

Amended Exhibit C - U.S. Patent No. 9,609,544 (“’544 Patent”)

Accused Instrumentalities: Samsung Galaxy phones and tablets, and all versions and variations thereof since the issuance of the asserted patent.

Claim 1

Issued Claim(s)	Public Documentation
1. A wireless end-user device, comprising:	The Accused Instrumentalities are each “a wireless end-user device.” For example, the Galaxy S22 is a “wireless end-user device.” 
[1a] a wireless modem to communicate data for network service usage activities between the device and a wireless network;	The Accused Instrumentalities comprise “a wireless modem to communicate data for network service usage activities between the device and a wireless network.” For example, the Galaxy S22 includes a wireless modem for communicating with mobile service base stations.

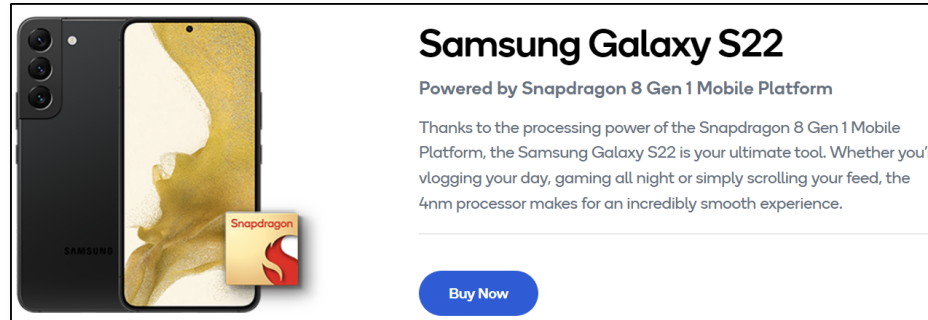
Network & Connectivity	5G
	5G Non-Standalone (NSA), Standalone (SA), Sub6 / mmWave
	LTE
	Enhanced 4x4 MIMO, Up to 7CA, LTE Cat.20 Up to 2.0Gbps Download / Up to 200Mbps Upload
	Wi-Fi
	Wi-Fi 802.11 a/b/g/n/ac/ax 2.4G+5GHz+6GHz, HE160, MIMO, 1024-QAM Up to 2.4Gbps Download / Up to 2.4Gbps Upload
	Bluetooth
	Bluetooth® v 5.2, USB type-C, NFC, Location(GPS, Galileo, Glonass, BeiDou)
	Ultra Wide Band
	*Requires optimal connection. Actual speed may vary depending on country, carrier and user environment. *The bandwidths supported by the device may vary depending on the region or service provider. *Download and upload speeds reaching up to 2.4Gbps only available with Wi-Fi 6E. Wi-Fi 6E only supported on Galaxy S22 Ultra and S22+. Galaxy S22 has Wi-Fi 6. *Galileo and BeiDou coverage may be limited. BeiDou may not be available for certain countries.

<https://www.samsung.com/us/smartphones/galaxy-s22/models/>

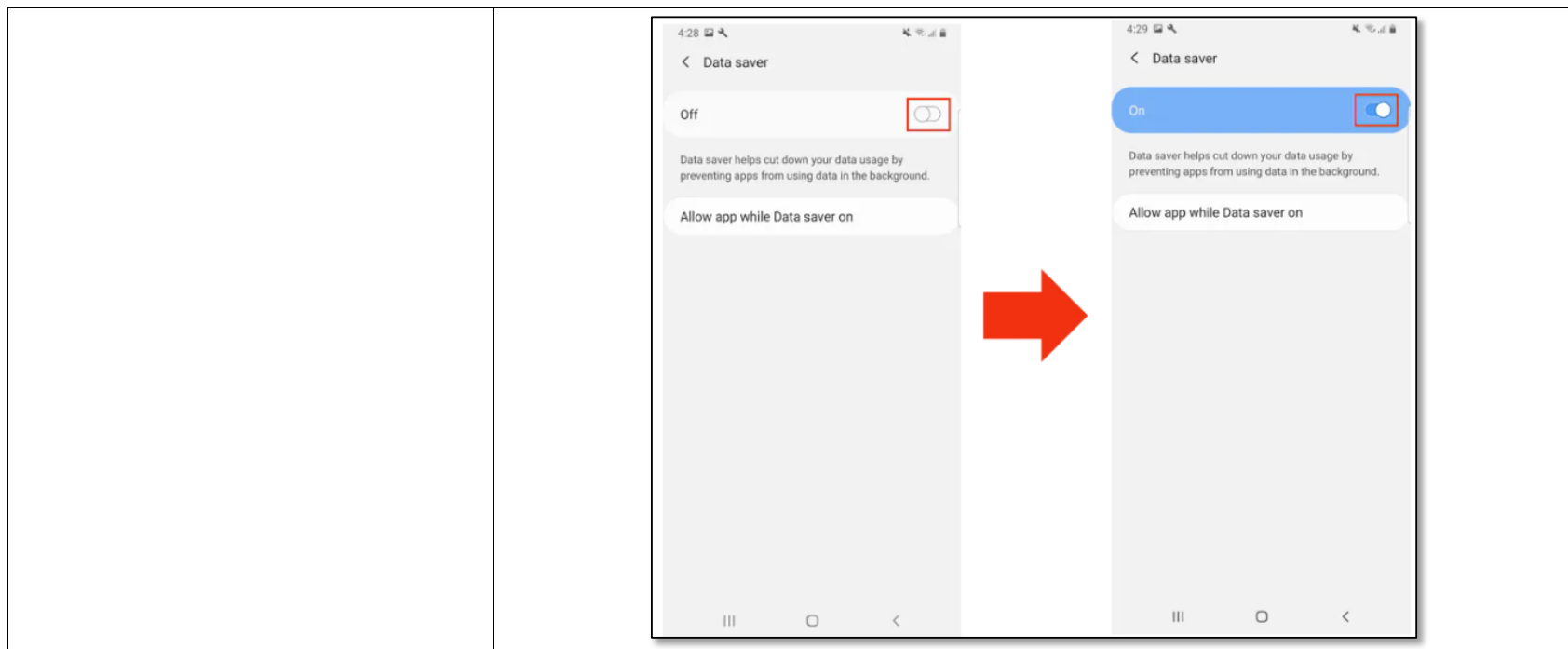
[1b] a processor that executes instructions to associate network service usage activity, on behalf of a first device application, and that occurs when the first device application is not in the foreground of user interaction, with a network service usage control policy,

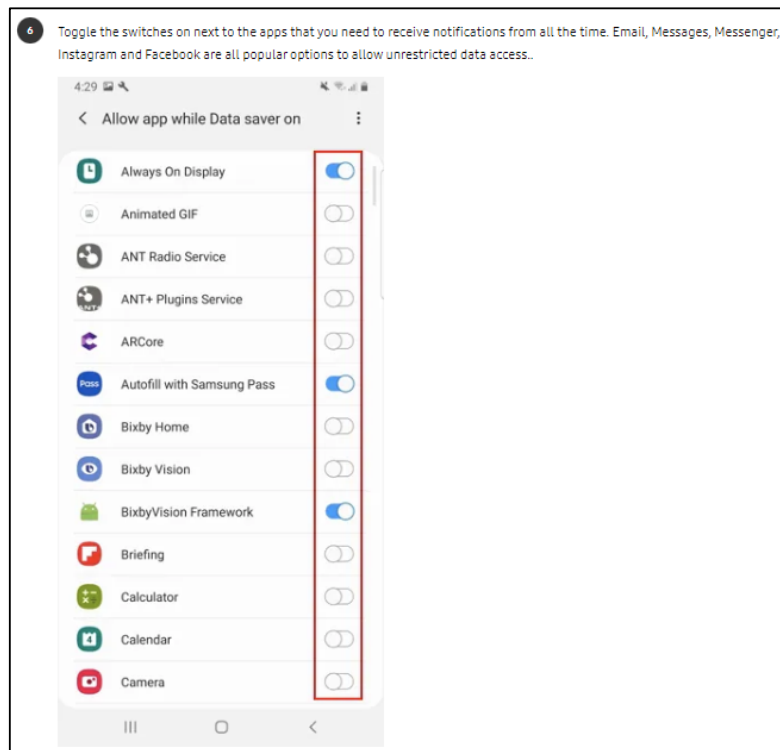
The Accused Instrumentalities comprise “a processor that executes instructions to associate network service usage activity, on behalf of a first device application, and that occurs when the first device application is not in the foreground of user interaction, with a network service usage control policy.”

For example, the Galaxy S22 uses a Snapdragon 8 (Gen 1) processor manufactured by Qualcomm. See <https://www.qualcomm.com/snapdragon/device-finder/smartphones/samsung-galaxy-s22>.



The Galaxy S22’s processor executes instructions that associate network service usage activity, on behalf of a first device application, and that occurs when the first device application is not in the foreground of user interaction, with a network service usage control policy, as shown by the exemplary citations below.





<https://www.samsung.com/ae/support/mobile-devices/android-pie-what-is-the-data-saver-feature/>.

For further example, the Android operating system running on Samsung Galaxy phones and tablets associate network service usage activity, on behalf of a first device application, and that occurs when the first device application is not in the foreground of user interaction, with a network service usage control policy, as shown by the exemplary citations below. As further illustration, the Accused Devices are capable of operating in modes, including without limitation App Standby, wherein application network access is associated with network access policies.

1. A **foreground process** is one that is required for what the user is currently doing. Various application components can cause its containing process to be considered foreground in different ways. A process is considered to be in the foreground if any of the following conditions hold:

- It is running an **Activity** at the top of the screen that the user is interacting with (its **onResume()** method has been called).
- It has a **BroadcastReceiver** that is currently running (its **BroadcastReceiver.onReceive()** method is executing).
- It has a **Service** that is currently executing code in one of its callbacks (**Service.onCreate()**, **Service.onStart()**, or **Service.onDestroy()**).

There will only ever be a few such processes in the system, and these will only be killed as a last resort if memory is so low that not even these processes can continue to run. Generally, at this point, the device has reached a memory paging state, so this action is required in order to keep the user interface responsive.

<https://developer.android.com/guide/components/activities/process-lifecycle>;

Definition of background work

An app is running in the *background* when both the following conditions are satisfied:

- None of the app's activities are currently visible to the user.
- The app isn't running any **foreground services** that started while an activity from the app was visible to the user.

Otherwise, the app is running in the *foreground*.

