock(IBinder lock,int flags,String tag,String pkgeName,in WorkSource ws);
 /*  2 */ void acquireWakeLockWithUid(IBinder lock,int flags,String tag,String pkgName, int uidblame);
 /*  3 */ void releaseWakeLock(IBinder lock, int flags);
 /*  4 */ void updateWakeLockUids(IBinder lock, in int[] uids);
 /*  5 */ void updateWakeLockWorkSource(IBinder lock, in WorkSource ws);
 /*  6 */ boolean isWakeLockLevelSupported(int level);
 /*  7 */ void userActivity(long time, int event, int flags);
 /*  8 */ void wakeUp(long time);
 /*  9 */ void goToSleep(long time, int reason);
 /* 10 */ void nap(long time);
 /* 11 */ boolean isScreenOn();

 /* These require the REBOOT permission (for obvious reasons) */
 /* 12 */ void reboot(boolean confirm, String reason, boolean wait);
 /* 13 */ void shutdown(boolean confirm, boolean wait);
 /* 14 */ void crash(String message);

 /* 15 */ void setStayOnSetting(int val);
 /* 16 */ void setMaximumScreenOffTimeoutFromDeviceAdmin(int timeMs);

 // temporarily overrides the screen brightness settings to allow the user to
 // see the effect of a settings change without applying it immediately
 /* 17 */ void setTemporaryScreenBrightnessSettingOverride(int brightness);
 /* 18 */ void setTemporaryScreenAutoBrightnessAdjustmentSettingOverride(float adj);

 // sets the attention light (used by phone app only)
 /* 19 */ void setAttentionLight(boolean on, int color);
```

The `PowerManagerService` is one of the first services created, but it is not actually initialized until the Lights, Battery, Content Provider and AppOps services are started. Those services are needed throughout the lifecycle for operations (for example, lights for `setAttentionLight`, and others shown in Figure s2-pma) and therefore constitute dependencies. These dependencies are uses internally: a call to `systemReady()` further provides the `TwilightService` and `DreamManager` services, which in turn help support device idle states, wherein these services need to be started. The `PowerManagerService` also uses a `DisplayPowerController` object internally, which helps toggle power to the display, based on certain events, such as idle time, proximity sensing, and others, as discussed later in this chapter. The service spawns a dedicated thread in `system_server` to handle events and notifications from multiple sources.

Wake Locks

Application developers are likely familiar with the concept of WakeLocks (what iOS developers would recognize as `IOPMAssertions`). These are, as far as their consumers are involved, opaque objects which ensure the device will stay awake, or leave the screen on. In practice, wakeLocks are implemented over "suspend blockers", an Androidism provided by the kernel, and discussed in Chapter 18. Along with more advanced functions (such as setting screen brightness), wake locks make use of the `power` Hardware Abstraction Layer (HAL) module (and therefore require JNI), as discussed in Chapter 10.

> ⚠ There is a difference between WakeLocks and **suspend blockers**: The former are the Android objects, whereas the latter are the Linux kernel Androidism which, at the native level, actually keeps the CPU alive, preventing Linux Power Management from kicking in and suspending the device. As an Android particularity, is has been the subject of intense debate with the mainline kernel developers. Linux 3.5 and later offer an alternative called "opportunistic suspend", which will hopefully be picked up by Android.

