Performance Tips

At each step in the development of your app, consider the implications of your design choices on the overall performance of your app. Power usage and memory consumption are extremely important considerations for iOS apps, and there are many other considerations as well. The following sections describe the factors you should consider throughout the development process.

Reduce Your App's Power Consumption

Power consumption on mobile devices is always an issue. The power management system in iOS conserves power by shutting down any hardware features that are not currently being used. You can help improve battery life by optimizing your use of the following features:

- The CPU
- Wi-Fi, Bluetooth, and baseband (EDGE, 3G) radios
- The Core Location framework
- The accelerometers
- The disk

The goal of your optimizations should be to do the most work you can in the most efficient way possible. You should always optimize your app's algorithms using Instruments. But even the most optimized algorithm can still have a negative impact on a device's battery life. You should therefore consider the following guidelines when writing your code:

- Avoid doing work that requires polling. Polling prevents the CPU from going to sleep. Instead of polling, use the `NSRunLoop` or `NSTimer` classes to schedule work as needed.
- Leave the `idleTimerDisabled` property of the shared `UIApplication` object set to `NO` whenever possible. The idle timer turns off the device's screen after a specified period of inactivity. If your app does not need the screen to stay on, let the system turn it off. If your app experiences side effects as a result of the screen being turned off, you should modify your code to eliminate the side effects rather than disable the idle timer unnecessarily.
- Coalesce work whenever possible to maximize idle time. It generally takes less power to perform a set of calculations all at once than it does to perform them in small chunks over an extended period of time. Doing small bits of work periodically requires waking up the CPU more often and getting it into a state where it can perform your tasks.
- Avoid accessing the disk too frequently. For example, if your app saves state information to the disk, do so only when that state information changes, and coalesce changes whenever possible to avoid writing small changes at frequent intervals.
- Do not draw to the screen faster than is needed. Drawing is an expensive operation when it comes to power. Do not rely on the hardware to throttle your frame rates. Draw only as many frames as your app actually needs.
- If you use the `UIAccelerometer` class to receive regular accelerometer events, disable the delivery of those events when you do not need them. Similarly, set the frequency of event delivery to the smallest value that is suitable for your needs. For more information, see *Event Handling Guide for iOS*.

The more data you transmit to the network, the more power must be used to run the radios. In fact, accessing the network is the most power-intensive operation you can perform. You can minimize that time by following these guidelines:

- Connect to external network servers only when needed, and do not poll those servers.
- When you must connect to the network, transmit the smallest amount of data needed to do the job. Use compact data formats, and do not include excess content that simply is ignored.

- Transmit data in bursts rather than spreading out transmission packets over time. The system turns off the Wi-Fi and cell radios when it detects a lack of activity. When it transmits data over a longer period of time, your app uses much more power than when it transmits the same amount of data in a shorter amount of time.

  When using the `NSURLSession` class to enqueue multiple upload or download tasks, enqueue those items together rather than waiting for one to finish before starting the next one. The system manages automatically executes queued tasks when it is most efficient to do so.

- Connect to the network using the Wi-Fi radios whenever possible. Wi-Fi uses less power and is preferred over cellular radios.
- If you use the Core Location framework to gather location data, disable location updates as soon as you can and set the distance filter and accuracy levels to appropriate values. Core Location uses the available GPS, cell, and Wi-Fi networks to determine the user's location. Although Core Location works hard to minimize the use of these radios, setting the accuracy and filter values gives Core Location the option to turn off hardware altogether in situations where it is not needed. For more information, see *Location and Maps Programming Guide*.

The Instruments app includes several instruments for gathering power-related information. You can use these instruments to gather general information about power consumption and to gather specific measurements for hardware such as the Wi-Fi and Bluetooth radios, GPS receiver, display, and CPU. You can also enable Energy Diagnostics Logging on a device to gather information. For information about using Instruments to gather power-related data, see *Instruments User Guide*. For information about how to enable Energy Diagnostics Logging on a device, see *Instruments Help*.

Use Memory Efficiently

Apps are encouraged to use as little memory as possible so that the system may keep more apps in memory or dedicate more memory to foreground apps that truly need it. There is a direct correlation between the amount of free memory available to the system and the relative performance of your app. Less free memory means that the system is more likely to have trouble fulfilling future memory requests.

To ensure there is always enough free memory available, you should minimize your app's memory usage and be responsive when the system asks you to free up memory.

