Android Developers

Keeping the Device Awake

To avoid draining the battery, an Android device that is left idle quickly falls asleep. However, there are times when an application needs to wake up the screen or the CPU and keep it awake to complete some work.

The approach you take depends on the needs of your app. However, a general rule of thumb is that you should use the most lightweight approach possible for your app, to minimize your app's impact on system resources. The following sections describe how to handle the cases where the device's default sleep behavior is incompatible with the requirements of your app.

This lesson teaches you to

- Keep the Screen On
- Keep the CPU On

Try it out

DOWNLOAD THE SAMPLE

Scheduler.zip

Keep the Screen On

Certain apps need to keep the screen turned on, such as games or movie apps. The best way to do this is to use the `FLAG_KEEP_SCREEN_ON` (https://developer.android.com/reference/android/view/WindowManager.LayoutParams.html#FLAG_KEEP_SCREEN_ON) in your activity (and only in an activity, never in a service or other app component). For example:

```
public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
  }
```

The advantage of this approach is that unlike wake locks (discussed in Keep the CPU On (#cpu)), it doesn't require special permission, and the platform correctly manages the user moving between applications, without your app needing to worry about releasing unused resources.

Another way to implement this is in your application's layout XML file, by using the `android:keepScreenOn` (https://developer.android.com/reference/android/R.attr.html#keepScreenOn) attribute:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:keepScreenOn="true">
    ...
</RelativeLayout>
```

Using `android:keepScreenOn="true"` is equivalent to using `FLAG_KEEP_SCREEN_ON` (https://developer.android.com/reference/android/view/WindowManager.LayoutParams.html#FLAG_KEEP_SCREEN_ON). You can use whichever approach is best for your app. The advantage of setting the flag programmatically in your activity is that it gives you the option of programmatically clearing the flag later and thereby allowing the screen to turn off.

> **Note:** You don't need to clear the `FLAG_KEEP_SCREEN_ON` (https://developer.android.com/reference/android/view/WindowManager.LayoutParams.html#FLAG_KEEP_SCREEN_ON) flag unless you no longer want the screen to stay on in your running application (for example, if you want the screen to time out after a certain period of inactivity). The window manager takes care of ensuring that the right things happen when the app goes into the background or returns to the foreground. But if you want to explicitly clear the flag and thereby allow the screen to turn off again, use `clearFlags()` (https://developer.android.com/reference/android/view/Window.html#clearFlags(int)): `getWindow().clearFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON)`.

Keep the CPU On

If you need to keep the CPU running in order to complete some work before the device goes to sleep, you can use a `PowerManager` (https://developer.android.com/reference/android/os/PowerManager.html) system service feature called wake locks. Wake locks allow your application to control the power state of the host device.

Creating and holding wake locks can have a dramatic impact on the host device's battery life. Thus you should use wake locks only when strictly necessary and hold them for as short a time as possible. For example, you should never need to use a wake lock in an activity. As described above, if you want to keep the screen on in your activity, use `FLAG_KEEP_SCREEN_ON` (https://developer.android.com/reference/android/view/WindowManager.LayoutParams.html#FLAG_KEEP_SCREEN_ON).

One legitimate case for using a wake lock might be a background service that needs to grab a wake lock to keep the CPU running to do work while the screen is off. Again, though, this practice should be minimized because of its impact on battery life.

To use a wake lock, the first step is to add the `WAKE_LOCK` (https://developer.android.com/reference/android/Manifest.permission.html#WAKE_LOCK) permission to your application's manifest file:

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

If your app includes a broadcast receiver that uses a service to do some work, you can manage your wake lock through a `WakefulBroadcastReceiver` (https://developer.android.com/reference/android/support/v4/content/WakefulBroadcastReceiver.html), as described in Using a WakefulBroadcastReceiver (#wakeful). This is the preferred approach. If your app doesn't follow that pattern, here is how you set a wake lock directly:

```
PowerManager powerManager = (PowerManager) getSystemService(POWER_SERVICE);
WakeLock wakeLock = powerManager.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
        "MyWakelockTag");
wakeLock.acquire();
```

To release the wake lock, call `wakelock.release()` (https://developer.android.com/reference/android/os/PowerManager.WakeLock.html#release()). This releases your claim to the CPU. It's important to release a wake lock as soon as your app is finished using it to avoid draining the battery.

Alternatives to using wake locks

- If your app is performing long-running HTTP downloads, consider using `DownloadManager` (https://developer.android.com/reference/android/app/DownloadManager.html).
- If your app is synchronizing data from an external server, consider creating a sync adapter (https://developer.android.com/training/sync-adapters/index.html).
- If your app relies on background services, consider using repeating alarms (https://developer.android.com/training/scheduling/alarms.html) or Google Cloud Messaging (https://developer.android.com/google/gcm/index.html) to trigger these services at specific intervals.

Using WakefulBroadcastReceiver

Using a broadcast receiver in conjunction with a service lets you manage the life cycle of a background task.

A `WakefulBroadcastReceiver` (https://developer.android.com/reference/android/support/v4/content/WakefulBroadcastReceiver.html) is a special type of broadcast receiver that takes care of creating and managing a `PARTIAL_WAKE_LOCK` (https://developer.android.com/reference/android/os/PowerManager.html#PARTIAL_WAKE_LOCK) for your app. A `WakefulBroadcastReceiver` (https://developer.android.com/reference/android/support/v4/content/WakefulBroadcastReceiver.html) passes off the work to a `Service` (https://developer.android.com/reference/android/app/Service.html) (typically an `IntentService` (https://developer.android.com/reference/android/app/IntentService.html)), while ensuring that the device does not go back to sleep in the transition. If you don't hold a wake lock while transitioning the work to a service, you are effectively allowing the device to go back to sleep before the work completes. The net result is that the app might not finish doing the work until some arbitrary point in the future, which is not what you want.

The first step in using a `WakefulBroadcastReceiver` (https://developer.android.com/reference/android/support/v4/content/WakefulBroadcastReceiver.html) is to add it to your manifest, as with any other broadcast receiver:

```
<receiver android:name=".MyWakefulReceiver"></receiver>
```

The following code starts `MyIntentService` with the method `startWakefulService()` (https://developer.android.com/reference/android/support/v4/content/WakefulBroadcastReceiver.html#startWakefulService(android.content.Context, android.content.Intent)). This method

is comparable to `startService()` (https://developer.android.com/reference/android/content/Context.html#startService(android.content.Intent)), except that the `WakefulBroadcastReceiver` (https://developer.android.com/reference/android/support/v4/content/WakefulBroadcastReceiver.html) is holding a wake lock when the service starts. The intent that is passed with `startWakefulService()` (https://developer.android.com/reference/android/support/v4/content/WakefulBroadcastReceiver.html#startWakefulService(android.content.Context, android.content.Intent)) holds an extra identifying the wake lock:

```
public class MyWakefulReceiver extends WakefulBroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {

        // Start the service, keeping the device awake while the service is
        // launching. This is the Intent to deliver to the service.
        Intent service = new Intent(context, MyIntentService.class);
        startWakefulService(context, service);
    }
}
```

When the service is finished, it calls `MyWakefulReceiver.completeWakefulIntent()` (https://developer.android.com/reference/android/support/v4/content/WakefulBroadcastReceiver.html#completeWakefulIntent(android.content.Intent)) to release the wake lock. The `completeWakefulIntent()` (https://developer.android.com/reference/android/support/v4/content/WakefulBroadcastReceiver.html#completeWakefulIntent(android.content.Intent)) method has as its parameter the same intent that was passed in from the `WakefulBroadcastReceiver` (https://developer.android.com/reference/android/support/v4/content/WakefulBroadcastReceiver.html):

```
public class MyIntentService extends IntentService {
    public static final int NOTIFICATION_ID = 1;
    private NotificationManager mNotificationManager;
    NotificationCompat.Builder builder;
    public MyIntentService() {
        super("MyIntentService");
    }
    @Override
    protected void onHandleIntent(Intent intent) {
        Bundle extras = intent.getExtras();
        // Do the work that requires your app to keep the CPU running.
        // ...
        // Release the wake lock provided by the WakefulBroadcastReceiver.
        MyWakefulReceiver.completeWakefulIntent(intent);
    }
}
```