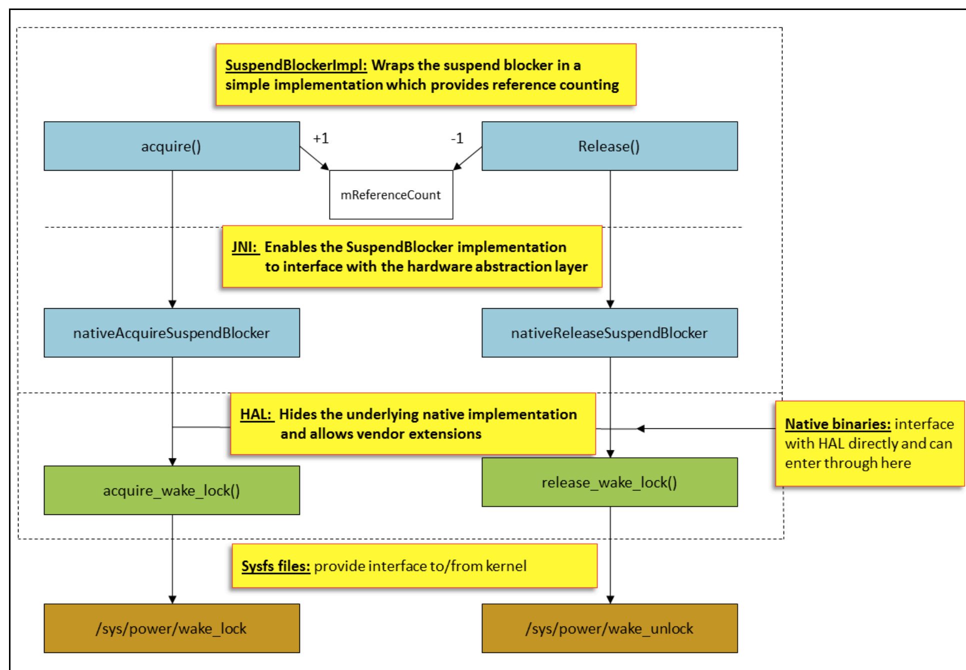
Suspend Blockers can be set from user mode by means of two files in sysfs: /sys/power/wake_lock, and wake_unlock (or, in older Android versions, /sys/android_power/acquire_partial_wake_lock and release_wake_lock). Both these files are owned by `radio:system`. Usage is straightforward: A caller need only open the file, and write the wakelock name into it. Because of the permission restrictions, however, only system processes can write to that file. The wakelocks requested by applications **do not** end up in sysfs files. Instead, the service uses four suspend blockers:

- **`PowerManagerService.WakeLocks`** - Used on behalf of applications requesting a wake lock
- **`PowerManagerService.Display`** - Used during display state transitions
- **`PowerManagerService.Broadcasts`** - used internally by the `Notifier` component, when sending out broadcasts, to prevent the device suspending until receivers can proecss them.
- **`PowerManagerService.WirelessChargerDetector`** - used internally by `WirelessChargerDetector`.

The suspend blockers are kept in an internal `mSuspendBlockers` arrayList. When the `PowerManagerService` routinely updates its state (shown later in Figure s2-upsl), the first phase involves iterating over the list and summarizing the wake locks into a bitmap (`mWakeLockSummary`). The last stage of the update acquires (or releases) the suspend blockers. This is abstracted through a `SuspendBlocker` object, whose implementation performs the operation natively (i.e. via JNI). Other framework components can skip the lengthy path through Binder and invoke the native operation directly. Additional suspend blockers used are `radio-interface`, `SensorService`, `KeyEvents` (owned by the `InputManager`'s `EventHub`, `GPS`, and `FLP` (Fused Location Provider).

Figure s2-awl depicts the process of suspend blocker acquisition:

Figure s2-awl: Acquiring a suspend blocker



Experiment: Observing wake lock activity

The /sys/power/wake_lock and wake_unlock files serve not one, but two purposes - in addition to their use to `write(2)` wake locks to acquire or release, they can also be `read(2)` to show which wake locks are currently acquired, and which have been released:

Output s2-wl: Observing wakelocks through /sys/power

