Background Execution

When the user is not actively using your app, the system moves it to the background state. For many apps, the background state is just a brief stop on the way to the app being suspended. Suspending apps is a way of improving battery life it also allows the system to devote important system resources to the new foreground app that has drawn the user's attention.

Most apps can move to the suspended state easily enough but there are also legitimate reasons for apps to continue running in the background. A hiking app might want to track the user's position over time so that it can display that course overlaid on top of a hiking map. An audio app might need to continue playing music over the lock screen. Other apps might want to download content in the background so that it can minimize the delay in presenting that content to the user. When you find it necessary to keep your app running in the background, iOS helps you do so efficiently and without draining system resources or the user's battery. The techniques offered by iOS fall into three categories:

- Apps that start a short task in the foreground can ask for time to finish that task when the app moves to the background.
- Apps that initiate downloads in the foreground can hand off management of those downloads to the system, thereby allowing the app to be suspended or terminated while the download continues.
- Apps that need to run in the background to support specific types of tasks can declare their support for one or more background execution modes.

Always try to avoid doing any background work unless doing so improves the overall user experience. An app might move to the background because the user launched a different app or because the user locked the device and is not using it right now. In both situations, the user is signaling that your app does not need to be doing any meaningful work right now. Continuing to run in such conditions will only drain the device's battery and might lead the user to force quit your app altogether. So be mindful about the work you do in the background and avoid it when you can.

Executing Finite-Length Tasks

Apps moving to the background are expected to put themselves into a quiescent state as quickly as possible so that they can be suspended by the system. If your app is in the middle of a task and needs a little extra time to complete that task, it can call the `beginBackgroundTaskWithName:expirationHandler:` or `beginBackgroundTaskWithExpirationHandler:` method of the `UIApplication` object to request some additional execution time. Calling either of these methods delays the suspension of your app temporarily, giving it a little extra time to finish its work. Upon completion of that work, your app must call the `endBackgroundTask:` method to let the system know that it is finished and can be suspended.

Each call to the `beginBackgroundTaskWithName:expirationHandler:` or `beginBackgroundTaskWithExpirationHandler:` method generates a unique token to associate with the corresponding task. When your app completes a task, it must call the `endBackgroundTask:` method with the corresponding token to let the system know that the task is complete. Failure to call the `endBackgroundTask:` method for a background task will result in the termination of your app. If you provided an expiration handler when starting the task, the system calls that handler and gives you one last chance to end the task and avoid termination.

You do not need to wait until your app moves to the background to designate background tasks. A more useful design is to call the `beginBackgroundTaskWithName:expirationHandler:` or `beginBackgroundTaskWithExpirationHandler:` method before starting a task and call the `endBackgroundTask:` method as soon as you finish. You can even follow this pattern while your app is executing in the foreground.

Listing 3-1 shows how to start a long-running task when your app transitions to the background. In this example, the request to start a background task includes an expiration handler just in case the task takes too long. The task itself is then submitted to a dispatch queue for asynchronous execution so that the applicationDidEnterBackground: method can return normally. The use of blocks simplifies the code needed to maintain references to any important variables, such as the background task identifier. The bgTask variable is a member variable of the class that stores a pointer to the current background task identifier and is initialized prior to its use in this method.

Listing 3-1 Starting a background task at quit time

```
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    bgTask = [application beginBackgroundTaskWithName:@"MyTask" expirationHandler:^{
        // Clean up any unfinished task business by marking where you
        // stopped or ending the task outright.
        [application endBackgroundTask:bgTask];
        bgTask = UIBackgroundTaskInvalid;
    }];

    // Start the long-running task and return immediately.
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

        // Do the work associated with the task, preferably in chunks.

        [application endBackgroundTask:bgTask];
        bgTask = UIBackgroundTaskInvalid;
    });
}
```

Note: Always provide an expiration handler when starting a task, but if you want to know how much time your app has left to run, get the value of the backgroundTimeRemaining property of UIApplication.

In your own expiration handlers, you can include additional code needed to close out your task. However, any code you include must not take too long to execute because, by the time your expiration handler is called, your app is already very close to its time limit. For this reason, perform only minimal cleanup of your state information and end the task.