<https://developer.android.com/guide/background>;

Services overview

A [Service](#) is an [application component](#) that can perform long-running operations in the background. It does not provide a user interface. Once started, a service might continue running for some time, even after the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC). For example, a service can handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

Types of Services

These are the three different types of services:

Foreground

A foreground service performs some operation that is noticeable to the user. For example, an audio app would use a foreground service to play an audio track. Foreground services must display a [Notification](#). Foreground services continue running even when the user isn't interacting with the app.

When you use a foreground service, you must display a notification so that users are actively aware that the service is running. This notification cannot be dismissed unless the service is either stopped or removed from the foreground.

Learn more about how to configure [foreground services](#) in your app.

★ **Note:** The [WorkManager](#) API offers a flexible way of scheduling tasks, and is able to [run these jobs as foreground services](#) if needed. In many cases, using WorkManager is preferable to using foreground services directly.

Background

A background service performs an operation that isn't directly noticed by the user. For example, if an app used a service to compact its storage, that would usually be a background service.

★ **Note:** If your app targets API level 26 or higher, the system imposes [restrictions on running background services](#) when the app itself isn't in the foreground. In most situations, for example, you shouldn't [access location information from the background](#). Instead, [schedule tasks using WorkManager](#).

	https://developer.android.com/guide/components/services;
--	---

App Standby Buckets

Android 9 (API level 28) and higher support **App Standby Buckets**. App Standby Buckets help the system prioritize apps' requests for resources based on how recently and how frequently the apps are used. Based on app usage patterns, each app is placed in one of five priority **buckets**. The system limits the device resources available to each app based on which bucket the app is in.

Priority buckets

The system dynamically assigns each app to a priority bucket, reassigning the apps as needed. The system may rely on a preloaded app that uses machine learning to determine how likely each app is to be used, and assigns apps to the appropriate buckets. If the system app is not present on a device, the system defaults to sorting apps based on how recently they were used. More active apps are assigned to buckets that give the apps higher priority, making more system resources available to the app. In particular, the bucket determines how frequently the app's jobs run, and how often the app can trigger alarms. These restrictions apply only while the device is on battery power; the system does not impose these restrictions on apps while the device is charging.

★ **Note:** Every manufacturer can set their own criteria for how non-active apps are assigned to buckets. You should not try to influence which bucket your app is assigned to. Instead, focus on making sure your app behaves well in whatever bucket it might be in. Your app can find out what bucket it's currently in by calling [UsageStatsManager.getAppStandbyBucket\(\)](#).

The buckets are:

1. **Active:** App is currently being used or was very recently used.
2. **Working set:** App is in regular use.
3. **Frequent:** App is often used, but not every day.
4. **Rare:** App is not frequently used.
5. **Restricted:** App consumes a great deal of system resources, or may exhibit undesirable behavior.

In addition, there's a special **never** bucket for apps that have been installed but have never been run. The system imposes severe restrictions on these apps.

[https://developer.android.com/topic/performance/appstandby;](https://developer.android.com/topic/performance/appstandby)

ConnectivityManager



Added in API level 1

[Kotlin](#) | [Java](#)

```
public class ConnectivityManager  
extends Object
```

[java.lang.Object](#)

↳ [android.net.ConnectivityManager](#)

Class that answers queries about the state of network connectivity. It also notifies applications when network connectivity changes.

The primary responsibilities of this class are to:

1. Monitor network connections (Wi-Fi, GPRS, UMTS, etc.)
2. Send broadcast intents when network connectivity changes
3. Attempt to "fail over" to another network when connectivity to a network is lost
4. Provide an API that allows applications to query the coarse-grained or fine-grained state of the available networks
5. Provide an API that allows applications to request and select networks for their data traffic

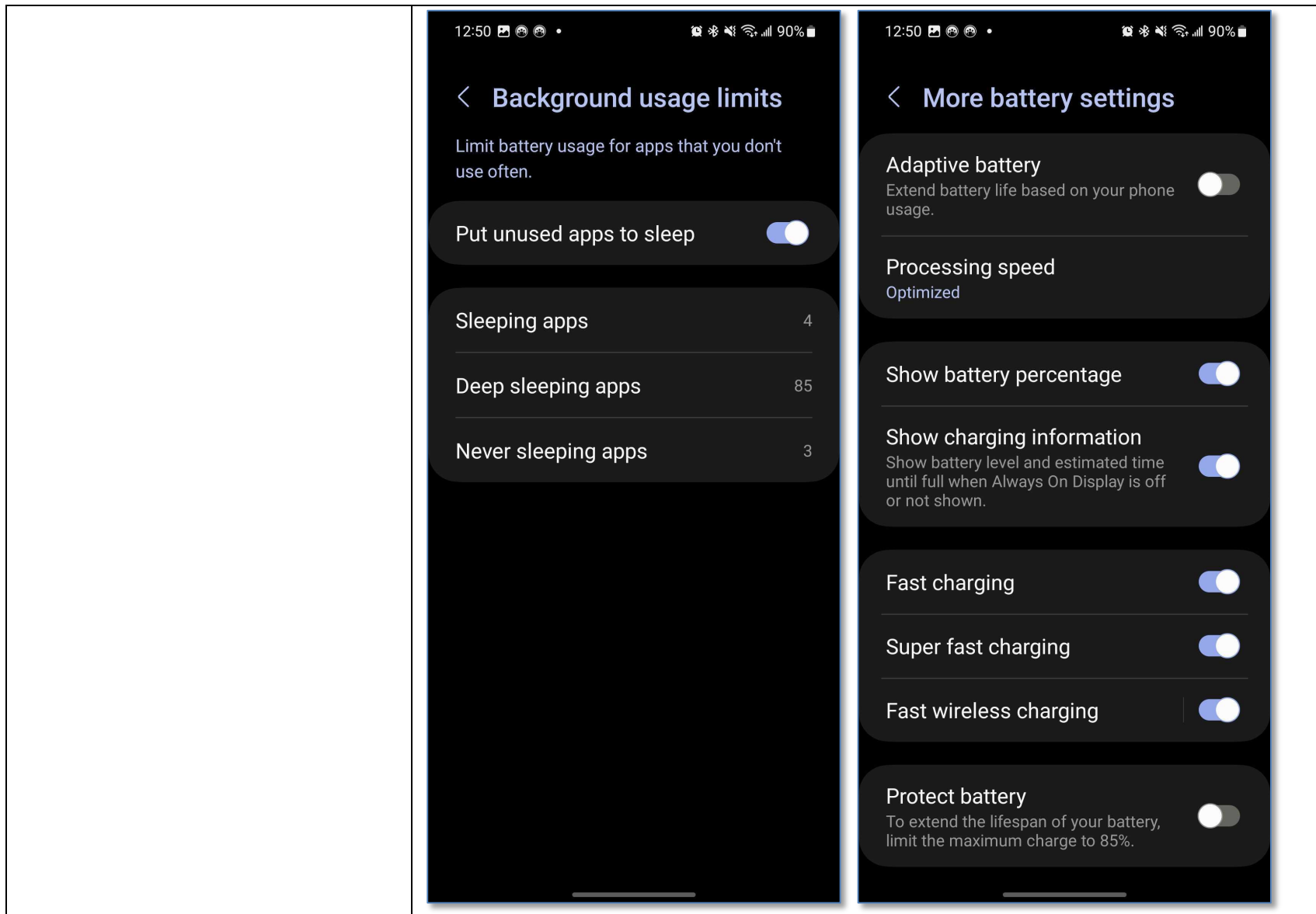
[https://developer.android.com/reference/android/net/ConnectivityManager;](https://developer.android.com/reference/android/net/ConnectivityManager)

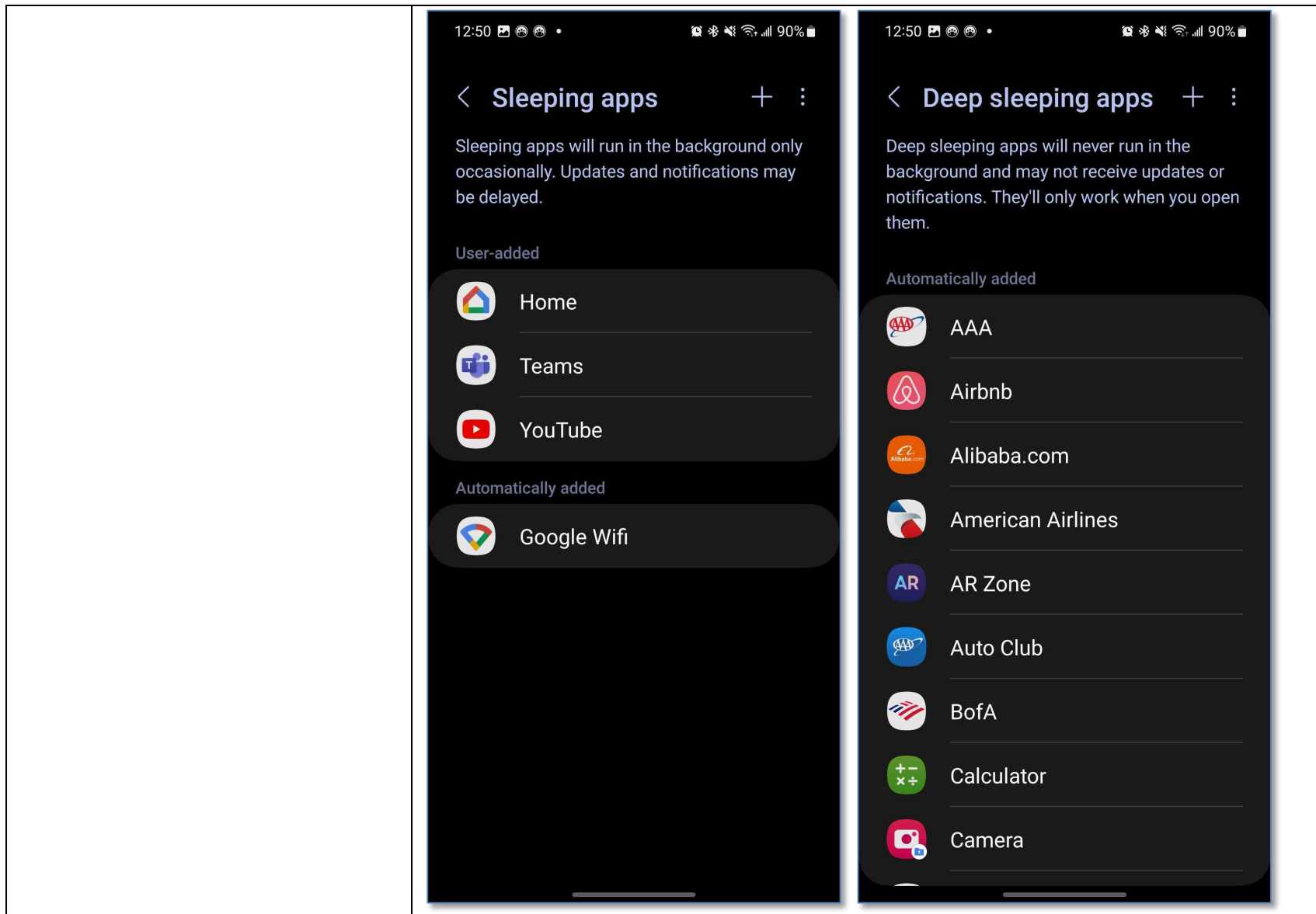
Monitor connectivity status and connection metering