```
# Must be root to view or write to /sys/power/wake* files.
# Phone turned on: Note Display lock, and radio-interface readying
root@htc_m8wl:/ # cat /sys/power/wake_lock
PowerManagerService.Display PowerManagerService.WakeLocks radio-interface
# Screen still on, but radio interface idling
root@htc_m8wl:/ # cat /sys/power/wake_lock
PowerManagerService.Display
# A few seconds later, screen shuts off
root@htc_m8wl:/ # cat /sys/power/wake_lock
# wake_unlock shows all defined, but released locks - including kernel ones
root@htc_m8wl:/ # cat /sys/power/wake_unlock
KeyEvents PowerManagerService.Broadcasts PowerManagerService.Display PowerManagerService.WakeLocks
dmagent_wakelock dofstrim qcril qcril_pre_client_init qmuxd_port_wl_0 qmuxd_port_wl_1 qmuxd_port_wl
qmuxd_port_wl_3 qmuxd_port_wl_4 qmuxd_port_wl_5 qmuxd_port_wl_6 qmuxd_port_wl_7 radio-interface
rmt_storage_-1215036872 rmt_storage_-1215037720
```

Older versions of Android maintain detailed wakelock state in the /proc/wakelocks file entry. This file, however, has been removed in newer Android kernels, in favor of /sys/kernel/debug/wakeup_sources, which contains a verbose output on all sources, not just wakelocks. The author had a (truly marvellous) example of this file to present, but the margins of the page are too narrow. You are therefore encouraged to try this on your device (No root required, as the file is world readable). Table s2-wus shows the columns of this file:

Table s2-wus: Columns of /sys/kernel/debug/wakeup_sources

Column	Value
name	The tag specified for the wakeup source. Usually the name of a driver. For wakelocks, this is the tag provided by the framework
active_count	Counts how many times the wake lock has been active
event_count	Counts how many times the wakeup source event was triggered
wakeup_count	Counts how many times the wakeup source has forced a device wakeup. For wakelocks: 0
expire_count	Counts how many times the wakeup source has expired. For wakelocks: 0
active_since	Count in jiffies (timer ticks) specifying since when source was active
total_time	Count in jiffies specifying total time source was active
max_time	Max time this source has been continuously active
last_change	Timestamp of when source activity was last changed
prevent_suspend_time	How much time could have been saved in suspension if not for this source. This is especially useful for figuring out the impact on battery life.

Keep in mind that the wakeup_sources provides information for all wakeup sources in the kernel - not just wakelocks. In fact, in this file (which also exists on desktop Linux) you're likely to see more driver installed sources there than Android wakelocks.

Application-requested wakelocks, which are obtained via the `PowerManager` proxy object, are objects. The `PowerManagerService` holds them in an internal ArrayList of `WakeLock` objects. Each object contains the details of its requestor (PackageName,Owner UID/PID, Tag, and user specified flags. Most importantly, the `WakeLock` also holds an `IBinder` reference to the caller, and implements the `IBinder.DeathRecipient` interface (i.e. implements the `binderDied()` method), which allows the `PowerManagerService` to `handleWakeLockDeath()` and automatically remove the wakelock (and possibly update the power state) if the calling application has been terminated or crashed without releasing it.