Downloading Content in the Background

When downloading files, apps should use an NSURLSession object to start the downloads so that the system can take control of the download process in case the app is suspended or terminated. When you configure an NSURLSession object for background transfers, the system manages those transfers in a separate process and reports status back to your app in the usual way. If your app is terminated while transfers are ongoing, the system continues the transfers in the background and launches your app (as appropriate) when the transfers finish or when one or more tasks need your app's attention.

To support background transfers, you must configure your NSURLSession object appropriately. To configure the session, you must first create a NSURLSessionConfiguration object and set several properties to appropriate values. You then pass that configuration object to the appropriate initialization method of NSURLSession when creating your session.

The process for creating a configuration object that supports background downloads is as follows:

1. Create the configuration object using the `backgroundSessionConfigurationWithIdentifier:` method of `NSURLSessionConfiguration`.
2. Set the value of the configuration object's `sessionSendsLaunchEvents` property to `YES`.
3. if your app starts transfers while it is in the foreground, it is recommend that you also set the `discretionary` property of the configuration object to `YES`.
4. Configure any other properties of the configuration object as appropriate.
5. Use the configuration object to create your `NSURLSession` object.

Once configured, your `NSURLSession` object seamlessly hands off upload and download tasks to the system at appropriate times. If tasks finish while your app is still running (either in the foreground or the background), the session object notifies its delegate in the usual way. If tasks have not yet finished and the system terminates your app, the system automatically continues managing the tasks in the background. If the user terminates your app, the system cancels any pending tasks.

When all of the tasks associated with a background session are complete, the system relaunches a terminated app (assuming that the `sessionSendsLaunchEvents` property was set to `YES` and that the user did not force quit the app) and calls the app delegate's `application:handleEventsForBackgroundURLSession:completionHandler:` method. (The system may also relaunch the app to handle authentication challenges or other task-related events that require your app's attention.) In your implementation of that delegate method, use the provided identifier to create a new `NSURLSessionConfiguration` and `NSURLSession` object with the same configuration as before. The system reconnects your new session object to the previous tasks and reports their status to the session object's delegate.

Implementing Long-Running Tasks

For tasks that require more execution time to implement, you must request specific permissions to run them in the background without their being suspended. In iOS, only specific app types are allowed to run in the background:

- Apps that play audible content to the user while in the background, such as a music player app
- Apps that record audio content while in the background
- Apps that keep users informed of their location at all times, such as a navigation app
- Apps that support Voice over Internet Protocol (VoIP)
- Apps that need to download and process new content regularly
- Apps that receive regular updates from external accessories

Apps that implement these services must declare the services they support and use system frameworks to implement the relevant aspects of those services. Declaring the services lets the system know which services you use, but in some cases it is the system frameworks that actually prevent your application from being suspended.

Declaring Your App's Supported Background Tasks

Support for some types of background execution must be declared in advance by the app that uses them. In Xcode 5 and later, you declare the background modes your app supports from the Capabilities tab of your project settings. Enabling the Background Modes option adds the `UIBackgroundModes` key to your app's `Info.plist` file. Selecting one or more checkboxes adds the corresponding background mode values to that key. Table 3-1 lists the background modes you can specify and the values that Xcode assigns to the `UIBackgroundModes` key in your app's `Info.plist` file.

Table 3-1 Background modes for apps

Xcode		

background mode	UIBackgroundModes value	Description
Audio and AirPlay	`audio`	The app plays audible content to the user or records audio while in the background. (This content includes streaming audio or video content using AirPlay.) The user must grant permission for apps to use the microphone prior to the first use; for more information, see Supporting User Privacy.
Location updates	`location`	The app keeps users informed of their location, even while it is running in the background.
Voice over IP	`voip`	The app provides the ability for the user to make phone calls using an Internet connection.
Newsstand downloads	`newsstand-content`	The app is a Newsstand app that downloads and processes magazine or newspaper content in the background.
External accessory communication	`external-accessory`	The app works with a hardware accessory that needs to deliver updates on a regular schedule through the External Accessory framework.
Uses Bluetooth LE accessories	`bluetooth-central`	The app works with a Bluetooth accessory that needs to deliver updates on a regular schedule through the Core Bluetooth framework.
Acts as a Bluetooth LE accessory	`bluetooth-peripheral`	The app supports Bluetooth communication in peripheral mode through the Core Bluetooth framework. Using this mode requires user authorization; for more information, see Supporting User Privacy.
Background fetch	`fetch`	The app regularly downloads and processes small amounts of content from the network.
Remote notifications	`remote-notification`	The app wants to start downloading content when a push notification arrives. Use this notification to minimize the delay in showing content related to the push notification.

