Executing Code in the Background

Most applications that enter the background state are immediately suspended and do not execute any code. If your application does need to execute code, it must notify the system of that fact. There are several options available for running background tasks and each option allows your application to support specific types of behavior.

Designing Your Application to Support Multitasking

Applications linked against iPhone OS 4.0 and later are automatically assumed to support multitasking and to implement the appropriate methods to handle transitions to the background state. However, work is still required to ensure that your application transitions smoothly between the foreground and background.

Determining Whether Multitasking Support is Available

The ability to run background tasks is not supported on all iPhone OS–based devices. If a device is not running iPhone OS 4.0 and later, or if the hardware is not capable of running applications in the background, the system reverts to the previously defined behavior for handling applications. Specifically, when an application is quit, its application delegate receives an `applicationWillTerminate:` message, after which the application is terminated and purged from memory.

Applications should be prepared to handle situations where multitasking is not available. If your code supports background tasks but is able to run without them, you can use the `multitaskingSupported` property of the `UIDevice` class to determine whether multitasking is available. Of course, if your application supports versions of the system earlier than iPhone OS 4.0, you should always check the availability of this property before accessing it, as shown in Listing 4-1.

Listing 4-1 Checking for background support on earlier versions of iPhone OS

```
UIDevice* device = [UIDevice currentDevice];
BOOL backgroundSupported = NO;
if ([device respondsToSelector:@selector(isMultitaskingSupported)])
   backgroundSupported = device.multitaskingSupported;
```

Declaring the Background Tasks You Support

Support for some types of background execution must be declared in advance by the application that uses them. An application does this by including the `UIBackgroundModes` key in its `Info.plist` file. This key identifies which background tasks your application supports. Its value is an array that contains one or more strings with the following values:

- `audio` - The application plays audible content to the user while in the background.

- `location` - The application keeps users informed of their location, even while running in the background.
- `voip` - The application provides the ability for the user to make phone calls using an Internet connection.

Each of the preceding values lets the system know that your application should be woken up at appropriate times to respond to relevant events. For example, an application that begins playing music and then moves to the background still needs execution time to fill the audio output buffers. Including the `audio` key tells the system frameworks that they should continue playing and make the necessary callbacks to the application at appropriate intervals. If the application did not include this key, any audio being played by the application would stop when the application moved to the background.

In addition to the preceding keys, iPhone OS provides two other ways to do work in the background:

- Applications can ask the system for extra time to complete a given task.
- Applications can schedule local notifications to be delivered at a predetermined time.

For more information about how to initiate background tasks from your code, see "Initiating Background Tasks" (page 56).

Supporting Background State Transitions

Supporting the background state transition is part of the fundamental architecture for applications in iPhone OS 4.0 and later. Although technically the only thing you have to do to support this capability is link against iPhone OS 4.0 and later, properly supporting it requires some additional work. Specifically, your application delegate should implement the following methods and implement appropriate behaviors in each of them:

- `application:didFinishLaunchingWithOptions:`
- `applicationDidBecomeActive:`
- `applicationWillResignActive:`
- `applicationDidEnterBackground:`
- `applicationWillEnterForeground:`
- `applicationWillTerminate:`

These methods are called at key times during your application's execution to let you know when your application is changing states. Although many of these methods have been present in iPhone applications for some time, their purpose was more narrowly defined previously. Applications implementing these methods now must use them to address some additional behaviors that are needed to operate safely in the background. For example, an application that went into the background and purged from memory might be relaunched and expected to continue operating from where it last was. Thus, your application's `application:didFinishLaunchingWithOptions:` method may now need to check and see what additional launch information is available and use that information plus any saved state to restore the application's user interface.

For information and guidance on how to implement the methods in your application, see "Understanding an Application's States and Transitions" (page 27).

Being a Responsible, Multitasking-Aware Application

Applications running in the background are more limited in what they can do than a foreground application. Although the window and view objects of a background application still exist in memory, they are not displayed on screen.