Responding to system events

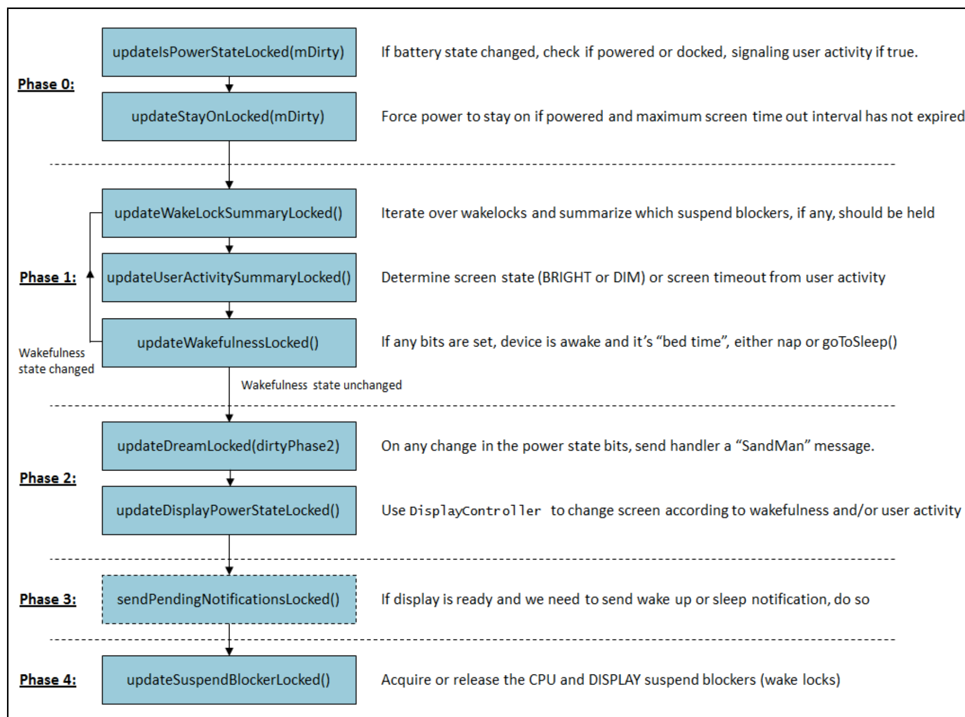
The PowerManagerService doesn't actually do too much until SystemServer indicates the system is ready. Once its systemReady method is called, the PowerManagerService creates a DisplayPowerController object (to toggle screen brightness), a WirelessChargerDetector (for those devices which support it), and registers several broadcast receivers, to install handlers for important system events. It likewise registers content observes from the system Settings provider, so as to respond to changes in system settings*. When a trigger is detected, the PowerManagerService sets corresponding bits in an internal bitmask called mDirty. There are additional bits for other triggers, as shown in Table s2-pm:

Table s2-pm: Event sources possibly leading to update of system power state

Trigger	Source	Corresponding mDirty Bit
ACTION_BATTERY_CHANGED	BatteryService	DIRTY_BATTERY_STATE (0x100) DIRTY_IS_POWERED (0x40)
ACTION_BOOT_COMPLETED	ActivityManagerService	DIRTY_BOOT_COMPLETED (0x10)
ACTION_DREAMING_STARTED ACTION_DREAMING_STOPPED	DreamController	N/A (schedules MSG_SANDMAN)
ACTION_DOCK_EVENT	DockObserver	DIRTY_DOCK_STATE (0x800)
ACTION_USER_SWITCHED	ActivityManagerService	DIRTY_SETTINGS (0x20) on change
SCREENSAVER_ENABLED, SCREENSAVER_ACTIVATE_ON_SLEEP, SCREENSAVER_ACTIVATE_ON_DOCK	Settings.Secure.*	
STAY_ON_WHILE_PLUGGED_IN		DIRTY_SETTINGS (0x20) on change DIRTY_STAY_ON (0x80) if set
SCREEN_BRIGHTNESS, SCREEN_BRIGHTNESS_MODE, SCREEN_OFF_TIMEOUT		DIRTY_SETTINGS (0x20) on change
onProximityPositive callback	DisplayPowerController	DIRTY_PROXIMITY_POSITIVE (0x200)
Wakelock holder death	WakeLock.binderDied()	DIRTY_WAKE_LOCKS (0x01)
Internal wakefulness state	wakeUp() goToSleep() nap()	DIRTY_WAKEFULNESS (0x02)
User Activity	IPowerManager call #7 Native code	DIRTY_USER_ACTIVITY (0x04)

The PowerManagerService uses the broadcast receivers and notification handlers to create an informal state machine. "Informal", because unlike other state machines in the Android framework (notably, the WiFiStateMachine) the transition triggers are encoded by bits in the internal bitmask (mDirty). All these cases lead to updatePowerStateLocked, whose flow can be simplified as in Figure s2-upsl:

Figure s2-upsl: The flow of updatePowerStateLocked()

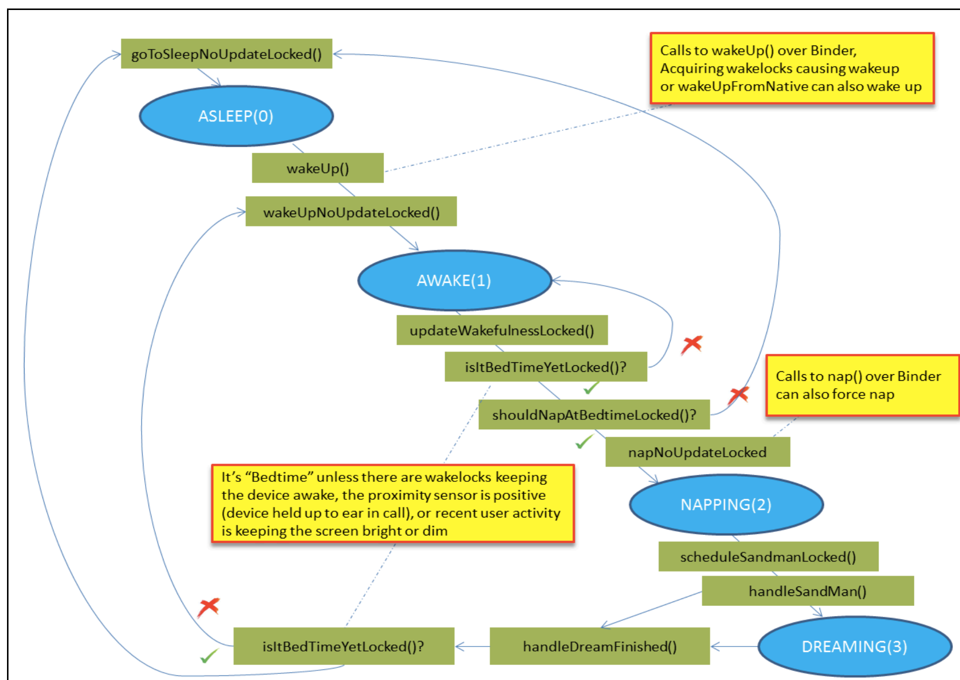