Each of the preceding modes lets the system know that your app should be woken up or launched at appropriate times to respond to relevant events. For example, an app that begins playing music and then moves to the background still needs execution time to fill the audio output buffers. Enabling the Audio mode tells the system frameworks that they should continue to make the necessary callbacks to the app at appropriate intervals. If the app does not select this mode, any audio being played or recorded by the app stops when the app moves to the background.

Tracking the User’s Location

There are several ways to track the user’s location in the background, most of which do not actually require your app to run continuously in the background:

- The significant-change location service (Recommended)
- Foreground-only location services
- Background location services

The significant-change location service is highly recommended for apps that do not need high-precision location data. With this service, location updates are generated only when the user's location changes significantly; thus, it is ideal for social apps or apps that provide the user with noncritical, location-relevant information. If the app is suspended when an update occurs, the system wakes it up in the background to handle the update. If the app starts this service and is then terminated, the system relaunches the app automatically when a new location becomes available. This service is available in iOS 4 and later, and it is available only on devices that contain a cellular radio.

The foreground-only and background location services both use the standard location Core Location service to retrieve location data. The only difference is that the foreground-only location services stop delivering updates if the app is ever suspended, which is likely to happen if the app does not support other background services or tasks. Foreground-only location services are intended for apps that only need location data while they are in the foreground.

You enable location support from the Background modes section of the Capabilities tab in your Xcode project. (You can also enable this support by including the `UIBackgroundModes` key with the `location` value in your app's `Info.plist` file.) Enabling this mode does not prevent the system from suspending the app, but it does tell the system that it should wake up the app whenever there is new location data to deliver. Thus, this key effectively lets the app run in the background to process location updates whenever they occur.

Important: You are encouraged to use the standard services sparingly or use the significant location change service instead. Location services require the active use of an iOS device's onboard radio hardware. Running this hardware continuously can consume a significant amount of power. If your app does not need to provide precise and continuous location information to the user, it is best to minimize the use of location services.

For information about how to use each of the different location services in your app, see *Location and Maps Programming Guide*.

Playing and Recording Background Audio

An app that plays or records audio continuously (even while the app is running in the background) can register to perform those tasks in the background. You enable audio support from the Background modes section of the Capabilities tab in your Xcode project. (You can also enable this support by including the `UIBackgroundModes` key with the `audio` value in your app's `Info.plist` file.) Apps that play audio content in the background must play audible content and not silence.

Typical examples of background audio apps include:

- Music player apps
- Audio recording apps
- Apps that support audio or video playback over AirPlay
- VoIP apps

When the `UIBackgroundModes` key contains the `audio` value, the system's media frameworks automatically prevent the corresponding app from being suspended when it moves to the background. As long as it is playing audio or video content or recording audio content, the app continues to run in the background. However, if recording or playback stops, the system suspends the app.

You can use any of the system audio frameworks to work with background audio content, and the process for using those frameworks is unchanged. (For video playback over AirPlay, you can use the Media Player or AV Foundation framework to present your video.) Because your app is not suspended while playing media files, callbacks operate normally while your app is in the background. In your callbacks, though, you should do only the work necessary to provide data for playback. For example, a streaming audio app would need to download the music stream data from its server and push the current audio samples out for playback. Apps should not perform any extraneous tasks that are unrelated to playback.

Because more than one app may support audio, the system determines which app is allowed to play or record audio at any given time. The foreground app always has priority for audio operations. It is possible for more than one background app to be allowed to play audio and such determinations are based on the configuration of each app's audio session objects. You should always configure your app's audio session object appropriately and work carefully with the system frameworks to handle interruptions and other types of audio-related notifications. For information on how to configure audio session objects for background execution, see *Audio Session Programming Guide*.