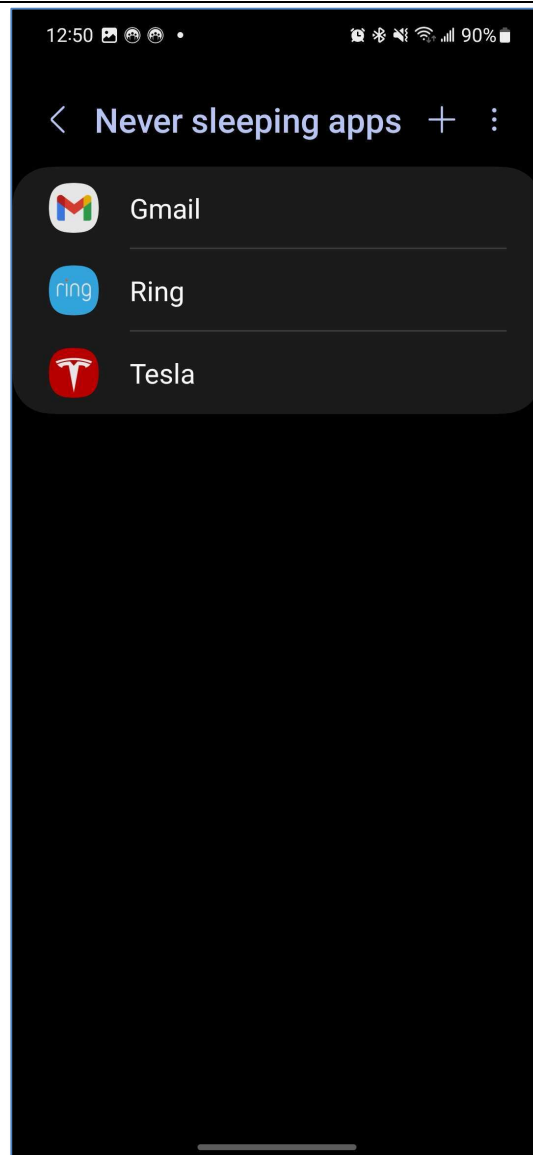
The `ConnectivityManager` provides an API that enables you to request that the device connect to a network based on various conditions that include device capabilities and data transport options.

The callback implementation provides information to your app about the device's connection status as well as the capabilities of the currently connected network. The API enables you to determine whether the device is currently connected to a network that satisfies your app's requirements.

<https://developer.android.com/training/monitoring-device-state/connectivity-status-type>;







[1c] set an application state indicating whether the first device application, associated with a particular network service usage activity, is in the foreground of user interaction, and

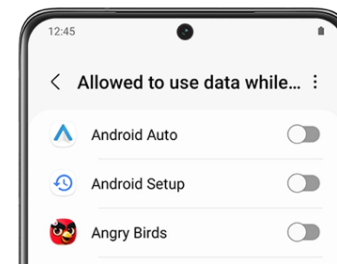
The Accused Instrumentalities comprise a processor which sets “an application state indicating whether the first device application, associated with a particular network service usage activity, is in the foreground of user interaction.”

For example, Samsung Galaxy phones and tablets utilize a “data saver” mode through which the device monitors and sets application states of applications indicating whether that application is in the background or foreground of user interaction, which in turn affects the network service usage policy applied by the device to that application.

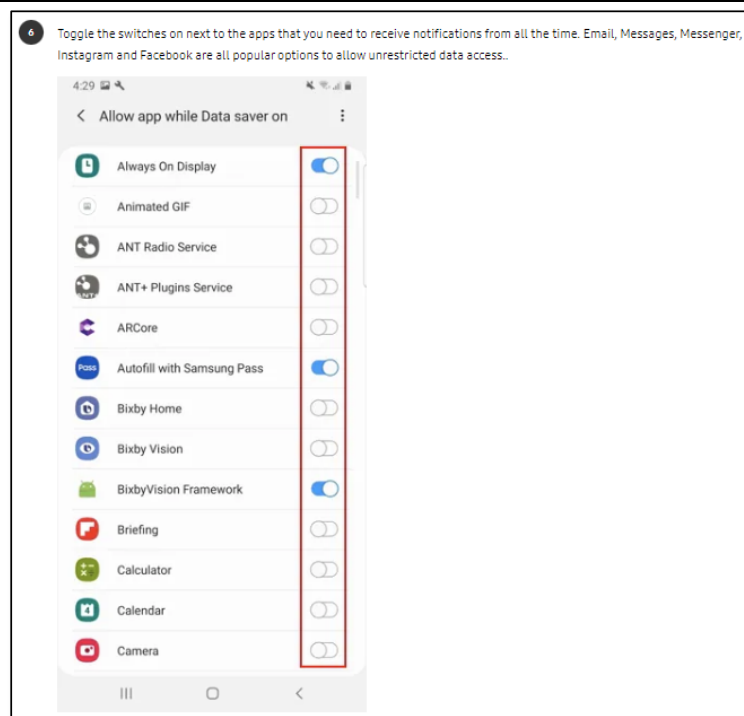
Turn Data saver on or off

Data saver prevents some apps from sending or receiving data in the background. So rest assured, you're not wasting any precious data.

1. Navigate to and open **Settings**, and then tap **Connections**.
2. Tap **Data usage**, tap **Data saver**, and then tap the **switch** next to Turn on now.
3. If there are still some apps you'd like to run in the background, you can set them as exceptions. Tap **Allowed to use data while Data saver is on** at the bottom of the screen.
4. Tap **More options** (the three vertical dots) and choose **Show system apps** or **Show allowed apps first** to narrow down the list.
5. Finally, tap the **switch(es)** next to your desired app(s).



[https://www.samsung.com/us/support/answer/ANS00079018/;](https://www.samsung.com/us/support/answer/ANS00079018/)



<https://www.samsung.com/ae/support/mobile-devices/android-pie-what-is-the-data-saver-feature/>.

For further example, the Android operating system running on Samsung Galaxy phones and tablets sets application states of applications indicating whether that application is in the background or foreground of user interaction, which in turn affects the network service usage policy applied by the device to that application as shown in the exemplary citations below.

1. A **foreground process** is one that is required for what the user is currently doing. Various application components can cause its containing process to be considered foreground in different ways. A process is considered to be in the foreground if any of the following conditions hold:

- It is running an **Activity** at the top of the screen that the user is interacting with (its **onResume()** method has been called).
- It has a **BroadcastReceiver** that is currently running (its **BroadcastReceiver.onReceive()** method is executing).
- It has a **Service** that is currently executing code in one of its callbacks (**Service.onCreate()**, **Service.onStart()**, or **Service.onDestroy()**).

There will only ever be a few such processes in the system, and these will only be killed as a last resort if memory is so low that not even these processes can continue to run. Generally, at this point, the device has reached a memory paging state, so this action is required in order to keep the user interface responsive.

<https://developer.android.com/guide/components/activities/process-lifecycle>;

Definition of background work

An app is running in the *background* when both the following conditions are satisfied:

- None of the app's activities are currently visible to the user.
- The app isn't running any **foreground services** that started while an activity from the app was visible to the user.

Otherwise, the app is running in the *foreground*.

<https://developer.android.com/guide/background>;

Services overview

A [Service](#) is an [application component](#) that can perform long-running operations in the background. It does not provide a user interface. Once started, a service might continue running for some time, even after the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC). For example, a service can handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

Types of Services

These are the three different types of services:

Foreground

A foreground service performs some operation that is noticeable to the user. For example, an audio app would use a foreground service to play an audio track. Foreground services must display a [Notification](#). Foreground services continue running even when the user isn't interacting with the app.

When you use a foreground service, you must display a notification so that users are actively aware that the service is running. This notification cannot be dismissed unless the service is either stopped or removed from the foreground.

Learn more about how to configure [foreground services](#) in your app.

★ **Note:** The [WorkManager](#) API offers a flexible way of scheduling tasks, and is able to [run these jobs as foreground services](#) if needed. In many cases, using WorkManager is preferable to using foreground services directly.

Background

A background service performs an operation that isn't directly noticed by the user. For example, if an app used a service to compact its storage, that would usually be a background service.

★ **Note:** If your app targets API level 26 or higher, the system imposes [restrictions on running background services](#) when the app itself isn't in the foreground. In most situations, for example, you shouldn't [access location information from the background](#). Instead, [schedule tasks using WorkManager](#).

	https://developer.android.com/guide/components/services;
--	---

App Standby Buckets

Android 9 (API level 28) and higher support **App Standby Buckets**. App Standby Buckets help the system prioritize apps' requests for resources based on how recently and how frequently the apps are used. Based on app usage patterns, each app is placed in one of five priority **buckets**. The system limits the device resources available to each app based on which bucket the app is in.

Priority buckets

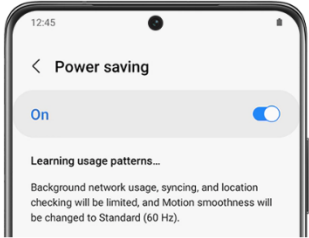
The system dynamically assigns each app to a priority bucket, reassigning the apps as needed. The system may rely on a preloaded app that uses machine learning to determine how likely each app is to be used, and assigns apps to the appropriate buckets. If the system app is not present on a device, the system defaults to sorting apps based on how recently they were used. More active apps are assigned to buckets that give the apps higher priority, making more system resources available to the app. In particular, the bucket determines how frequently the app's jobs run, and how often the app can trigger alarms. These restrictions apply only while the device is on battery power; the system does not impose these restrictions on apps while the device is charging.

★ **Note:** Every manufacturer can set their own criteria for how non-active apps are assigned to buckets. You should not try to influence which bucket your app is assigned to. Instead, focus on making sure your app behaves well in whatever bucket it might be in. Your app can find out what bucket it's currently in by calling [UsageStatsManager.getAppStandbyBucket\(\)](#).