---

* - Yet another trigger occurs when the WindowManager uses private APIs - set....OverrideFromWindowManager, to directly set values for the cached settings values temporarily. These, too, result in a DIRTY_SETTINGS bit setting.

The key phase in updating the power state is the call to `updateWakefulnessLocked()` (at the end of phase 1). This is where the system wake state (`mWakefulness`) is changed. A diagram showing the possible states and a large subset of their transitions are shown in Figure s2-wst:

Figure s2-wst: Wakefulness state transitions



A change often requires recalculation of the state, which is why Phase 1 needs to be potentially re-executed. When the system state has stabilized, updating the power state may continue with Phase 2 and onward. Phase 2 begins with a call to `updateDreamLocked`, which controls the device's "Dream" state: On changes in the `mDirty` bits, it sends a `MSG_SANDMAN` message. If the power manager determines the preconditions for dreaming (dreams are supported and enabled, the device is powered and kept awake, and the display is on) are satisfied, and the device wakefulness has been set to "NAPPING", then a dream sequence can begin, by a call to the `DreamManagerService`.

DreamManagerService

"Dreams" are to Android devices what Screensavers are to desktops. Android introduced support for this feature with API 17 and the `DreamService` class, which allows developers to create custom screensavers. Dream support is configurable from the Settings UI, which mapps to the `screensaver_enabled` setting in /data/data/com.android.providers.settings/databases.

DreamManagerService	
Name:	dreams
Interface:	[android.service.dreams.IDreamManager]()
Proxy:	None
unless:	`disable_noncore`, `!config_dreamsSupported`

The `DreamManagerService` is exactly that - it manages dreams, but does not implement them: Rather, it keeps a simple internal state, and uses a `DreamController` to invoke the dreams. The dreams are implemented as instance of the `DreamService` class, which is an extension of the standard Android service, well detailed in [the documentation](). Implementing a `DreamService` requires the provider to further extend the class so as to override several pre-defined callbacks, which make up the dream lifecycle - i.e. the methods called by the `DreamController()`.