Implementing a VoIP App

A *Voice over Internet Protocol (VoIP)* app allows the user to make phone calls using an Internet connection instead of the device's cellular service. Such an app needs to maintain a persistent network connection to its associated service so that it can receive incoming calls and other relevant data. Rather than keep VoIP apps awake all the time, the system allows them to be suspended and provides facilities for monitoring their sockets for them. When incoming traffic is detected, the system wakes up the VoIP app and returns control of its sockets to it.

To configure a VoIP app, you must do the following:

1. Enable support for Voice over IP from the Background modes section of the Capabilities tab in your Xcode project. (You can also enable this support by including the `UIBackgroundModes` key with the `voip` value in your app's `Info.plist` file.)
2. Configure one of the app's sockets for VoIP usage.
3. Before moving to the background, call the `setKeepAliveTimeout:handler:` method to install a handler to be executed periodically. Your app can use this handler to maintain its service connection.
4. Configure your audio session to handle transitions to and from active use.

Including the `voip` value in the `UIBackgroundModes` key lets the system know that it should allow the app to run in the background as needed to manage its network sockets. An app with this key is also relaunched in the background immediately after system boot to ensure that the VoIP services are always available.

Most VoIP apps also need to be configured as background audio apps to deliver audio while in the background. Therefore, you should include both the `audio` and `voip` values to the `UIBackgroundModes` key. If you do not do this, your app cannot play or record audio while it is in the background. For more information about the `UIBackgroundModes` key, see *Information Property List Key Reference*.

For specific information about the steps you must take to implement a VoIP app, see Tips for Developing a VoIP App.

Fetching Small Amounts of Content Opportunistically

Apps that need to check for new content periodically can ask the system to wake them up so that they can initiate a fetch operation for that content. To support this mode, enable the Background fetch option from the Background modes section of the Capabilities tab in your Xcode project. (You can also enable this support by including the `UIBackgroundModes` key with the `fetch` value in your app's `Info.plist` file.) Enabling this mode is not a guarantee that the system will give your app any time to perform background fetches. The system must balance your app's need to fetch content with the needs of other apps and the system itself. After assessing that information, the system gives time to apps when there are good opportunities to do so.

When a good opportunity arises, the system wakes or launches your app into the background and calls the app delegate's `application:performFetchWithCompletionHandler:` method. Use that method to check for new content and initiate a download operation if content is available. As soon as you finish downloading the new content, you must execute the provided completion handler block, passing a result that indicates whether content was available. Executing this block tells the system that it can move your app back to the suspended state and evaluate its power usage. Apps that download small amounts of content quickly, and accurately reflect when they had content available to download, are more likely to receive execution time in the future than apps that take a long time to download their content or that claim content was available but then do not download anything.

When downloading any content, it is recommended that you use the `NSURLSession` class to initiate and manage your downloads. For information about how to use this class to manage upload and download tasks, see *URL Session Programming Guide*.

Using Push Notifications to Initiate a Download

If your server sends push notifications to a user's device when new content is available for your app, you can ask the system to run your app in the background so that it can begin downloading the new content right away. The intent of this background mode is to minimize the amount of time that elapses between when a user sees a push notification and when your app is able to able to display the associated content. Apps are typically woken up at roughly the same time that the user sees the notification but that still gives you more time than you might have otherwise.

To support this background mode, enable the Remote notifications option from the Background modes section of the Capabilities tab in your Xcode project. (You can also enable this support by including the `UIBackgroundModes` key with the `remote-notification` value in your app's `Info.plist` file.)

For a push notification to trigger a download operation, the notification's payload must include the `content-available` key with its value set to `1`. When that key is present, the system wakes the app in the background (or launches it into the background) and calls the app delegate's `application:didReceiveRemoteNotification:fetchCompletionHandler:` method. Your implementation of that method should download the relevant content and integrate it into your app.

When downloading any content, it is recommended that you use the `NSURLSession` class to initiate and manage your downloads. For information about how to use this class to manage upload and download tasks, see *URL Session Programming Guide*.

Downloading Newsstand Content in the Background

A Newsstand app that downloads new magazine or newspaper issues can register to perform those downloads in the background. You enable support for newsstand downloads from the Background modes section of the Capabilities tab in your Xcode project. (You can also enable this support by including the `UIBackgroundModes` key with the `newsstand-content` value in your app's `Info.plist` file.) When this key is present, the system launches your app, if it is not already running, so that it can initiate the downloading of the new issue.