- **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. Using these calls will cause your application to be terminated immediately.
- **Cancel any network-related services before being suspended.** When your application 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 application cannot respond to incoming service requests anyway. Closing out those services prevents them from appearing to be available when they actually are not.
- **Save your application state before moving to the background.** During low-memory conditions, background applications are purged from memory to free up space. Suspended applications are purged first, and no notice is given to the application before it is purged. As a result, before moving to the background, an application should always save enough state information to reconstitute itself later if necessary. Restoring your application to its previous state also provides consistency for the user, who will see a snapshot of your application's main window briefly when it is relaunched.
- **Release any unneeded memory when moving to the background.** Background applications must maintain enough state information to be able to resume operation quickly when they move back to the foreground. But if there are objects or relatively large memory blocks that you no longer need (such as unused images), you should consider releasing that memory when moving to the background.
- **Avoid using shared system resources.** Applications that interact with shared system resources such as Address Book should avoid accessing those resources while in the background. Priority over such resources always goes to the foreground application, and any background applications found to be holding onto a shared resource will be terminated.
- **Avoid updating your windows and views.** While in the background, your application's windows and views are not visible, so you should not try to update them. Although creating and manipulating window and view objects in the background will not cause your application to be terminated, this work should be postponed until your application moves to the foreground.
- **Remove sensitive information from views before moving to the background.** When an application transitions to the background, the system takes a snapshot of the application's main window, which it then presents briefly when transitioning your application 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 applications is more constrained than the amount of time given to the foreground application. If your application plays background audio or monitors location changes, you should focus on that task only and defer any nonessential tasks until later. Applications that spend too much time executing in the background can be throttled back further by the system or terminated altogether.

If you are implementing a background audio application, or any other type of application that is not suspended while in the background, your application responds to incoming messages in the usual way. In other words, the system may notify your application of low-memory warnings when they occur. And in situations where the system needs to terminate applications to free up even more memory, the application calls its delegate's `applicationWillTerminate:` method to perform any final tasks before exiting.

Initiating Background Tasks

The steps for initiating a background task depend entirely on the task at hand. Some tasks must be initiated explicitly, while others happen in a more automatic way for your application. The following sections provide guidance and examples on how to initiate each type of background task.

Completing a Long-Running Task in the Background

Any time before it is suspended, an application can call the `beginBackgroundTaskWithExpirationHandler:` method to ask the system for extra time to complete some long-running task in the background. If the request is granted, and if the application goes into the background while the task is in progress, the system lets the application run for an additional amount of time instead of suspending it. (The amount of time left for the application to run is available in the `backgroundTimeRemaining` property of the `UIApplication` object.)

You can use background tasks to ensure that important but potentially long-running operations do not end abruptly when the user quits the application. For example, you might use this technique to finish downloading an important file from a network server. There are a couple of design patterns you can use to initiate such tasks:

- Wrap any long-running critical tasks with `beginBackgroundTaskWithExpirationHandler:` and `endBackgroundTask:` calls. This protects those tasks in situations where your application is suddenly moved to the background.
- Wait for your application delegate's `applicationDidEnterBackground:` method to be called and start one or more tasks then.

All calls to the `beginBackgroundTaskWithExpirationHandler:` method must be balanced with a corresponding call to the `endBackgroundTask:` method. The `endBackgroundTask:` method lets the system know that the desired task is complete and that the application can now be suspended. Because applications running in the background have a limited amount of execution time, calling this method is also critical to avoid the termination of your application. (You can always check the value in the `backgroundTimeRemaining` property to see how much time is left.) An application with outstanding background tasks is terminated rather than suspended when its time limit expires. You can also provide an expiration handler for each task and call the `endBackgroundTask:` from there.

An application may have any number of background tasks running at the same time. Each time you start a task, the `beginBackgroundTaskWithExpirationHandler:` method returns a unique identifier for the task. You must pass this same identifier to the `endBackgroundTask:` method when it comes time to end the task.

Listing 4-2 shows how to start a long-running task when your application 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.

Listing 4-2 Starting a background task at quit time

```
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    UIApplication*    app = [UIApplication sharedApplication];

   // Request permission to run in the background. Provide an
   // expiration handler in case the task runs long.
   NSAssert(self->bgTask == UIInvalidBackgroundTask, nil);

   self->bgTask = [app beginBackgroundTaskWithExpirationHandler:^{
      // Synchronize the cleanup call on the main thread in case
      // the task actually finishes at around the same time.
      dispatch_async(dispatch_get_main_queue(), ^{
         if (self->bgTask != UIInvalidBackgroundTask)
         {
            [app endBackgroundTask:self->bgTask];
            self->bgTask = UIInvalidBackgroundTask;
         }
      });
   }];

   // 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.

      // Synchronize the cleanup call on the main thread in case
      // the expiration handler is fired at the same time.
      dispatch_async(dispatch_get_main_queue(), ^{
         if (self->bgTask != UIInvalidBackgroundTask)
         {
            [app endBackgroundTask:self->bgTask];
            self->bgTask = UIInvalidBackgroundTask;
         }
      });
   });
}
```

In your own expiration handlers, you can include additional code needed to close out your task. However, any code you include should not take long to execute. By the time your expiration handler is called, your application is already very close to its time limit. Doing anything other than clean up your state information and exiting could cause your application to be terminated.