Unlike other services, the `DreamManagerService` has no proxy object, as it is not meant to be called from application context. It does, however, offer an interface over binder, as shown in Listing s2-dms:

Listing s2-dms: The methods exposed by `IDreamManager`

```
interface IDreamManager {
 /* 1 */ void dream();
 /* 2 */ void awaken();
 /* 3 */ void setDreamComponents(in ComponentName[] componentNames);
 /* 4 */ ComponentName[] getDreamComponents();
 /* 5 */ ComponentName getDefaultDreamComponent();
 /* 6 */ void testDream(in ComponentName componentName);
 /* 7 */ boolean isDreaming();
 /* 8 */ void finishSelf(in IBinder token);
}
```

Experiment: Viewing the `DreamManagerService` settings.

The `DreamManagerService` maintains its configuration with `screensaver_` keys in the settings database, which you can easily retrieve (on a rooted device) if you have the SQLite3 binary installed:

Output s2-dsq: Viewing the `DreamManagerService` settings

```
root@android /# sqlite3 /data/data/com.android.providers.settings/databases/settings.db \
                 "select * from secure where name like 'screensaver%'"
30|screensaver_activate_on_dock|1
31|screensaver_activate_on_sleep|0
# Returned by getDefaultDreamComponent()
33|screensaver_default_component|com.google.android.deskclock/com.android.deskclock.Screensav
# Returned by getDreamComponents()
164|screensaver_components|com.android.dreams.basic/com.android.dreams.basic.Colors
# Controls if Dreams are globablly enabled
168|screensaver_enabled|1
```

Although you can use SQLite3 to alter any of these values, the values will not take effect, because the `DreamManagerService` caches them. You can, however, see the values change if you alter dream settings from the GUI. Assuming `screensaver_enabled` is set to 1 (true), and you have a `screensaver_default_component` value, you can use `service call dream 1` to start dreaming, `service call dream 2` to end a dream, and `service call dream 7` to return 0 or 1, depending on whether the device is dreaming. If the device is dreaming, `dumpsys dreams` will display more detailed information:

Output s2-dds: The output of `dumpsys dreams` during a dream

```
root@android /# dumpsys dreams
DREAM MANAGER (dumpsys dreams)

mCurrentDreamToken=android.os.Binder@41be56d8
mCurrentDreamName=ComponentInfo{com.android.dreams.basic/com.android.dreams.basic.Colors}
mCurrentDreamUserId=0
mCurrentDreamIsTest=false

Dreamland:
  mCurrentDream:
    mToken=android.os.Binder@41be56d8
    mName=ComponentInfo{com.android.dreams.basic/com.android.dreams.basic.Colors}
    mIsTest=false
    mUserId=0
    mBound=true
    mService=android.service.dreams.IDreamService$Stub$Proxy@424256b0
    mSentStartBroadcast=true
```

Putting the `DreamService` class lifecycle, `DreamManager` settings and interface calls together, we arrive at Figure 2-dr:

This file is an excerpt from the full book - "Android Internals: A confectioner's cookbook" - which can be found on http://NewAndroidBook.com/. The preview was made available for free as a preview of the book (think of it like Amazon's "Look Inside" :-). . I encourage you to check out Technologeeks.com Android Internals training, which builds on the book and expands it further with Instructor Led Training, as well as Twitter @Technologeeks (or the the RSS feed) for more updates.

You can also preorder the book by emailing p r eorder @ The Book's domain.

Note some links (to other chapters in the book) will not work in this file (since it is partial), but external links will. Feedback, questions and requests are always welcome.

Though this file is provided as a free preview, this is still copyrighted material. Respect the great effort put into researching and elucidating this - and at the very least give credit where due, even if you're not planning to ask permission for using tables or images.