The buckets are:

1. **Active:** App is currently being used or was very recently used.
2. **Working set:** App is in regular use.
3. **Frequent:** App is often used, but not every day.
4. **Rare:** App is not frequently used.
5. **Restricted:** App consumes a great deal of system resources, or may exhibit undesirable behavior.

In addition, there's a special **never** bucket for apps that have been installed but have never been run. The system imposes severe restrictions on these apps.

	https://developer.android.com/topic/performance/appstandby .
[1d] dynamically determine whether to apply the network service usage control policy to the particular network service usage activity, based on the application state and based on a power control state; and	The Accused Instrumentalities comprise a processor which “dynamically determine[s] whether to apply the network service usage control policy to the particular network service usage activity, based on the application state and based on a power control state.” As shown in the above exemplary citations, the Galaxy S22 provides a “data saver” mode which controls which application state (for example, “background” or “foreground”) applies to a given application. The Galaxy S22 determines which network service usage control policy to apply based on such application states as well as a power control state. For example, functionality including, but not limited to Android Doze or App Standby as well as Samsung’s Adaptive battery functionality apply network usage policies to usage activities, as illustrated by the below exemplary citations. As a further example, the Accused Devices are capable of aggregating network requests and transmitting them at predefined intervals, or at predefined times during “maintenance windows” as identified in the below disclosures. Power saving mode If you know you'll be away from a charger for an extended period and you want to make sure your phone will last, just turn on Power saving mode. This way, you won't have to worry about a low battery. First, navigate to Settings, and then tap Battery and device care. Tap Battery, and then tap Power saving. Tap the switch to turn it on. This will limit certain things on your device, such as background network usage and syncing.  You can use additional power saving options as well: • Turn off Always on Display: This will disable the Always on Display feature. • Limit CPU speed to 70%: Decreases the processing speed of your device. • Decrease brightness by 10%: Tap this option to dim your device's screen by 10%. • Limit apps and Home screen: This option will provide you with maximum power saving by limiting background activity, using a simplified Home screen, and limiting available apps. It will also turn off Edge panels. In order to turn these options on or off, Power saving must be disabled. Once you've made your desired adjustments, you can turn Power saving back on. You can also save battery power by putting unused apps to sleep with sleeping apps settings. Note: Devices with One UI 2.5 or lower will have different power saving options, such as Optimized, Medium power saving, and Maximum power saving. To find these options, open Settings, tap Device care, tap Battery, and then tap Power mode.

<https://www.samsung.com/us/support/answer/ANS00079037/#:~:text=First%2C%20navigate%20to%20Settings%2C%20and,background%20network%20usage%20and%20syncing>.

For further example, the Android operating system running on Samsung Galaxy phones and tablets determines which network service usage control policy to apply based on application states as well as a power control state as shown in the exemplary citations below. As a further example, functionality including, but not limited to Android Doze or App Standby as well as Samsung's Adaptive battery are responsible for practicing these claims.

Services overview



A **Service** is an **application component** that can perform long-running operations in the background. It does not provide a user interface. Once started, a service might continue running for some time, even after the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC). For example, a service can handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

Types of Services

These are the three different types of services:

Foreground

A foreground service performs some operation that is noticeable to the user. For example, an audio app would use a foreground service to play an audio track. Foreground services must display a [Notification](#). Foreground services continue running even when the user isn't interacting with the app.

When you use a foreground service, you must display a notification so that users are actively aware that the service is running. This notification cannot be dismissed unless the service is either stopped or removed from the foreground.

Learn more about how to configure [foreground services](#) in your app.

★ **Note:** The [WorkManager](#) API offers a flexible way of scheduling tasks, and is able to [run these jobs as foreground services](#) if needed. In many cases, using WorkManager is preferable to using foreground services directly.

Background

A background service performs an operation that isn't directly noticed by the user. For example, if an app used a service to compact its storage, that would usually be a background service.

★ **Note:** If your app targets API level 26 or higher, the system imposes [restrictions on running background services](#) when the app itself isn't in the foreground. In most situations, for example, you shouldn't [access location information from the background](#). Instead, [schedule tasks using WorkManager](#).

<https://developer.android.com/guide/components/services>;

1. A **foreground process** is one that is required for what the user is currently doing. Various application components can cause its containing process to be considered foreground in different ways. A process is considered to be in the foreground if any of the following conditions hold:

- It is running an **Activity** at the top of the screen that the user is interacting with (its **onResume()** method has been called).
- It has a **BroadcastReceiver** that is currently running (its **BroadcastReceiver.onReceive()** method is executing).
- It has a **Service** that is currently executing code in one of its callbacks (**Service.onCreate()**, **Service.onStart()**, or **Service.onDestroy()**).

There will only ever be a few such processes in the system, and these will only be killed as a last resort if memory is so low that not even these processes can continue to run. Generally, at this point, the device has reached a memory paging state, so this action is required in order to keep the user interface responsive.

<https://developer.android.com/guide/components/activities/process-lifecycle>;

Definition of background work

An app is running in the *background* when both the following conditions are satisfied:

- None of the app's activities are currently visible to the user.
- The app isn't running any **foreground services** that started while an activity from the app was visible to the user.

Otherwise, the app is running in the *foreground*.

<https://developer.android.com/guide/background>;

Optimize for Doze and App Standby

Starting from Android 6.0 (API level 23), Android introduces two power-saving features that extend battery life for users by managing how apps behave when a device is not connected to a power source. *Doze* reduces battery consumption by deferring background CPU and network activity for apps when the device is unused for long periods of time. *App Standby* defers background network activity for apps with which the user has not recently interacted.

While the device is in Doze, apps' access to certain battery-intensive resources is deferred until maintenance windows. The specific restrictions are listed in [Power Management Restrictions](#).

Doze and App Standby manage the behavior of all apps running on Android 6.0 or higher, regardless whether they are specifically targeting API level 23. To ensure the best experience for users, test your app in Doze and App Standby modes and make any necessary adjustments to your code. The sections below provide details.

Understanding Doze

If a user leaves a device unplugged and stationary for a period of time, with the screen off, the device enters Doze mode. In Doze mode, the system attempts to conserve battery by restricting apps' access to network and CPU-intensive services. It also prevents apps from accessing the network and defers their jobs, syncs, and standard alarms.

Periodically, the system exits Doze for a brief time to let apps complete their deferred activities. During this *maintenance window*, the system runs all pending syncs, jobs, and alarms, and lets apps access the network.

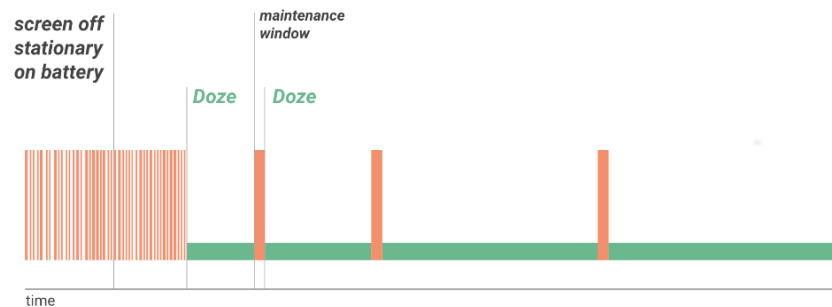


Figure 1. Doze provides a recurring maintenance window for apps to use the network and handle pending activities.

At the conclusion of each maintenance window, the system again enters Doze, suspending network access and deferring jobs, syncs, and alarms. Over time, the system schedules maintenance windows less and less frequently, helping to reduce battery consumption in cases of longer-term inactivity when the device is not connected to a charger.

As soon as the user wakes the device by moving it, turning on the screen, or connecting a charger, the system exits Doze and all apps return to normal activity.

The Doze restriction on network access is also likely to affect your app, especially if the app relies on real-time messages such as tickles or notifications. If your app requires a persistent connection to the network to receive messages, you should use [Firebase Cloud Messaging \(FCM\)](#) if possible.

Understanding App Standby

App Standby allows the system to determine that an app is idle when the user is not actively using it. The system makes this determination when the user does not touch the app for a certain period of time and none of the following conditions applies:

- The user explicitly launches the app.
- The app has a process currently in the foreground (either as an activity or foreground service, or in use by another activity or foreground service).

★ **Note:** You should only use a [foreground service](#) for tasks the user expects the system to execute immediately or without interruption. Such cases include uploading a photo to social media, or playing music even while the music-player app is not in the foreground. You should not start a foreground service simply to prevent the system from determining that your app is idle.

- The app generates a notification that users see on the lock screen or in the notification tray.
- The app is an active device admin app (for example, a [device policy controller](#)). Although they generally run in the background, device admin apps never enter App Standby because they must remain available to receive policy from a server at any time.

When the user plugs the device into a power supply, the system releases apps from the standby state, allowing them to freely access the network and to execute any pending jobs and syncs. If the device is idle for long periods of time, the system allows idle apps network access around once a day.

<https://developer.android.com/training/monitoring-device-state/doze-standby>.

App Standby Buckets

Android 9 (API level 28) and higher support **App Standby Buckets**. App Standby Buckets help the system prioritize apps' requests for resources based on how recently and how frequently the apps are used. Based on app usage patterns, each app is placed in one of five priority **buckets**. The system limits the device resources available to each app based on which bucket the app is in.

Priority buckets

The system dynamically assigns each app to a priority bucket, reassigning the apps as needed. The system may rely on a preloaded app that uses machine learning to determine how likely each app is to be used, and assigns apps to the appropriate buckets. If the system app is not present on a device, the system defaults to sorting apps based on how recently they were used. More active apps are assigned to buckets that give the apps higher priority, making more system resources available to the app. In particular, the bucket determines how frequently the app's jobs run, and how often the app can trigger alarms. These restrictions apply only while the device is on battery power; the system does not impose these restrictions on apps while the device is charging.

★ **Note:** Every manufacturer can set their own criteria for how non-active apps are assigned to buckets. You should not try to influence which bucket your app is assigned to. Instead, focus on making sure your app behaves well in whatever bucket it might be in. Your app can find out what bucket it's currently in by calling [UsageStatsManager.getAppStandbyBucket\(\)](#).

The buckets are:

1. **Active:** App is currently being used or was very recently used.
2. **Working set:** App is in regular use.
3. **Frequent:** App is often used, but not every day.
4. **Rare:** App is not frequently used.
5. **Restricted:** App consumes a great deal of system resources, or may exhibit undesirable behavior.

In addition, there's a special **never** bucket for apps that have been installed but have never been run. The system imposes severe restrictions on these apps.