Scheduling the Delivery of Local Notifications

The `UILocalNotification` class in UIKit provides a way to schedule the delivery of push notifications locally. Unlike push notifications, which require setting up a remote server, local notifications are scheduled by your application and executed on the current device. You can use this capability to achieve the following behaviors:

- A time-based application can ask the system to post an alert at a specific time in the future. For example, an alarm clock application would use this to implement alarms.
- An application running in the background can post a local notification to get the user's attention.

To schedule the delivery of a local notification, create an instance of the `UILocalNotification` class, configure it, 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 4-3 shows an example that schedules a single alarm using a date and time that is set by the user. This example supports only one alarm at a time and therefore cancels the previous alarm before scheduling a new one. Your own applications can schedule up to 128 simultaneous notifications, any of which can be configured to repeat at a specified interval. The alarm itself consists of a sound file and an alert box that is played if the application is not running or is in the background when the alarm fires. If the application is active and therefore running in the foreground, the application delegate's `application:didReceiveLocalNotification:` method is called instead.

Listing 4-3 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] autorelease];
    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 application's main bundle and support one of the following formats: linear PCM, MA4, uLaw, or aLaw. You can also specify a sound name of `default` 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 Push Notification Programming Guide*.

Receiving Location Events in the Background

Applications that use location data to track the user's position now have the option to do so while running in the background. In fact, applications now have more options for receiving location events:

- **Applications can register for significant location changes only. (Recommended)** The significant location change service is available in iPhone OS 4.0 and later for devices with a cellular radio. It offers a low-power way to receive location data and is highly recommended. New location updates are provided only when the user's position changes significantly. If the application is suspended while running this service, new location updates will cause the application to be woken up in the background to handle them. Similarly, if the application is terminated while running this service, the system relaunches the application automatically when new location data becomes available.
- **Applications can continue to use the standard location services.** These services are available in all versions of iPhone OS and provide continuous updates while the application is running in the foreground or background. However, if the application is suspended or terminated, new location events do not cause the application to be woken up or relaunched.
- **An application can declare itself as a continuous background location application.** An application that needs the regular location updates offered by the standard location services may declare itself as a continuous background application. It does this by including the `UIBackgroundModes` key in its `Info.plist` file and setting the value of this key to an array containing the `location` string. If an application with this key is running location services when it enters the background, it is not suspended by the system. Instead, it is allowed to continue running so that it may perform location-related tasks in the background.

Regardless of which option you pick, running location services continuously in the background requires enabling the appropriate radio hardware and keeping it active, which can require a significant amount of battery power. If your application does not require precise location information, the best solution is to use the significant location service. You start this service by calling the `startMonitoringSignificantLocationChanges` method of `CLLocationManager`. This service gives the system more freedom to power down the radio hardware and send notifications only when significant location changes are detected. For example, the location manager typically waits until the cell radio detects and starts using a new cell tower for communication.

Another advantage of the significant location service is its ability to wake up and relaunch your application. If a location event arrives while the application is suspended, the application is woken up and given a small amount of time to handle the event. If the application is terminated and a new event arrives, the application is relaunched. Upon relaunch, the options dictionary passed to the `application:didFinishLaunchingWithOptions:` method contains the `UIApplicationLaunchOptionsLocationKey` key to let the application know that a new location event is available. The application can then restart location services and receive the new event.

For applications that need more precise location data at regular intervals, such as navigation applications, you may need to declare the application as a continuous background application. This option is the least desirable option because it increases power usage considerably. Not only does the application consume power by running in the background, it must also keep location services active, which also consumes power. As a result, you should avoid this option whenever possible.

Regardless of whether you use the significant location change service or the standard location services, the process for actually receiving location updates is unchanged. After starting location services, the `locationManager:didUpdateToLocation:fromLocation:` method of your location manager delegate is called whenever an event arrives. Of course, applications that run continuously in the background may also want to adjust the implementation of this method to do less work while in the background. For example, you might want to update the application's views only while the application is in the foreground.

For more information about how to use location services in your application, see *Location Awareness Programming Guide*.

Playing Background Audio

Applications that play audio can continue playing that audio while in the background. To indicate that your application plays background audio, include the `UIBackgroundModes` key to its `Info.plist` file. The value for this key is an array containing the `audio` string. When this key is present, the system's audio frameworks automatically prevent your application from being suspended when it moves to the background. Your application continues to run in the background as long as it is playing audio. However, if this key is not present when the application moves to the background, or if your application stops playing audio while in the background, your application is suspended.

