This is doable but somewhat dangerous, in that too many on-boot requests slow down the device startup and may make things sluggish for the user. Moreover, the more services that are running all the time, the worse the device performance will be.

A better pattern is to get control on boot to arrange for a service to do something periodically using the `AlarmManager` or via other system events. In this section, we will examine the on-boot portion of the problem – in the next chapter, we will investigate `AlarmManager` and how it can keep services active yet not necessarily resident in memory all the time.

The Permission

In order to be notified when the device has completed is system boot process, you will need to request the `RECEIVE_BOOT_COMPLETED` permission. Without this, even if you arrange to receive the boot broadcast Intent, it will not be dispatched to your receiver.

As the Android documentation describes it:

> *Though holding this permission does not have any security implications, it can have a negative impact on the user experience by increasing the amount of time it takes the system to start and allowing applications to have themselves running without the user being aware of them. As such, you must explicitly declare your use of this facility to make that visible to the user.*

The Receiver Element

There are two ways you can receive a broadcast `Intent`. One is to use `registerReceiver()` from an existing `Activity`, `Service`, or `ContentProvider`. The other is to register your interest in the `Intent` in the manifest in the form of a `<receiver>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.sysevents.boot"
      android:versionCode="1"
      android:versionName="1.0">
  <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
    <application android:label="@string/app_name">
        <receiver android:name=".OnBootReceiver">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED" />
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

The above `AndroidManifest.xml`, from the `SystemEvents/OnBoot` sample project, shows that we have registered a broadcast receiver named `OnBootReceiver`, set to be given control when the `android.intent.action.BOOT_COMPLETED` Intent is broadcast.

In this case, we have no choice but to implement our receiver this way – by the time any of our other components (e.g., an `Activity`) were to get control and be able to call `registerReceiver()`, the `BOOT_COMPLETED Intent` will be long gone.

The Receiver Implementation

Now that we have told Android that we would like to be notified when the boot has completed, and given that we have been granted permission to do so by the user, we now need to actually do something to receive the Intent. This is a simple matter of creating a `BroadcastReceiver`, such as seen in the `OnBootCompleted` implementation shown below:

```
package com.commonsware.android.sysevents.boot;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;

public class OnBootReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    Log.d("OnBootReceiver", "Hi, Mom!");
```

```
  }
}
```

A `BroadcastReceiver` is not a `Context`, and so it gets passed a suitable `Context` object in `onReceive()` to use for accessing resources and the like. The `onReceive()` method also is passed the `Intent` that caused our `BroadcastReceiver` to be created, in case there are "extras" we need to pull out (none in this case).

In `onReceive()`, we can do whatever we want, subject to some limitations:

1. We are not a `Context`, like an `Activity`, so we cannot modify a UI or anything such as that
2. If we want to do anything significant, it is better to delegate that logic to a service that we start from here (e.g., calling `startService()` on the supplied `Context`) rather than actually doing it here, since `BroadcastReceiver` implementations need to be fast
3. We cannot start any background threads, directly or indirectly, since the `BroadcastReceiver` gets discarded as soon as `onReceive()` returns

In this case, we simply log the fact that we got control. In the next chapter, we will see what else we can do at boot time, to ensure one of our services gets control later on as needed.

To test this, install it on an emulator (or device), shut down the emulator, then restart it.

I Sense a Connection Between Us...

Generally speaking, Android applications do not care what sort of Internet connection is being used – 3G, GPRS, WiFi, lots of trained carrier pigeons, or whatever. So long as there is an Internet connection, the application is happy.

Sometimes, though, you may specifically want WiFi. This would be true if your application is bandwidth-intensive and you want to ensure that, should WiFi stop being available, you cut back on your work so as not to consume too much 3G/GPRS bandwidth, which is usually subject to some sort of cap or metering.

There is an `android.net.wifi.WIFI_STATE_CHANGED Intent` that will be broadcast, as the name suggests, whenever the state of the WiFi connection changes. You can arrange to receive this broadcast and take appropriate steps within your application.