<https://developer.android.com/topic/performance/appstandby;>

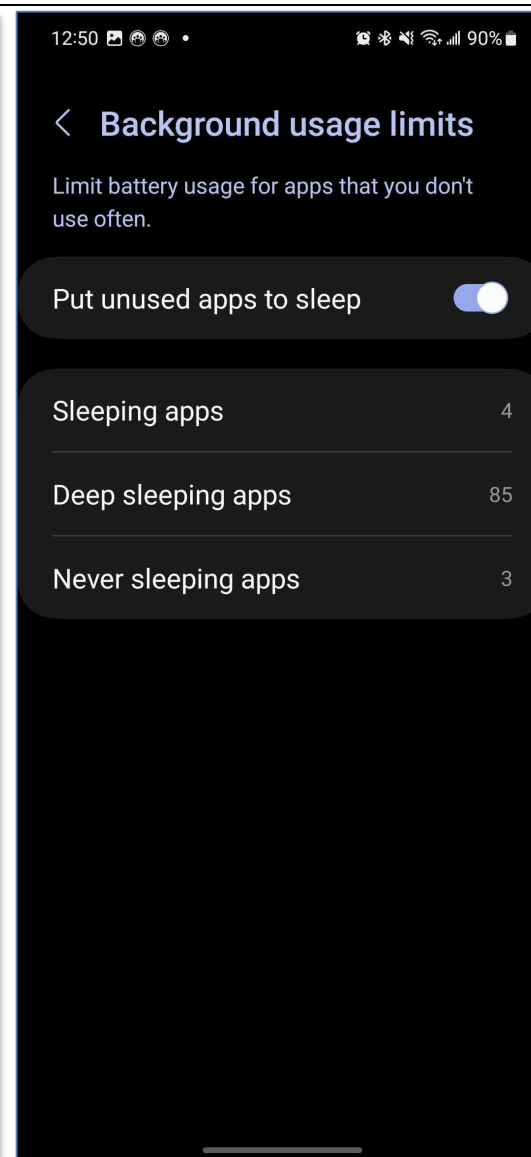
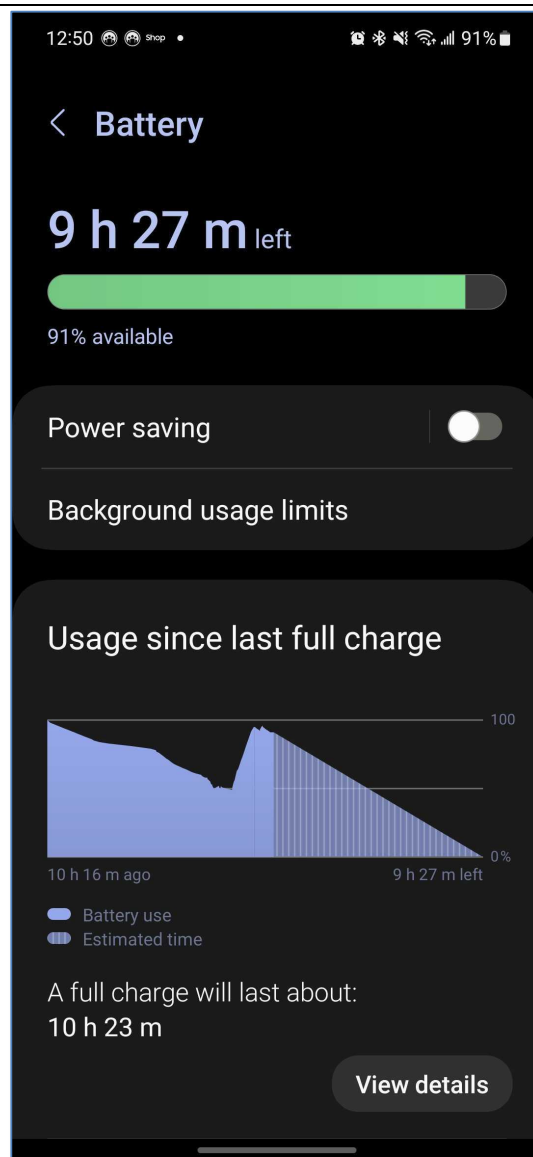
Power Management

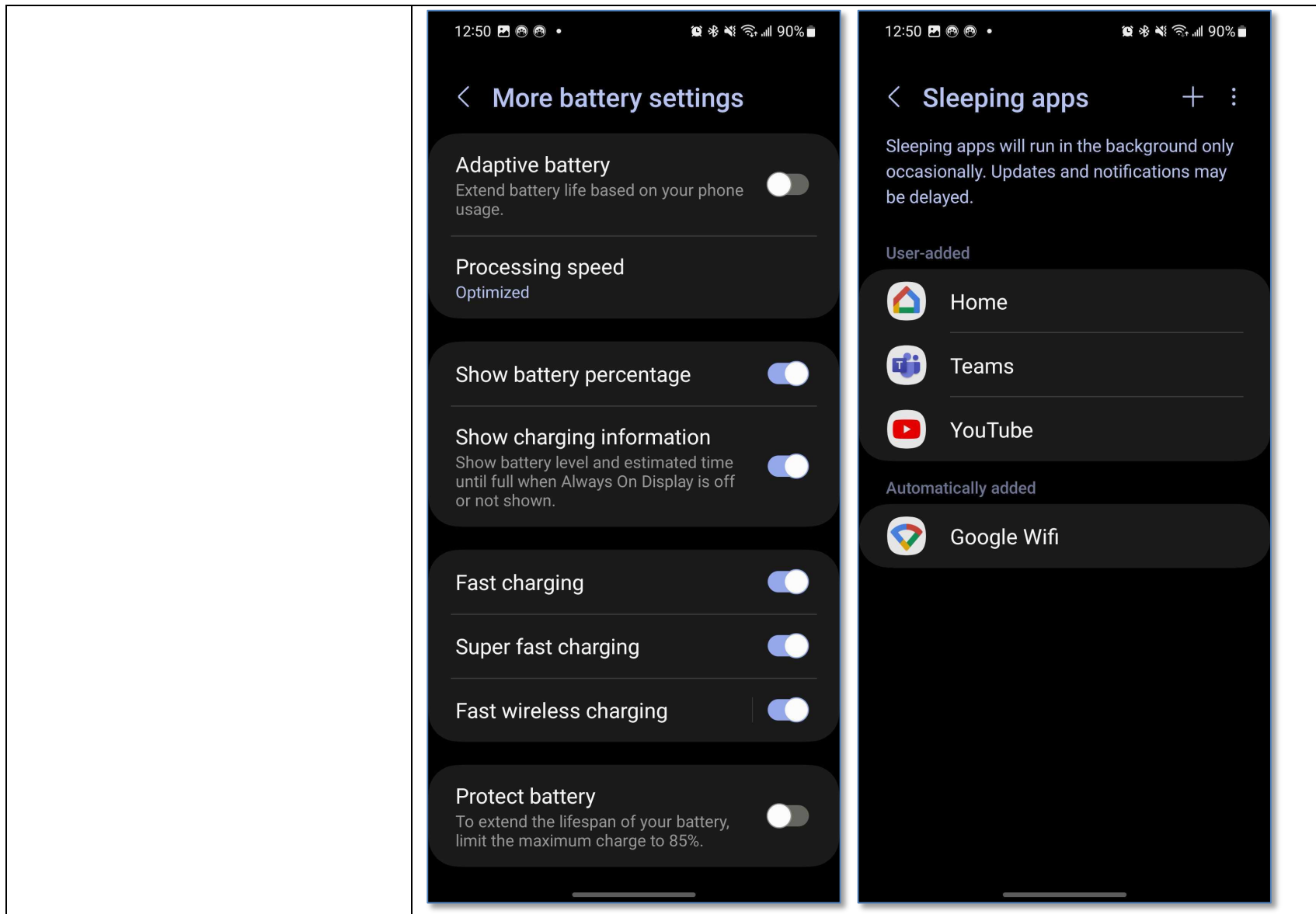
Battery life is a perennial user concern. To extend battery life, Android continually adds new features to help the platform optimize the off-charger behavior of apps and devices.

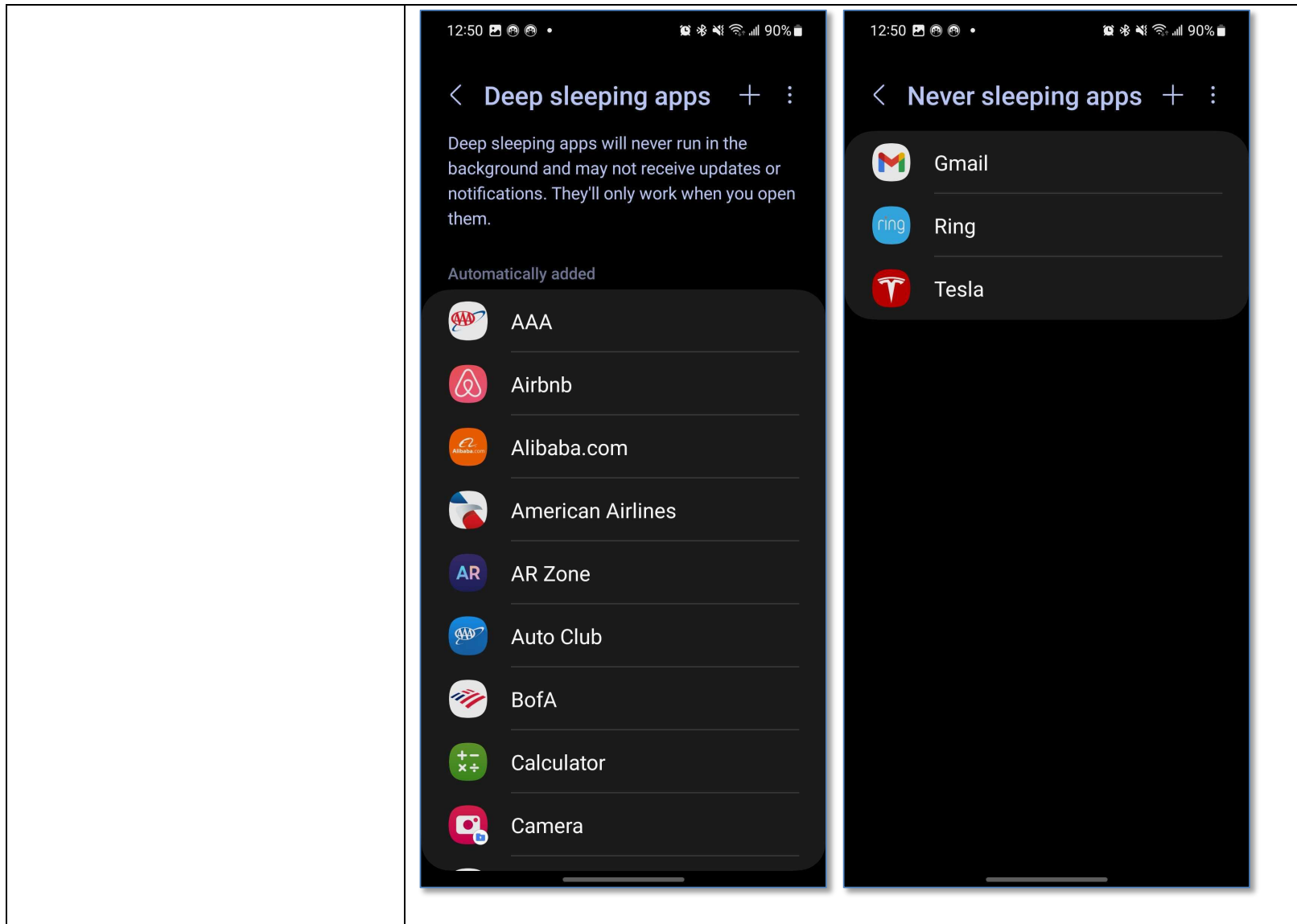
Android includes the following battery life enhancements:

- [App restrictions](#). The platform can suggest apps that negatively affect battery life, so that users can choose to restrict those apps from consuming resources. Apps aren't background restricted by default.
- [App standby](#). The platform can place unused apps in App standby mode, temporarily restricting network access and deferring syncs and jobs for those apps.
- [Doze](#). The platform can enter a state of deep sleep (periodically resuming normal operations) if users haven't actively used their device (screen off and stationary) for extended periods of time. Android 7.0 and higher also enables Doze to trigger a lighter set of optimizations when users turn off the device screen yet continue to move around.
- Limiting power transfer during USB backup and restore. When performing a USB backup or restore, set the `UsbPort` class's `enableLimitPowerTransfer` method to `true` to disable charging and limit the sourcing power. To verify that power is limited, call `UsbPortStatus` class's `isPowerTransferLimited` method. When `isPowerTransferLimited` is `true`, the Android device reduces the power draw as a sink to 0 (or the lowest possible value allowed by hardware) and limits the source current to 0. Limiting source power can be enforced only when both port partners support USB power delivery (PD) specification. To return power usage to the default state, set `enableLimitPowerTransfer` to `false`. The default state is also returned when the USB cable is disconnected.
- [Exemptions](#). Preloaded system apps and cloud messaging services are typically exempted from App standby and Doze by default. App developers can use intents to apply these settings to their apps. Users can exempt apps from App standby and Doze power-saving modes in the Settings menu.
- [Trackers](#). App background behavior trackers monitor apps' foreground and background battery usage to determine if the apps violate some policy.

<https://source.android.com/docs/core/power/mgmt;>







As yet another example, applications running on the Galaxy S22 use PowerManager API, which determines whether to apply network usage policies to network usage activities. As a further example, classes within the Android API responsible for practicing these claims are, including but not limited to: `isIgnoringBatteryOptimizations`. *See, e.g.,*

PowerManager



Added in API level 1

[Kotlin](#) | [Java](#)

```
public final class PowerManager  
extends Object
```

[java.lang.Object](#)

↳ [android.os.PowerManager](#)

This class lets you query and request control of aspects of the device's power state.

isIgnoringBatteryOptimizations

Added in API level 23

```
public boolean isIgnoringBatteryOptimizations (String packageName)
```



Return whether the given application package name is on the device's power allowlist. Apps can be placed on the allowlist through the settings UI invoked by

`Settings.ACTION_IGNORE_BATTERY_OPTIMIZATION_SETTINGS`.

Being on the power allowlist means that the system will not apply most power saving features to the app. Guardrails for extreme cases may still be applied.

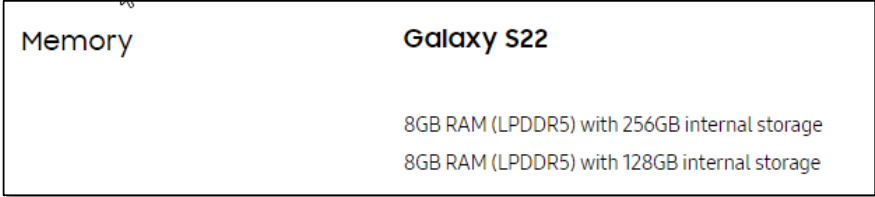
Parameters

packageName	String
-------------	--------

Returns

boolean

<https://developer.android.com/reference/android/os/PowerManager>;
[https://developer.android.com/reference/android/os/PowerManager#isIgnoringBatteryOptimizations\(java.lang.String\)](https://developer.android.com/reference/android/os/PowerManager#isIgnoringBatteryOptimizations(java.lang.String)).

[1e] a memory coupled to the processor to provide the processor with the instructions.	The Accused Instrumentalities comprise a “a memory coupled to the processor to provide the processor with the instructions.” For example, the Galaxy S22 has both RAM and internal storage:  The diagram shows a box labeled 'Memory' and 'Galaxy S22'. Inside the box, there are two lines of text: '8GB RAM (LPDDR5) with 256GB internal storage' and '8GB RAM (LPDDR5) with 128GB internal storage'.
---	---

Claim 2

2. The wireless end-user device of claim 1, wherein the processor executing the instructions determines that the first device application is in the foreground of user interaction when the user of the device is directly interacting with that application or perceiving any benefit from that application.	The Accused Instrumentalities comprise the “wireless end-user device of claim 1, wherein the processor executing the instructions determines that the first device application is in the foreground of user interaction when the user of the device is directly interacting with that application or perceiving any benefit from that application.” See, for example, the disclosures identified for claims [1b]-[1d].
--	--

Claim 3

3. The wireless end-user device of claim 1, wherein the processor executing the instructions determines that the first device application is in the foreground of user interaction based on a state of user interface priority for the application.	The Accused Instrumentalities comprise the “wireless end-user device of claim 1, wherein the processor executing the instructions determines that the first device application is in the foreground of user interaction based on a state of user interface priority for the application.” See, for example, the disclosures identified for claims [1b]-[1d].
--	--

Claim 4

4. The wireless end-user device of claim 1, wherein when the network service usage control policy is applied it disallows the particular network service usage activity, and wherein the processor executes the instruction further to, when it is determined that the particular network access request is disallowed, queue that particular network service usage activity until a power state change occurs.	The Accused Instrumentalities comprise the “wireless end-user device of claim 1, wherein when the network service usage control policy is applied it disallows the particular network service usage activity, and wherein the processor executes the instruction further to, when it is determined that the particular network access request is disallowed, queue that particular network service usage activity until a power state change occurs.” See, for example, the disclosures identified for claims [1b]-[1d].
--	--

Claim 5

5. The wireless end-user device recited in claim 1, wherein the processor executes the instructions further to: implement the network service usage control policy at least in part via communication with the first device application through an emulated network access API that indicates to the first device application that an available wireless network is unavailable when a dynamic determination applies the network service usage control policy to the particular network service usage activity.	The Accused Instrumentalities comprise the “wireless end-user device recited in claim 1, wherein the processor executes the instructions further to: implement the network service usage control policy at least in part via communication with the first device application through an emulated network access API that indicates to the first device application that an available wireless network is unavailable when a dynamic determination applies the network service usage control policy to the particular network service usage activity.” See, for example, the disclosures identified for claims [1b]-[1d]. As another example, the Android application programming interface (API), indicates to device applications that an available wireless network is unavailable based on a determination that a network service usage control policy applies to a network service usage activity. As a further example, classes within the Android API responsible for practicing these claims include, but are not limited to: <code>RESTRICT_BACKGROUND_STATUS_DISABLED</code>, <code>RESTRICT_BACKGROUND_STATUS_ENABLED</code>, <code>RESTRICT_BACKGROUND_STATUS_WHITELISTED</code>. <i>See, e.g.,</i>
--	---

ConnectivityManager

Added in API level 1

[Kotlin](#) | **Java**

```
public class ConnectivityManager  
    extends Object
```

[java.lang.Object](#)
↳ [android.net.ConnectivityManager](#)

Class that answers queries about the state of network connectivity. It also notifies applications when network connectivity changes.

The primary responsibilities of this class are to:

1. Monitor network connections (Wi-Fi, GPRS, UMTS, etc.)
2. Send broadcast intents when network connectivity changes
3. Attempt to "fail over" to another network when connectivity to a network is lost
4. Provide an API that allows applications to query the coarse-grained or fine-grained state of the available networks
5. Provide an API that allows applications to request and select networks for their data traffic

RESTRICT_BACKGROUND_STATUS_DISABLED

Added in API level 24

```
public static final int RESTRICT_BACKGROUND_STATUS_DISABLED
```



Device is not restricting metered network activity while application is running on background.

Constant Value: 1 (0x00000001)

RESTRICT_BACKGROUND_STATUS_ENABLED

Added in API level 24

```
public static final int RESTRICT_BACKGROUND_STATUS_ENABLED
```



Device is restricting metered network activity while application is running on background.

In this state, application should not try to use the network while running on background, because it would be denied.

Constant Value: 3 (0x00000003)

RESTRICT_BACKGROUND_STATUS_WHITELISTED

Added in API level 24

```
public static final int RESTRICT_BACKGROUND_STATUS_WHITELISTED
```



Device is restricting metered network activity while application is running on background, but application is allowed to bypass it.

In this state, application should take action to mitigate metered network access. For example, a music streaming application should switch to a low-bandwidth bitrate.

Constant Value: 2 (0x00000002)

	https://developer.android.com/reference/android/net/ConnectivityManager .
--	---

Claim 6

6. The wireless end-user device of claim 1, wherein the processor further executes the instruction to, for at least a second device application or service, allow network service usage activities without regard to the power control state.	The Accused Instrumentalities comprise the “wireless end-user device of claim 1, wherein the processor further executes the instruction to, for at least a second device application or service, allow network service usage activities without regard to the power control state.” See, for example, the disclosures identified for claims [1b]-[1d].
---	--

Claim 7

7. The wireless end-user device of claim 1, wherein the power control state is a power state of the device.	The Accused Instrumentalities comprise the “wireless end-user device of claim 1, wherein the power control state is a power state of the device.” See, for example, the disclosures identified for claims [1b]-[1d].
---	--

Claim 8

8. The wireless end-user device recited in claim 1, wherein the processor executes the instructions further to: execute a router for dynamically managing one or more network capacity controlled services and/or QoS sessions for the wireless end-user device.	The Accused Instrumentalities comprise the “wireless end-user device recited in claim 1, wherein the processor executes the instructions further to: execute a router for dynamically managing one or more network capacity controlled services and/or QoS sessions for the wireless end-user device.” See, for example, the disclosures identified for claims [1b]-[1d].
--	---

Claim 11

11. The wireless end-user device recited in claim 1, wherein dynamically determining whether to apply the network service usage control policy to the particular network service usage activity is further based on a current wireless network and/or the network service usage control policy is based on a current wireless network.