You can use any of the system audio frameworks to initiate the playback of background audio and the process for using those frameworks is unchanged. Because your application is not suspended while playing audio, the audio callbacks operate normally while your application is in the background. While running in the background, your application should limit itself to doing only the work necessary to provide audio data for playback. Thus, a streaming audio application would download any new data from its server and push the current audio samples out for playback.

Implementing a VoIP Application

A **Voice over Internet Protocol (VoIP)** application allows the user to make phone calls using an Internet connection instead of the device's cellular service. Such an application needs to maintain a persistent network connection to its associated service. Among other things, this persistent connection allows the application to receive and respond to incoming calls. Rather than keep a VoIP application awake so that it can maintain its network connection, the system provides facilities for managing the sockets associated with that connection while the application is suspended.

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

1. Add the `UIBackgroundModes` key to your application's `Info.plist` file. Set the value of this key to an array that includes the `voip` string.

2. Configure your socket for VoIP usage.

3. Before moving to the background, call the `setKeepAliveTimeout:handler:` method to specify the frequency at which your application must be woken to maintain your service.

Most VoIP applications also need to be configured as background audio applications in order to process audio while in the background. To do so, you must add both the `audio` and `voip` values to the `UIBackgroundModes` key. Otherwise, there are no changes to how you use the audio frameworks to play and record audio.

Configuring Sockets for VoIP Usage

In order for your application to maintain a persistent connection while it is in the background, you must configure the sockets used to communicate with your VoIP service. In iPhone OS, most sockets are managed using higher-level constructs such as streams. As a result, configuration of a socket to support VoIP occurs through the higher-level interfaces. The only thing you have to do beyond the normal configuration is add a special key that tags the interface as being used for a VoIP service. Table 4-1 lists the interfaces that you can configure for VoIP usage and the keys you assign.

Table 4-1 Configuring stream interfaces for VoIP usage

Interface	Configuration
`NSInputStream` and `NSOutputStream`	For Cocoa streams, use the `setProperty:forKey:` method to add the `NSStreamNetworkServiceType` property to the stream. The value of this property should be set to `NSStreamNetworkServiceTypeVoIP`.
`NSURLRequest`	When using the URL loading system, use the `setNetworkServiceType:` method of your `NSMutableURLRequest` object to set the network service type of the request. The service type should be set to `NSURLNetworkServiceTypeVoIP`.
`CFReadStreamRef` and `CFWriteStreamRef`	For Core Foundation streams, use the `CFReadStreamSetProperty` or `CFWriteStreamSetProperty` function to add the `kCFStreamNetwork-ServiceType` property to the stream. The value for this property should be set to `kCFStreamNetworkServiceTypeVoIP`.

When you configure a stream for VoIP usage, the system takes over management of the underlying socket while your application is suspended. This handoff to the system is transparent to your application. If new data arrives while your application is suspended, the system wakes up your application so that it can process the data. In the case of an incoming phone call, your application would typically alert the user immediately using a local notification. For other noncritical data, or if the user ignores the call, the system returns your application to the suspended state when it finishes processing the data.

Because VoIP applications need to stay running in order to receive incoming calls, the system automatically relaunches the application if it exits with a nonzero exit code. (This could happen in cases where there is memory pressure and your application is terminated as a result.) However, the system does not maintain your socket connections between different launches of your application. Therefore, at launch time, your application always needs to recreate those connections from scratch.

For more information about configuring Cocoa stream objects, see *Stream Programming Guide for Cocoa*. For information about using URL requests, see *URL Loading System Programming Guide*. And for information about configuring streams using the CFNetwork interfaces, see *CFNetwork Programming Guide*.

Installing a Keep-Alive Handler

To prevent the loss of its connection, a VoIP application typically needs to wake up periodically and check in with its server. To facilitate this behavior, iPhone OS lets you install a special handler using the `setKeepAliveTimeout:handler:` method of `UIApplication`. You typically install this handler when moving to the background. Once installed, the system calls your handler at least once before the timeout interval expires, waking up your application as needed to do so. You can use this handler to check in with your VoIP service and perform any other relevant housekeeping chores.

Your keep-alive handler executes in the background and must return as quickly as possible. Handlers are given a maximum of 30 seconds to perform any needed tasks and return. If a handler has not returned after 30 seconds, the system terminates the application.