This `Intent` requires no special permission, unlike the `BOOT_COMPLETED` Intent from the previous section. Hence, all you need to do is register a `BroadcastReceiver` for `android.net.wifi.WIFI_STATE_CHANGED`, either via `registerReceiver()`, or via the `<receiver>` element in `AndroidManifest.xml`, such as the one shown below, from the `SystemEvents/OnWiFiChange` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.sysevents.wifi"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:label="@string/app_name">
        <receiver android:name=".OnWiFiChangeReceiver">
            <intent-filter>
                <action android:name="android.net.wifi.WIFI_STATE_CHANGED" />
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

All we do in the manifest is tell Android to create an `OnWiFiChangeReceiver` object when a `android.net.wifi.WIFI_STATE_CHANGED Intent` is broadcast, so the receiver can do something useful.

In the case of `OnWiFiChangeReceiver`, it examines the value of the `EXTRA_WIFI_STATE` "extra" in the supplied `Intent` and logs an appropriate message:

```
package com.commonsware.android.sysevents.wifi;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.net.wifi.WifiManager;
import android.util.Log;

public class OnWiFiChangeReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    int state=intent.getIntExtra(WifiManager.EXTRA_WIFI_STATE, -1);
    String msg=null;

    switch (state) {
      case WifiManager.WIFI_STATE_DISABLED:
        msg="is disabled";
        break;

      case WifiManager.WIFI_STATE_DISABLING:
        msg="is disabling";
        break;

      case WifiManager.WIFI_STATE_ENABLED:
        msg="is enabled";
        break;

      case WifiManager.WIFI_STATE_ENABLING:
        msg="is enabling";
        break;

      case WifiManager.WIFI_STATE_UNKNOWN :
        msg="has an error";
        break;

      default:
        msg="is acting strangely";
        break;
    }

    if (msg!=null) {
      Log.d("OnWiFiChanged", "WiFi "+msg);
    }
  }
}
```

The `EXTRA_WIFI_STATE` "extra" tells you what the state has become (e.g., we are now disabling or are now disabled), so you can take appropriate steps in your application.

Note that, to test this, you will need an actual Android device, as the emulator does not specifically support simulating WiFi connections.

Feeling Drained

One theme with system events is to use them to help make your users happier by reducing your impacts on the device while the device is not in a great state. In the preceding section, we saw how you could find out when WiFi was disabled, so you might not use as much bandwidth when on 3G/GPRS. However, not every application uses so much bandwidth as to make this optimization worthwhile.

However, most applications are impacted by battery life. Dead batteries run no apps.

So whether you are implementing a battery monitor or simply want to discontinue background operations when the battery gets low, you may wish to find out how the battery is doing.

There is an `ACTION_BATTERY_CHANGED` `Intent` that gets broadcast as the battery status changes, both in terms of charge (e.g., 80% charged) and charging (e.g., the device is now plugged into AC power). You simply need to register to receive this `Intent` when it is broadcast, then take appropriate steps.

One of the limitations of `ACTION_BATTERY_CHANGED` is that you have to use `registerReceiver()` to set up a `BroadcastReceiver` to get this `Intent` when broadcast. You cannot use a manifest-declared receiver as shown in the preceding two sections.

In `SystemEvents/OnBattery`, you will find a layout containing a `ProgressBar`, a `TextView`, and an `ImageView`, to serve as a battery monitor:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
```

```
  >
  <ProgressBar android:id="@+id/bar"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    >
    <TextView android:id="@+id/level"
      android:layout_width="0px"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:textSize="16pt"
    />
    <ImageView android:id="@+id/status"
      android:layout_width="0px"
      android:layout_height="wrap_content"
      android:layout_weight="1"
    />
  </LinearLayout>
</LinearLayout>
```

This layout is used by a `BatteryMonitor` activity, which registers to receive the `ACTION_BATTERY_CHANGED` `Intent` in `onResume()` and unregisters in `onPause()`:

```
package com.commonsware.android.sysevents.battery;

import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;
import android.os.BatteryManager;
import android.widget.ProgressBar;
import android.widget.ImageView;
import android.widget.TextView;

public class BatteryMonitor extends Activity {
  private ProgressBar bar=null;
  private ImageView status=null;
  private TextView level=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    bar=(ProgressBar)findViewById(R.id.bar);
```