The Accused Instrumentalities further comprise the “wireless end-user device recited in claim 1, wherein dynamically determining whether to apply the network service usage control policy to the particular network service usage activity is further based on a current wireless network and/or the network service usage control policy is based on a current wireless network.”

See, for example, the disclosures identified for claims [1b]-[1d] as well as the following disclosures.

ConnectivityManager

Added in API level 1

Kotlin | Java

```
public class ConnectivityManager
    extends Object
```

[java.lang.Object](#)
↳ [android.net.ConnectivityManager](#)

Class that answers queries about the state of network connectivity. It also notifies applications when network connectivity changes.

The primary responsibilities of this class are to:

1. Monitor network connections (Wi-Fi, GPRS, UMTS, etc.)
2. Send broadcast intents when network connectivity changes
3. Attempt to "fail over" to another network when connectivity to a network is lost
4. Provide an API that allows applications to query the coarse-grained or fine-grained state of the available networks
5. Provide an API that allows applications to request and select networks for their data traffic

<https://developer.android.com/reference/android/net/ConnectivityManager>;

Monitor connectivity status and connection metering

The `ConnectivityManager` provides an API that enables you to request that the device connect to a network based on various conditions that include device capabilities and data transport options.

The callback implementation provides information to your app about the device's connection status as well as the capabilities of the currently connected network. The API enables you to determine whether the device is currently connected to a network that satisfies your app's requirements.

<https://developer.android.com/training/monitoring-device-state/connectivity-status-type>:

If the system places your app in the restricted bucket, the following restrictions apply:

- You can [run jobs](#) once per day, in a 10-minute batched session. During this session, the system groups your app's jobs with other apps' jobs.
 - Restricted jobs are not run by themselves – there must be at least one other job running/pending at the same time, which can include any other job.
- Your app can run fewer [expedited jobs](#), compared to when the system places your app in a less-restrictive bucket.
- Your app can invoke one alarm per day. This alarm can be either an [exact alarm](#) or an [inexact alarm](#).

★ **Note:** Unlike with other buckets, these power management restrictions apply to the restricted bucket even when the device is charging. However, restrictions are loosened when the device is charging, idle, and on an unmetered network.

<https://developer.android.com/topic/performance/appstandby>:

	Schedule work on unmetered connections When building a <code>WorkRequest</code>, add a <code>NetworkType.UNMETERED</code> <code>Constraint</code>. <pre> fun scheduleWork(context: Context) { val workManager = WorkManager.getInstance(context) val workRequest = OneTimeWorkRequestBuilder<MyWorker>() .setConstraints(Constraints.Builder() .setRequiredNetworkType(NetworkType.UNMETERED) .build()) .build() workManager.enqueue(workRequest) } </pre> When the conditions for your work are met, your app receives a callback to run the <code>doWork()</code> method in the specified <code>Worker</code> class. https://developer.android.com/guide/background/optimizations.
--	---

Claim 12

12. The wireless end-user device recited in claim 1, wherein dynamically determining whether to apply the network service usage control policy to the particular network service usage activity further includes dynamically assigning a network capacity controlled services priority level to the particular network service usage activity based on a network busy state.	The Accused Instrumentalities “wireless end-user device recited in claim 1, wherein dynamically determining whether to apply the network service usage control policy to the particular network service usage activity further includes dynamically assigning a network capacity controlled services priority level to the particular network service usage activity based on a network busy state.” For example, the <code>ConnectivityManager</code> API and the <code>ConnectivityManager.NetworkCallback</code> API dynamically assigning a network capacity controlled services priority level to the particular network service usage activity based on a network busy state, such as a network congestion state.
---	---

	NET_CAPABILITY_NOT_CONGESTED Added in API level 28 <pre>public static final int NET_CAPABILITY_NOT_CONGESTED</pre> Indicates that this network is not congested. When a network is congested, applications should defer network traffic that can be done at a later time, such as uploading analytics. Constant Value: 20 (0x00000014) https://developer.android.com/reference/android/net/NetworkCapabilities.
--	--

Claim 13

13. The wireless end-user device recited in claim 1, wherein associating includes querying a network element for determining an association of the first device application with the network service usage control policy.	The Accused Instrumentalities comprise the “wireless end-user device recited in claim 1, wherein associating includes querying a network element for determining an association of the first device application with the network service usage control policy.” See, for example, the disclosures identified for claim [1b]-[1d] as well as the following disclosures.
---	--

UICC Carrier Privileges

Android 5.1 introduced a mechanism to grant special privileges for APIs relevant to the owners of universal integrated circuit card (UICC) apps. The Android platform loads certificates stored on a UICC and grants permission to apps signed by these certificates to make calls to a handful of special APIs.

Android 7.0 extended this feature to support other storage sources for UICC carrier privilege rules, dramatically increasing the number of carriers that can use the APIs. For an API reference, see [CarrierConfigManager](#); for instructions, see [Carrier Configuration](#).

Carriers have full control of the UICC, so this mechanism provides a secure and flexible way to manage apps from the mobile network operator (MNO) hosted on generic app distribution channels (such as Google Play) while retaining special privileges on devices and without the need to sign apps with the per-device platform certificate or preinstall as a system app.

CarrierService

Service that exposes carrier-specific functionalities to the system. To extend this class, declare the service in the app manifest file with the `android.Manifest.permission#BIND_CARRIER_SERVICES` permission and include an intent filter with the `CARRIER_SERVICE_INTERFACE` action. If the service has a long-lived binding, set `android.service.carrier.LONG_LIVED_BINDING` to `true` in the metadata of the service.

The platform binds `CarrierService` with special flags to let the carrier service process run in a special [app standby bucket](#). This exempts the carrier service app from [app idle restriction](#) and makes it more likely to stay alive when device memory is low. However, if the carrier service app crashes for any reason, it loses all above privileges until the app restarts and the binding is reestablished. So it's critical to keep the carrier service app stable.

Methods in `CarrierService` include:

- To override and set the carrier specific configurations: [onLoadConfig](#)
- To inform the system of an intentional upcoming carrier network change by the carrier application: [notifyCarrierNetworkChange](#)

<https://source.android.com/docs/core/connect/uicc>.

Claim 14

14. The wireless end-user device recited in claim 1, wherein the network service usage control policy includes one or more of the following: block/allow settings, throttle settings, adaptive throttle settings, QoS class settings, packet error rate, jitter and delay settings, queue settings, and tag settings.	The Accused Instrumentalities comprise the “wireless end-user device recited in claim 1, wherein the network service usage control policy includes one or more of the following: block/allow settings, throttle settings, adaptive throttle settings, QoS class settings, packet error rate, jitter and delay settings, queue settings, and tag settings.” See, for example, the disclosures identified for claim [1b]-[1d].
---	--

Claim 15

15. The wireless end-user device recited in claim 1, wherein the network service usage control policy includes traffic control policy filters, the filters comprising a first filter specifying applicability of the policy only when the first device application is not in the foreground of user interaction, and a second filter specifying applicability of the policy only in one or more specific power control states.	The Accused Instrumentalities comprise the “wireless end-user device recited in claim 1, wherein the network service usage control policy includes traffic control policy filters, the filters comprising a first filter specifying applicability of the policy only when the first device application is not in the foreground of user interaction, and a second filter specifying applicability of the policy only in one or more specific power control states.” See, for example, the disclosures identified for claim [1b]-[1d].
--	---

Claim 16

16. The wireless end-user device recited in claim 1, wherein the network service usage control policy includes traffic control policy filters implemented as cascading filters.	The Accused Instrumentalities comprise the “wireless end-user device recited in claim 1, wherein the network service usage control policy includes traffic control policy filters implemented as cascading filters.” See, for example, the disclosures identified for claim [1b]-[1d].
---	--

Claim 17

17. The wireless end-user device recited in claim 1, wherein the network service usage control policy includes traffic control policy filters using a network busy state and/or a time of day as an index into a traffic control setting.	The Accused Instrumentalities comprise the “wireless end-user device recited in claim 1, wherein the network service usage control policy includes traffic control policy filters using a network busy state and/or a time of day as an index into a traffic control setting.” See, for example, the disclosures identified for claims [1b]-[1d] and 12.
---	--

Claim 18

18. The wireless end-user device recited in claim 1, wherein the processor executes the instructions further to: differentially control the particular network service usage activity based on the network service usage control policy based on a network busy state.	The Accused Instrumentalities comprise the “wireless end-user device recited in claim 1, wherein the network service usage control policy includes traffic control policy filters using a network busy state and/or a time of day as an index into a traffic control setting.” See, for example, the disclosures identified for claims [1b]-[1d] and 12.
--	--

Claim 19

19. The wireless end-user device recited in claim 1, wherein the processor executes the instructions further to: differentially control the particular network service usage activity based on the network service usage control policy based on a user input and/or a current wireless network.	The Accused Instrumentalities comprise the “wireless end-user device recited in claim 1, wherein the processor executes the instructions further to: differentially control the particular network service usage activity based on the network service usage control policy based on a user input and/or a current wireless network.” See, for example, the disclosures identified for claims [1b]-[1d] and 11.
--	---