When installing your handler, you should specify the largest timeout value that is practical for your needs. The minimum allowable interval for running your handler is 300 seconds and attempting to install a handler with a smaller timeout value will fail. Although the system promises to call your handler block before the timeout value expires, it does not guarantee the exact call time. To improve battery life, the system typically groups the execution of your handler with other periodic system tasks, thereby processing all tasks in one quick burst. As a result, your handler code must be prepared to run earlier than the actual timeout period you specified.

Responding to System Changes While in the Background

While an application is in the suspended state, it does not receive system-related events that might be of interest. However, the most relevant events are captured by the system and queued for later delivery to your application. To prevent your application from becoming overloaded with notifications when it resumes, the system coalesces events and delivers a single event (of each relevant type) corresponding to the net change since your application was suspended.

To understand how this might work in your application, consider an example. Suppose that at the time when your application is suspended, the device is in a portrait orientation. While the application is suspended, the user rotates the device to landscape left, upside down, and finally landscape right orientations before launching your application again. Upon resumption, your application would receive a single device orientation event indicating that the device changed to a landscape-right orientation. Of course, if your application uses view controllers, the orientation of those view controllers would be updated automatically by UIKit. Your application would need to respond only if it tracked device orientation changes explicitly.

Table 4-2 lists the events that are coalesced and delivered to your application. In most cases, the events are delivered in the form of a notification object. However, some events may be intercepted by a system framework and delivered to your application by another means. Unless otherwise noted, all events are delivered regardless of whether your application resumes in the foreground or background.

Table 4-2 Notifications delivered to waking applications

Event	Notification mechanism
Your code marks a view as dirty	Any calls to `setNeedsDisplay` or `setNeedsDisplayInRect:` on one of your views are coalesced and stored until your application resumes in the foreground. These events are not delivered to applications running in the background.
An accessory is connected or disconnected	`EAAccessoryDidConnectNotification` `EAAccessoryDidDisconnectNotification`
The device orientation changes	`UIDeviceOrientationDidChangeNotification` In addition to this notification, view controllers update their interface orientations automatically.
There is a significant time change	`UIApplicationSignificantTimeChangeNotification`
The battery level or battery state changes	`UIDeviceBatteryLevelDidChangeNotification` `UIDeviceBatteryStateDidChangeNotification`
The proximity state changes	`UIDeviceProximityStateDidChangeNotification`
The status of protected files changes	`UIApplicationProtectedDataWillBecomeUnavailable` `UIApplicationProtectedDataDidBecomeAvailable`

Event	Notification mechanism
An external display is connected or disconnected	`UIScreenDidConnectNotification` `UIScreenDidDisconnectNotification`
The screen mode of a display changes	`UIScreenModeDidChangeNotification`
Preferences that your application exposes through the Settings application are changed	`NSUserDefaultsDidChangeNotification`
The current language or locale settings changes	`NSCurrentLocaleDidChangeNotification`

When your application resumes, any queued events are delivered via your application's main run loop. Because these events are queued right away, they are typically delivered before any touch events or other user input. Most applications should be able to handle these events quickly enough that they would not cause any noticeable lag when resumed. However, if your application appears sluggish in responding to user input when it is woken up, you might want to analyze your application using Instruments and see if your handler code is causing the delay.

Handling Locale Changes Gracefully

If the user changes the language or locale of the device while your application is suspended, the system notifies you of that change using the `NSCurrentLocaleDidChangeNotification` notification. You can use this notification to force updates to any views containing locale-sensitive information, such as dates, times, and numbers. Of course, you should also be careful to write your code in ways that might make it easy to update things easily:

- Use the `autoupdatingCurrentLocale` class method when retrieving `NSLocale` objects. This method returns a locale object that updates itself automatically in response to changes, so you never need to recreate it.
- Avoid caching `NSFormatter` objects. Date and number formatters must be recreated whenever the current locale information changes.

Responding to Changes in Your Application's Settings

If your application has settings that are managed by the Settings application, it should observe the `NSUserDefaultsDidChangeNotification` notification. Because the user can modify settings while your application is in the background, you can use this notification to respond to any important changes in those settings. For example, an email program would need to respond to changes in the user's mail account information. Failure to do so could have serious privacy and security implications. Specifically, the user might still be able to send email using the old account information, even if the account did not belong to that person.

Upon receiving the `NSUserDefaultsDidChangeNotification` notification, your application should reload any relevant settings and, if necessary, reset its user interface appropriately. In cases where passwords or other security-related information has changed, you should also hide any previously displayed information and force the user to enter the new password.