```
    status=(ImageView)findViewById(R.id.status);
    level=(TextView)findViewById(R.id.level);
  }

  @Override
  public void onResume() {
    super.onResume();

    registerReceiver(onBatteryChanged,
                     new IntentFilter(Intent.ACTION_BATTERY_CHANGED));
  }

  @Override
  public void onPause() {
    super.onPause();

    unregisterReceiver(onBatteryChanged);
  }

  BroadcastReceiver onBatteryChanged=new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
      int pct=100*intent.getIntExtra("level", 1)/intent.getIntExtra("scale", 1);

      bar.setProgress(pct);
      level.setText(String.valueOf(pct));

      switch(intent.getIntExtra("status", -1)) {
        case BatteryManager.BATTERY_STATUS_CHARGING:
          status.setImageResource(R.drawable.charging);
          break;

        case BatteryManager.BATTERY_STATUS_FULL:
          int plugged=intent.getIntExtra("plugged", -1);

          if (plugged==BatteryManager.BATTERY_PLUGGED_AC ||
              plugged==BatteryManager.BATTERY_PLUGGED_USB) {
            status.setImageResource(R.drawable.full);
          }
          else {
            status.setImageResource(R.drawable.unplugged);
          }
          break;

        default:
          status.setImageResource(R.drawable.unplugged);
          break;
      }
    }
  };
}
```

The key to `ACTION_BATTERY_CHANGED` is in the "extras". Many "extras" are packaged in the `Intent`, to describe the current state of the battery, such as:

- `health`, which should generally be `BATTERY_HEALTH_GOOD`
- `level`, which is the proportion of battery life remaining as an integer, specified on the scale described by the scale "extra"
- `plugged`, which will indicate if the device is plugged into AC power (`BATTERY_PLUGGED_AC`) or USB power (`BATTERY_PLUGGED_USB`)
- `scale`, which indicates the maximum possible value of level (e.g., `100`, indicating that level is a percentage of charge remaining)
- `status`, which will tell you if the battery is charging (`BATTERY_STATUS_CHARGING`), full (`BATTERY_STATUS_FULL`), or discharging (`BATTERY_STATUS_DISCHARGING`)
- `technology`, which indicates what sort of battery is installed (e.g., `"Li-Ion"`)
- `temperature`, which tells you how warm the battery is, in tenths of a degree Celsius (e.g., `213` is 21.3 degrees Celsius)
- `voltage`, indicating the current voltage being delivered by the battery, in millivolts

In the case of `BatteryMonitor`, when we receive an `ACTION_BATTERY_CHANGED` Intent, we do three things:

1. We compute the percentage of battery life remaining, by dividing the level by the scale
2. We update the `ProgressBar` and `TextView` to display the battery life as a percentage
3. We display an icon, with the icon selection depending on whether we are charging (`status` is `BATTERY_STATUS_CHARGING`), full but on the charger (`status` is `BATTERY_STATUS_FULL` and `plugged` is `BATTERY_PLUGGED_AC` or `BATTERY_PLUGGED_USB`), or are not plugged in

This only really works on a device, where you can plug and unplug it, plus get a varying charge level:

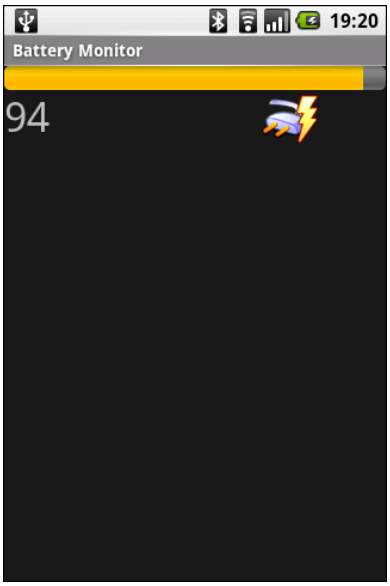

Figure 41. The BatteryMonitor application