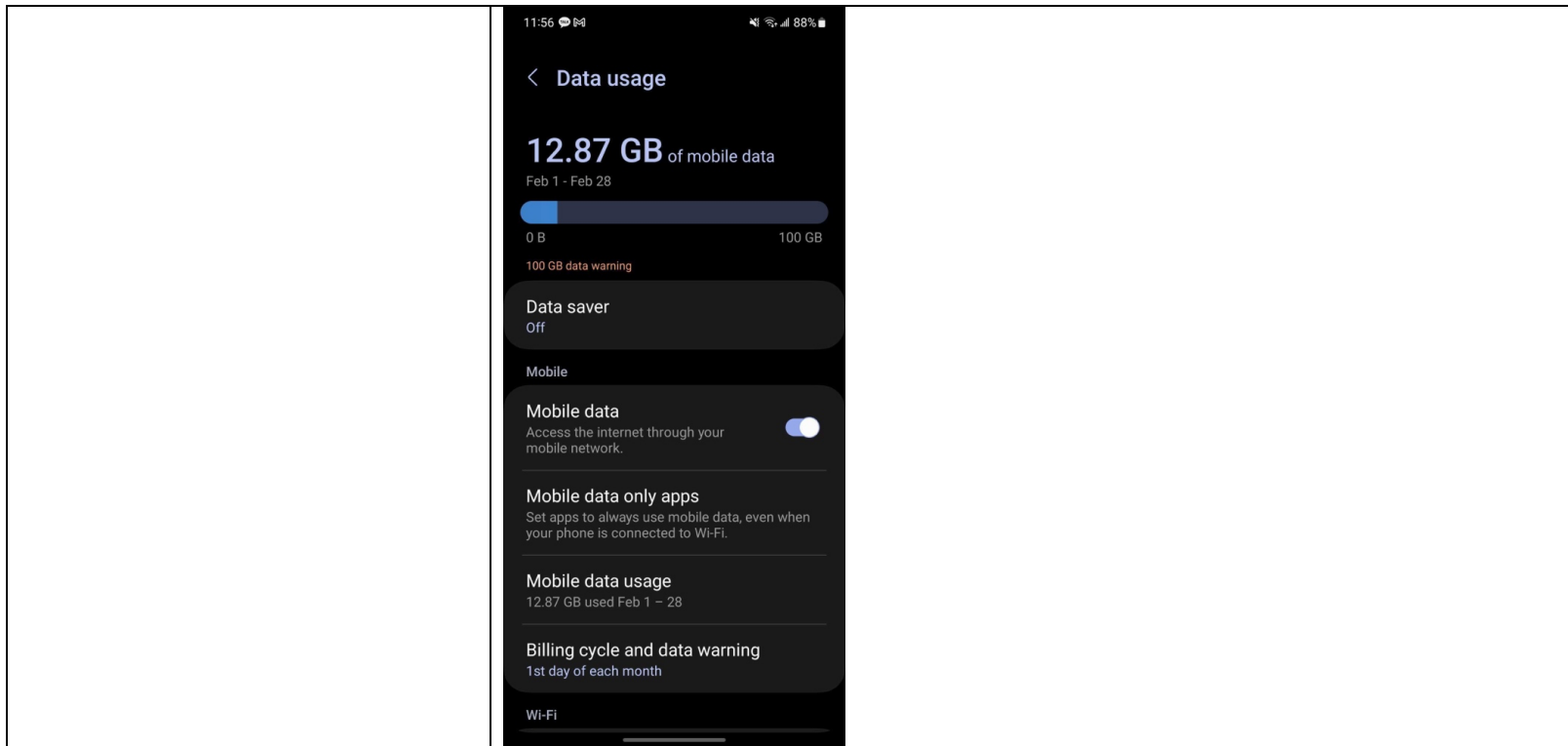
Claim 20

20. The wireless end-user device recited in claim 1, wherein the processor executes the instructions further to:	The Accused Instrumentalities comprise the “wireless end-user device recited in claim 1, wherein the processor executes the instructions further to: modify or replace a network stack interface of the wireless end-user device to provide for intercept or discontinuance of network
--	---

modify or replace a network stack interface of the wireless end-user device to provide for intercept or discontinuance of network access messaging for implementing traffic control of the particular network service usage activity.	access messaging for implementing traffic control of the particular network service usage activity.” See, for example, the disclosures identified for claim [1b]-[1d].
---	--

Claim 21

21. The wireless end-user device recited in claim 1, wherein the processor executes the instructions further to: store a network capacity controlled service list, wherein the network capacity controlled service list is periodically updated based on monitored network service usage activities.	The Accused Instrumentalities comprise the “wireless end-user device recited in claim 1, wherein the processor executes the instructions further to: store a network capacity controlled service list, wherein the network capacity controlled service list is periodically updated based on monitored network service usage activities.” See, for example, the disclosures identified for claim [1b]-[1d]. For further illustration, see the following disclosures:
--	--



Claim 22

22. The wireless end-user device of claim 1, wherein the power control state is a power save state of the device.	The Accused Instrumentalities comprise the “wireless end-user device of claim 1, wherein the power control state is a power save state of the device.” See, for example, the disclosures identified for claim [1b]-[1d].
---	--

Claim 23

23. The wireless end-user device of claim 1, wherein the power control state is a power state of the wireless modem.	The Accused Instrumentalities comprise the “wireless end-user device of claim 1, wherein the power control state is a power state of the wireless modem.” See, for example, the disclosures identified for claim [1b]-[1d] as well as the following disclosures. Optimize network access  Using the wireless radio to transfer data is potentially one of your app's most significant sources of battery drain. To minimize the battery drain associated with network activity, it's critical that you understand how your connectivity model will affect the underlying radio hardware. This section introduces the wireless radio state machine and explains how your app's connectivity model interacts with it. It then offers several techniques which, when followed, will help minimize the effect of your app's data consumption on the battery.
---	---

The radio state machine

The wireless radio on your user's device has built-in power-saving features that help minimize the amount of battery power it consumes. When fully active, the wireless radio consumes significant power, but when inactive or in standby, the radio consumes very little power.

One important factor to remember is that the radio cannot move from standby to fully active instantaneously. There is a latency period associated with "powering up" the radio. So the battery transitions from higher energy states to lower energy states slowly in order to conserve power when not in use while attempting to minimize the latency associated with "powering up" the radio.

The state machine for a typical 3G network radio consists of three energy states:

- **Full power:** Used when a connection is active, allowing the device to transfer data at its highest possible rate.
- **Low power:** An intermediate state that cuts battery power consumption by around 50%.
- **Standby:** The minimal power-consuming state during which no network connection is active.

While the low and standby states drain significantly less battery, they also introduce significant latency to network requests. Returning to full power from the low state takes around 1.5 seconds, and moving from standby to full power can take over 2 seconds.

To minimize latency, the state machine uses a delay to postpone the transition to lower energy states. Figure 1 uses AT&T's timings for a typical 3G radio.

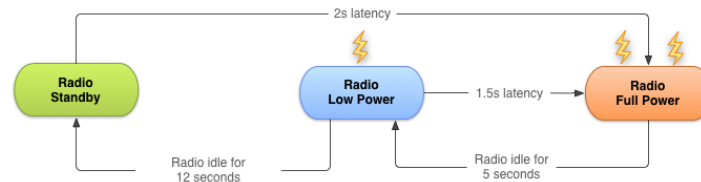


Figure 1. Typical 3G wireless radio state machine.

The radio state machine on each device, particularly the associated transition delay ("tail time") and startup latency, will vary based on the wireless radio technology employed (3G, LTE, 5G, and so on) and is defined and configured by the carrier network over which the device is operating.

This page describes a representative state machine for a typical 3G wireless radio, based on data provided by AT&T. However, the general principles and resulting best practices are applicable for all wireless radio implementations.

This approach is particularly effective for typical mobile web browsing as it prevents unwelcome latency while users browse the web. The relatively low tail-time also ensures that once a browsing session has finished, the radio can move to a lower energy state.

Unfortunately, this approach can lead to inefficient apps on modern smartphone operating systems like Android, where apps run both in the foreground (where latency is important) and in the background (where battery life should be prioritized).

How apps impact the radio state machine

Every time you create a new network connection, the radio transitions to the full power state. In the case of the typical 3G radio state machine described earlier, it will remain at full power for the duration of your transfer—plus an additional 5 seconds of tail time—followed by 12 seconds at the low energy state. So for a typical 3G device, every data transfer session will cause the radio to draw energy for at least 18 seconds.

In practice, this means that an app which makes a one second data transfer, three times a minute, will keep the wireless radio perpetually active, moving it back to high power just as it is entering standby mode.

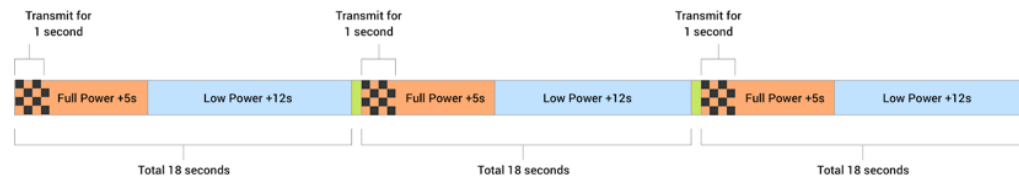


Figure 2. Relative wireless radio power use for one-second transfer running three times every minute. Figure excludes “power up” latency between runs.

By comparison, if the same app bundled its data transfers, running a single three-second transfer every minute, this would keep the radio in the high-power state for a total of only 20 seconds each minute. This would allow the radio to be on standby for 40 seconds of every minute, resulting in a significant reduction in battery consumption.



Figure 3. Relative wireless radio power use for three second transfers running once every minute.

Optimization techniques

Now that you understand how network access affects battery life, let's talk about a few things that you can do to help reduce battery drain, while also providing a fast and fluid user experience.

Bundle data transfers

As stated in the previous section, bundling your data transfers so that you're transferring more data less often is one of the best ways to improve battery efficiency.

Of course, this is not always possible to do if your app needs to receive or send data immediately in response to a user action. You can mitigate this by anticipating and [prefetching data](#). Other scenarios, such as sending logs or analytics to a server and other, non-urgent, app-initiated data transfers, lend themselves very well to batching and bundling. See [Optimizing app-initiated tasks](#) for tips on scheduling background network transfers.

Prefetch data

Prefetching data is another effective way to reduce the number of independent data transfer sessions that your app runs. With prefetching, when the user performs an action in your app, the app anticipates which data will most likely be needed for the next series of user actions and fetches that data in a single burst, over a single connection, at full capacity.

By front-loading your transfers, you reduce the number of radio activations required to download the data. As a result, you not only conserve battery life, but also improve the latency, lower the required bandwidth, and reduce download times.

Prefetching also provides an improved user experience by minimizing in-app latency caused by waiting for downloads to complete before performing an action or viewing data.

Check for connectivity before making requests

Searching for a cell signal is one of the most power-draining operations on a mobile device. A best practice for user-initiated requests is to first check for a connection using `ConnectivityManager`, as shown in [Monitor connectivity status and connection metering](#). If there's no network, the app can save battery by not forcing the mobile radio to search. The request can then be scheduled and performed in a batch with other requests when a connection is made.

Pool connections

An additional strategy that can help in addition to batching and prefetching, is to pool your app's network connections.

It's generally more efficient to reuse existing network connections than it is to initiate new ones. Reusing connections also allows the network to more-intelligently react to congestion and related network data issues.

`HttpURLConnection` and most HTTP clients, such as [OkHttp](#), enable connection-pooling by default, and reusing the same connection for multiple requests.

<https://developer.android.com/training/connectivity/network-access-optimization>.