When you use the Newsstand Kit framework to initiate a download, the system handles the download process for your app. The system continues to download the file even if your app is suspended or terminated. When the download operation is complete, the system transfers the file to your app sandbox and notifies your app. If the app is not running, this notification wakes it up and gives it a chance to process the newly downloaded file. If there are errors during the download process, your app is similarly woken up to handle them.

For information about how to download content using the Newsstand Kit framework, see *NewsstandKit Framework Reference*.

Communicating with an External Accessory

Apps that work with external accessories can ask to be woken up if the accessory delivers an update when the app is suspended. This support is important for some types of accessories that deliver data at regular intervals, such as heart-rate monitors. You enable support for external accessory communication from the Background modes section of the Capabilities tab in your Xcode project. (You can also enable this support by including the `UIBackgroundModes` key with the `external-accessory` value in your app's `Info.plist` file.) When you enable this mode, the external accessory framework does not close active sessions with accessories. (In iOS 4 and earlier, these sessions are closed automatically when the app is suspended.) When new data arrives from the accessory, the framework wakes your app so that it can process that data. The system also wakes the app to process accessory connection and disconnection notifications.

Any app that supports the background processing of accessory updates must follow a few basic guidelines:

- Apps must provide an interface that allows the user to start and stop the delivery of accessory update events. That interface should then open or close the accessory session as appropriate.

- Upon being woken up, the app has around 10 seconds to process the data. Ideally, it should process the data as fast as possible and allow itself to be suspended again. However, if more time is needed, the app can use the `beginBackgroundTaskWithExpirationHandler:` method to request additional time; it should do so only when absolutely necessary, though.

Communicating with a Bluetooth Accessory

Apps that work with Bluetooth peripherals can ask to be woken up if the peripheral delivers an update when the app is suspended. This support is important for Bluetooth-LE accessories that deliver data at regular intervals, such as a Bluetooth heart rate belt. You enable support for using bluetooth accessories from the Background modes section of the Capabilities tab in your Xcode project. (You can also enable this support by including the `UIBackgroundModes` key with the `bluetooth-central` value in your app’s `Info.plist` file.) When you enable this mode, the Core Bluetooth framework keeps open any active sessions for the corresponding peripheral. In addition, new data arriving from the peripheral causes the system to wake up the app so that it can process the data. The system also wakes up the app to process accessory connection and disconnection notifications.

In iOS 6, an app can also operate in peripheral mode with Bluetooth accessories. To act as a Bluetooth accessory, you must enable support for that mode from the Background modes section of the Capabilities tab in your Xcode project. (You can also enable this support by including the `UIBackgroundModes` key with the `bluetooth-peripheral` value in your app’s `Info.plist` file.) Enabling this mode lets the Core Bluetooth framework wake the app up briefly in the background so that it can handle accessory-related requests. Apps woken up for these events should process them and return as quickly as possible so that the app can be suspended again.

Any app that supports the background processing of Bluetooth data must be session-based and follow a few basic guidelines:

- Apps must provide an interface that allows the user to start and stop the delivery of Bluetooth events. That interface should then open or close the session as appropriate.
- Upon being woken up, the app has around 10 seconds to process the data. Ideally, it should process the data as fast as possible and allow itself to be suspended again. However, if more time is needed, the app can use the `beginBackgroundTaskWithExpirationHandler:` method to request additional time; it should do so only when absolutely necessary, though.

Getting the User’s Attention While in the Background

Notifications are a way for an app that is suspended, is in the background, or is not running to get the user’s attention. Apps can use local notifications to display alerts, play sounds, badge the app’s icon, or a combination of the three. For example, an alarm clock app might use local notifications to play an alarm sound and display an alert to disable the alarm. When a notification is delivered to the user, the user must decide if the information warrants bringing the app back to the foreground. (If the app is already running in the foreground, local notifications are delivered quietly to the app and not to the user.)

