Diving Into Android 'M' Doze

> NOTE: Everything described in this post is based off the Android 'M' Developer Preview, Release 1. As new releases (and the source code) are made available, behaviors are subject to change and the contents of this post may change along with them.

One of the headlining features of the upcoming Android 'M' release is something Google has termed "doze" - a mechanism by which applications are forced into limited activity when the device has been sitting dormant and inactive for a significant amount of time. In this post, we are going to attempt to uncover a bit about how this new feature works at the system level.

What is Doze?

The first time we saw the term "doze" used in Android, it was actually as a display state in KitKat for Wear (API 20), then later on primary devices with Lollipop. It was intended to indicate a new, low-power screen-on mode where the device shows static (non-interactive) content temporarily (think the black/white clock display on the Nexus 6 in response to significant motion, or the ambient watch display that doesn't actually update).

In the new Android 'M' Preview, Doze Mode takes on a slightly different meaning. It now refers to a sort of "forced idle" state where very little background processing is allowed to take place.

> EDITORIAL: Doze ends up being a pretty poor name for the feature, since that term is already all over the framework code in reference to the prior incarnation...which is controlled by the `DreamManager` and has no relation to what is now all over the press as "doze mode".

Say Hello to DeviceIdleController

`DeviceIdleController` is the primary driver of this new feature - which I will now refer to as *device idle mode* instead of *doze mode* since that better fits the code. If you've read the official docs (linked above) you may have noticed the following in the section on testing, which developers can use to manually test their application behavior with the device in this state:

```
$ adb shell dumpsys battery unplug
$ adb shell dumpsys deviceidle step
```

For those not familiar, `dumpsys` is a binary that interacts with system services (by name). It happens that `deviceidle` is a name we hadn't seen before, because it is a new system service:

```
$ adb shell service list | grep deviceidle
59  deviceidle: [android.os.IDeviceIdleController]
```

This new service is registered at all times to listen for the following system events, which can trigger it into (and out of) this new idle mode:

1. Screen on/off
2. Charging status
3. Significant motion detect

`DeviceIdleController` maintains a state machine that contains five states:

1. `ACTIVE` - Device is in use, or connected to a charge source.
2. `INACTIVE` - Device has recently come out of the active state (user turned off the display or unplugged it, for example)
3. `IDLE_PENDING` - Hold on to your hats, we are about to enter idle mode.
4. `IDLE` - Device is idle.

5. `IDLE_MAINTENANCE` - Window is open for applications to do processing.

When the device is awake and in-use, the controller is in the `ACTIVE` state. External events (inactivity timeout, user powering off the screen, etc.) will drive the state machine into `INACTIVE`. At that point, `DeviceIdleController` drives the process internally by setting its own alarms with `AlarmManager`:

1. An alarm will be set for a pre-determined time in the future (30 minutes in the 'M' preview).
2. When this alarm fires, `DeviceIdleController` will advance to `IDLE_PENDING` and set the same alarm again.
3. On the next trigger of this alarm, the controller advances to the `IDLE` state. This is the state where application features are fully restricted.
4. Moving forward, the service will jump between `IDLE` and `IDLE_MAINTENANCE` periodically. The latter is the state where pending application events are triggered before returning to full lockdown.

As noted above, developers can manually advance this state machine with the `step` command:

```
$ adb shell dumpsys deviceidle step
```

We can see all the options the `deviceidle` dump interface supports with the `-h` flag:

```
$ adb shell dumpsys deviceidle -h
Device idle controller (deviceidle) dump options:
  [-h] [CMD]
  -h: print this help text.
Commands:
  step
    Immediately step to next state, without waiting for alarm.
  disable
    Completely disable device idle mode.
  enable
    Re-enable device idle mode after it had previously been disabled.
  whitelist
    Add (prefix with +) or remove (prefix with -) packages.
```

The public API of the service (represented by the `IDeviceIdleController` interface) consists entirely of methods to access a whitelist. Applications (system or otherwise) do not have access to drive the controller state in any way.

Are You on the List?