To schedule the delivery of a local notification, create an instance of the `UILocalNotification` class, configure the notification parameters, and schedule it using the methods of the `UIApplication` class. The local notification object contains information about the type of notification to deliver (sound, alert, or badge) and the time (when applicable) at which to deliver it. The methods of the `UIApplication` class provide options for delivering notifications immediately or at the scheduled time.

Listing 3-2 shows an example that schedules a single alarm using a date and time that is set by the user. This example configures only one alarm at a time and cancels the previous alarm before scheduling a new one. (Your own apps can have no more than 128 local notifications active at any given time, any of which can be configured to repeat at a specified interval.) The alarm itself consists of an alert box and a sound file that is played if the app is not running or is in the background when the alarm fires. If the app is active and therefore running in the foreground, the app delegate’s `application:didReceiveLocalNotification:` method is called instead.

Listing 3-2 Scheduling an alarm notification

```
- (void)scheduleAlarmForDate:(NSDate*)theDate
{
    UIApplication* app = [UIApplication sharedApplication];
    NSArray*    oldNotifications = [app scheduledLocalNotifications];

    // Clear out the old notification before scheduling a new one.
    if ([oldNotifications count] > 0)
        [app cancelAllLocalNotifications];

    // Create a new notification.
    UILocalNotification* alarm = [[UILocalNotification alloc] init];
    if (alarm)
    {
        alarm.fireDate = theDate;
        alarm.timeZone = [NSTimeZone defaultTimeZone];
        alarm.repeatInterval = 0;
        alarm.soundName = @"alarmsound.caf";
        alarm.alertBody = @"Time to wake up!";

        [app scheduleLocalNotification:alarm];
    }
}
```

Sound files used with local notifications have the same requirements as those used for push notifications. Custom sound files must be located inside your app's main bundle and support one of the following formats: Linear PCM, MA4, µ-Law, or a-Law. You can also specify the `UILocalNotificationDefaultSoundName` constant to play the default alert sound for the device. When the notification is sent and the sound is played, the system also triggers a vibration on devices that support it.

You can cancel scheduled notifications or get a list of notifications using the methods of the `UIApplication` class. For more information about these methods, see *UIApplication Class Reference*. For additional information about configuring local notifications, see *Local and Remote Notification Programming Guide*.

Understanding When Your App Gets Launched into the Background

Apps that support background execution may be relaunched by the system to handle incoming events. If an app is terminated for any reason other than the user force quitting it, the system launches the app when one of the following events happens:

- For location apps:
    - The system receives a location update that meets the app's configured criteria for delivery.
    - The device entered or exited a registered region. (Regions can be geographic regions or iBeacon regions.)
- For audio apps, the audio framework needs the app to process some data. (Audio apps include those that play audio or use the microphone.)
- For Bluetooth apps:

- An app acting in the central role receives data from a connected peripheral.
- An app acting in the peripheral role receives commands from a connected central.

- For background download apps:
    - A push notification arrives for an app and the payload of the notification contains the `content-available` key with a value of `1`.
    - The system wakes the app at opportunistic moments to begin downloading new content.
    - For apps downloading content in the background using the `NSURLSession` class, all tasks associated with that session object either completed successfully or received an error.
    - A download initiated by a Newsstand app finishes.

In most cases, the system does not relaunch apps after they are force quit by the user. One exception is location apps, which in iOS 8 and later are relaunched after being force quit by the user. In other cases, though, the user must launch the app explicitly or reboot the device before the app can be launched automatically into the background by the system. When password protection is enabled on the device, the system does not launch an app in the background before the user first unlocks the device.

Being a Responsible Background App

The foreground app always has precedence over background apps when it comes to the use of system resources and hardware. Apps running in the background need to be prepared for this discrepancy and adjust their behavior when running in the background. Specifically, apps moving to the background should follow these guidelines:

- **Do not make any OpenGL ES calls from your code.** You must not create an `EAGLContext` object or issue any OpenGL ES drawing commands of any kind while running in the background. Using these calls causes your app to be killed immediately. Apps must also ensure that any previously submitted commands have completed before moving to the background. For information about how to handle OpenGL ES when moving to and from the background, see Implementing a Multitasking-aware OpenGL ES Application in *OpenGL ES Programming Guide for iOS*.
- **Cancel any Bonjour-related services before being suspended.** When your app moves to the background, and before it is suspended, it should unregister from Bonjour and close listening sockets associated with any network services. A suspended app cannot respond to incoming service requests anyway. Closing out those services prevents them from appearing to be available when they actually are not. If you do not close out Bonjour services yourself, the system closes out those services automatically when your app is suspended.
- **Be prepared to handle connection failures in your network-based sockets.** The system may tear down socket connections while your app is suspended for any number of reasons. As long as your socket-based code is prepared for other types of network failures, such as a lost signal or network transition, this should not lead to any unusual problems. When your app resumes, if it encounters a failure upon using a socket, simply reestablish the connection.
- **Save your app state before moving to the background.** During low-memory conditions, background apps may be purged from memory to free up space. Suspended apps are purged first, and no notice is given to the app before it is purged. As a result, apps should take advantage of the state preservation mechanism in iOS 6 and later to save their interface state to disk. For information about how to support this feature, see Preserving Your App’s Visual Appearance Across Launches.
- **Remove strong references to unneeded objects when moving to the background.** If your app maintains a large in-memory cache of objects (especially images), remove all strong references to those caches when moving to the background. For more information, see Reduce Your Memory Footprint.
- **Stop using shared system resources before being suspended.** Apps that interact with shared system resources such as the Address Book or calendar databases should stop using those resources before being suspended. Priority for such resources always goes to the foreground app. When your app is suspended, if it is found to be using a shared resource, the app is killed.

- **Avoid updating your windows and views.** Because your app's windows and views are not visible when your app is in the background, you should avoid updating them. The exception is in cases where you need to update the contents of a window prior to having a snapshot of your app taken.
- **Respond to connect and disconnect notifications for external accessories.** For apps that communicate with external accessories, the system automatically sends a disconnection notification when the app moves to the background. The app must register for this notification and use it to close out the current accessory session. When the app moves back to the foreground, a matching connection notification is sent, giving the app a chance to reconnect. For more information on handling accessory connection and disconnection notifications, see *External Accessory Programming Topics*.
- **Clean up resources for active alerts when moving to the background.** In order to preserve context when switching between apps, the system does not automatically dismiss action sheets (`UIActionSheet`) or alert views (`UIAlertView`) when your app moves to the background. It is up to you to provide the appropriate cleanup behavior prior to moving to the background. For example, you might want to cancel the action sheet or alert view programmatically or save enough contextual information to restore the view later (in cases where your app is terminated).
- **Remove sensitive information from views before moving to the background.** When an app transitions to the background, the system takes a snapshot of the app's main window, which it then presents briefly when transitioning your app back to the foreground. Before returning from your `applicationDidEnterBackground:` method, you should hide or obscure passwords and other sensitive personal information that might be captured as part of the snapshot.
- **Do minimal work while running in the background.** The execution time given to background apps is more constrained than the amount of time given to the foreground app. Apps that spend too much time executing in the background can be throttled back by the system or terminated.

If you are implementing a background audio app, or any other type of app that is allowed to run in the background, your app responds to incoming messages in the usual way. In other words, the system may notify your app of low-memory warnings when they occur. And in situations where the system needs to terminate apps to free even more memory, the app calls its delegate's `applicationWillTerminate:` method to perform any final tasks before exiting.

Opting Out of Background Execution

If you do not want your app to run in the background at all, you can explicitly opt out of background by adding the `UIApplicationExitsOnSuspend` key (with the value `YES`) to your app's `Info.plist` file. When an app opts out, it cycles between the not-running, inactive, and active states and never enters the background or suspended states. When the user presses the Home button to quit the app, the `applicationWillTerminate:` method of the app delegate is called and the app has approximately 5 seconds to clean up and exit before it is terminated and moved back to the not-running state.

Opting out of background execution is strongly discouraged but may be the preferred option under certain conditions. Specifically, if coding for background execution adds significant complexity to your app, terminating the app might be a simpler solution. Also, if your app consumes a large amount of memory and cannot easily release any of it, the system might kill your app quickly anyway to make room for other apps. Thus, opting to terminate, instead of switching to the background, might yield the same results and save you development time and effort.

For more information about the keys you can include in your app's `Info.plist` file, see *Information Property List Key Reference*.