As you can see from the help menu above, `DeviceIdleController` maintains a whitelist of applications. We can see the current list by dumping the service's state without any extra parameters:

```
$ adb shell dumpsys deviceidle
  Whitelist system apps:
    com.android.providers.downloads
    com.android.vending
    com.google.android.gms
  Whitelist app uids:
    UID=10012: true
    UID=10016: true
    UID=10026: true
  …
```

The list is broken down into two parts: *system apps* and *user apps*.

System Applications

System applications are whitelisted by the platform maker as part of their system configuration definition. Below is an example taken from the Nexus 6, which whitelists the GMS Core (used in GCM), Play Store, and an optional app for power monitoring:

`/system/etc/sysconfig/google.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<!-- These are configurations that must exist on all GMS devices. -->
<config>
    <allow-in-power-save package="com.google.android.gms" />

    <allow-in-power-save package="com.android.vending" />

    <allow-in-power-save package="com.google.android.volta" />
</config>
```

These values are accessible to other system services via the `SystemConfig` instance. `DeviceIdleController` uses `SystemConfig.getAllowInPowerSave()` to pull these system-defined elements into the whitelist.

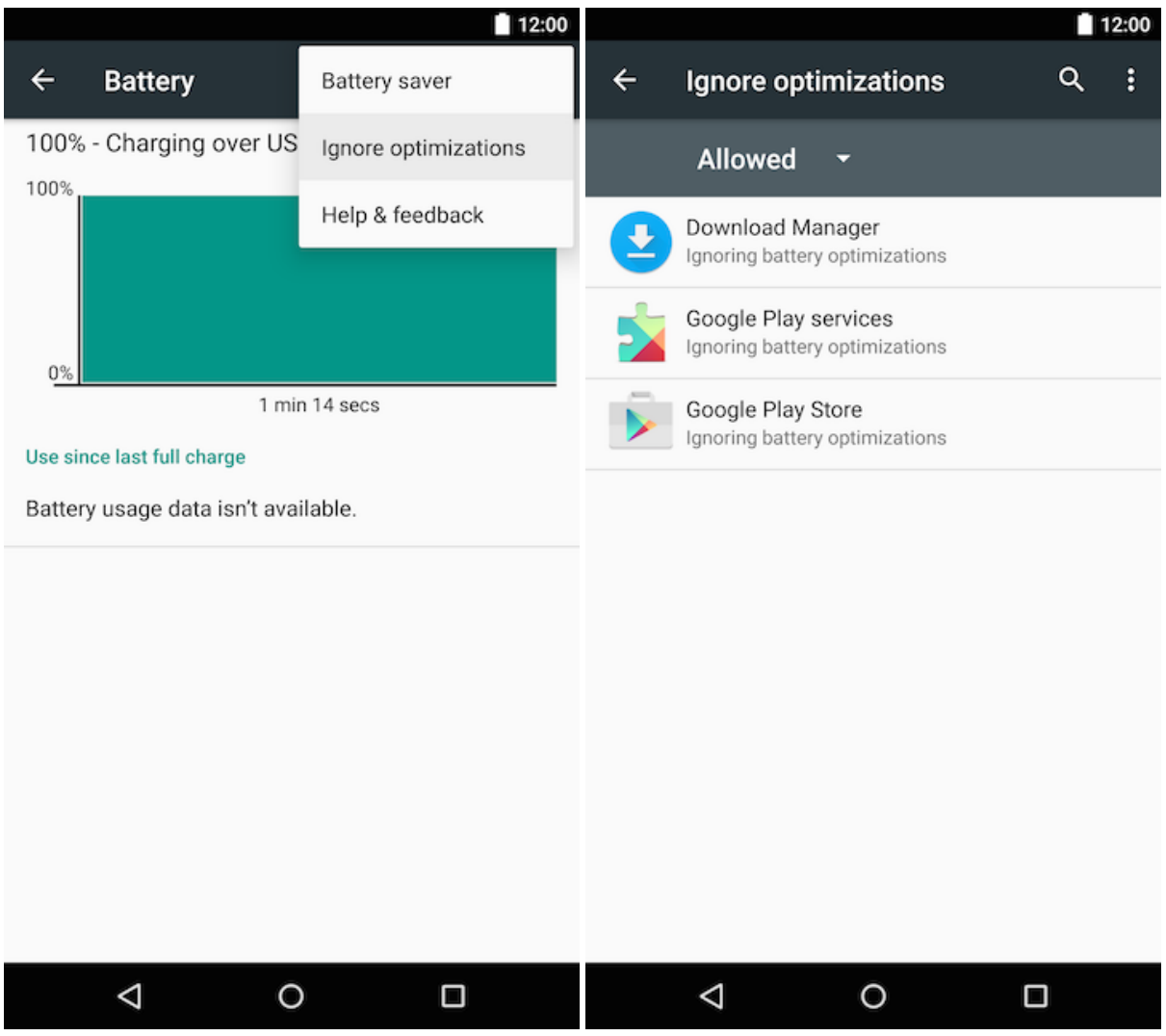
> NOTE: This is the same facility introduced in Lollipop to determine which system applications can have background access when the device is in power save (or "battery saver") mode.

User Applications

The remainder of the whitelist is user-defined, and items can be added/removed in two ways. First, developers can use the `whitelist` command via the `dumpsys` interface:

```
$ adb shell dumpsys deviceidle whitelist +com.example.myapplication
$ adb shell dumpsys deviceidle
  Whitelist system apps:
    com.android.providers.downloads
    com.android.vending
    com.google.android.gms
  Whitelist user apps:
    com.example.myapplication
  Whitelist app uids:
    UID=10012: true
    UID=10016: true
    UID=10026: true
    UID=10101: true
```

Second, users have control (via **Settings -> Battery -> Ignore optimizations**) to modify the whitelist.



This user control primarily exists because the whitelist is also used by the new App Standby feature of Android 'M'. The whitelist is only partially used when evaluating device idle behavior...

System Service Behavior in Device Idle Mode

Now that we know some of the basics, let's examine how the rest of the system services react when the device is idle, and determine if that whitelist is of any real use. The following is a list of the primary services touched by `DeviceIdleController`:

NetworkPolicyManagerService

Lollipop introduced battery saver mode, in which only whitelisted applications could do background processing. Device idle mode piggybacks on this same logic, granting network access only to apps on the whitelist. This service treats battery saver mode and device idle mode as equivalent.

JobSchedulerService

Currently executing jobs in all applications (no exceptions) are canceled. No pending jobs will be started until the device exits idle mode.

SyncManager

Currently active syncs in all applications (no exceptions) are canceled.

PowerManagerService

The application whitelist is used to determine which wake locks are valid. Apps that are not on the list will have their wake locks promptly disabled.

AlarmManagerService

`DeviceIdleController` registers its next wakeup alarm with a special private method (`AlarmManager.setIdleUntil()`). When `AlarmManagerService` sees this, all standard application alarms are forced into a pending state until the next `DeviceIdleController` alarm triggers. In this way, application alarms are batched together and run only on the `IDLE_MAINTENANCE` periods driven by the controller.

There are three flags that allow an alarm to escape this fate:

- `FLAG_ALLOW_WHILE_IDLE` - Set by any application using the new `setAndAllowWhileIdle()` API.
- `FLAG_WAKE_FROM_IDLE` - Set by any application using the `setAlarmClock()` API.
- `FLAG_IDLE_UNTIL` - Reserved for the alarm used by `DeviceIdleController` to advance its state machine.

Processes with a UID < 10000 (i.e. system services) are automatically granted `FLAG_ALLOW_WHILE_IDLE` on every alarm they set.

Final Thought

One thing many developers have asked about is that the docs indicate applications will be granted "brief network access" during idle if they receive high-priority messages using GCM. For my part, there is nothing I can see in the framework implementation that is linked to GCM explicitly (thank goodness, because that would be silly, right?). Since the GMS applications are closed source and obfuscated, we can only theorize how this will happen on Google devices.

As it stands, any system application could temporarily modify the app whitelist or network policy for a specific application. This would disable the networking restriction on a per-app basis without modifying the current idle state of the device. It is also a bit unclear in the current preview if there is still an open hole in the SyncManager. Existing syncs are canceled, but I'm not certain new syncs wouldn't be allowed. Applications *may* still be able to execute a new sync from a GCM trigger (which is one of the only apps allowed network access in this state).

...all of that, of course, is purely speculation.