Observe Low-Memory Warnings

When the system dispatches a low-memory warning to your app, *respond immediately*. Low-memory warnings are your opportunity to remove references to objects that you do not need. Responding to these warnings is crucial because apps that fail to do so are more likely to be terminated. The system delivers memory warnings to your app using the following APIs:

- The `applicationDidReceiveMemoryWarning:` method of your app delegate.
- The `didReceiveMemoryWarning` method of your `UIViewController` classes.
- The `UIApplicationDidReceiveMemoryWarningNotification`notification.
- Dispatch sources of type `DISPATCH_SOURCE_TYPE_MEMORYPRESSURE`. This technique is the only one that you can use to distinguish the severity of the memory pressure.

Upon receiving any of these warnings, your handler method should respond by immediately freeing up any unneeded memory. Use the warnings to clear out caches and release images. If you have large data structures that are not being used, write those structures to disk and release the in-memory copies of the data.

If your data model includes known purgeable resources, you can have a corresponding manager object register for the `UIApplicationDidReceiveMemoryWarningNotification` notification and remove strong references to its purgeable resources directly. Handling this notification directly avoids the need to route all memory warning calls through the app delegate.

Note: You can test your app's behavior under low-memory conditions using the Simulate Memory Warning command in iOS Simulator.

Reduce Your App's Memory Footprint

Starting off with a low footprint gives you more room for expanding your app later. Table 7-1 lists some tips on how to reduce your app's overall memory footprint.

Table 7-1 Tips for reducing your app's memory footprint

Tip	Actions to take
Eliminate memory leaks.	Because memory is a critical resource in iOS, your app should never have memory leaks. Use the Instruments app to track down leaks in your code, both in Simulator and on actual devices. For more information on using Instruments, see *Instruments User Guide*.
Make resource files as small as possible.	Files reside on disk but must be loaded into memory before they can be used. Compress all image files to make them as small as possible. (To compress PNG images—the preferred image format for iOS apps—use the `pngcrush` tool.) You can make property list files smaller by writing them out in a binary format using the `NSPropertyListSerialization` class.
Use Core Data or SQLite for large data sets.	If your app manipulates large amounts of structured data, store it in a Core Data persistent store or in a SQLite database instead of in a flat file. Both Core Data and SQLite provides efficient ways to manage large data sets without requiring the entire set to be in memory all at once.
Load resources lazily.	You should never load a resource file until it is actually needed. Prefetching resource files may seem like a way to save time, but this practice actually slows down your app right away. In addition, if you end up not using the resource, loading it wastes memory for no good purpose.

Allocate Memory Wisely

Table 7-2 lists tips for improving memory usage in your app.

Table 7-2 Tips for allocating memory

Tip	Actions to take
Impose size limits on resources.	Avoid loading a large resource file when a smaller one will do. Instead of using a high-resolution image, use one that is appropriately sized for iOS-based devices. If you must use large resource files, find ways to load only the portion of the file that you need at any given time. For example, rather than load the entire file into memory, use the `mmap` and `munmap` functions to map portions of the file into and out of memory. For more information about mapping files into memory, see *File-System Performance Guidelines*.
Avoid unbounded problem sets.	Unbounded problem sets might require an arbitrarily large amount of data to compute. If the set requires more memory than is available, your app may be unable to complete the calculations. Your apps should avoid such sets whenever possible and work on problems with known memory limits.

For detailed information about ARC and memory management, see *Transitioning to ARC Release Notes*.

Tune Your Networking Code

The networking stack in iOS includes several interfaces for communicating over the radio hardware of iOS devices. The main programming interface is the CFNetwork framework, which builds on top of BSD sockets and opaque types in the Core Foundation framework to communicate with network entities. You can also use the `NSStream` classes in the Foundation framework and the low-level BSD sockets found in the Core OS layer of the system.

For information about how to use the CFNetwork framework for network communication, see *CFNetwork Programming Guide* and *CFNetwork Framework Reference*. For information about using the `NSStream` class, see *Foundation Framework Reference*.

Tips for Efficient Networking

Implementing code to receive or transmit data across the network is one of the most power-intensive operations on a device. Minimizing the amount of time spent transmitting or receiving data helps improve battery life. To that end, you should consider the following tips when writing your network-related code:

- For protocols you control, define your data formats to be as compact as possible.
- Avoid using chatty protocols.
- Transmit data packets in bursts whenever you can.

Cellular and Wi-Fi radios are designed to power down when there is no activity. Depending on the radio, though, doing so can take several seconds. If your app transmits small bursts of data every few seconds, the radios may stay powered up and continue to consume power, even when they are not actually doing anything. Rather than transmit small amounts of data more often, it is better to transmit a larger amount of data once or at relatively large intervals.

When communicating over the network, packets can be lost at any time. Therefore, when writing your networking code, you should be sure to make it as robust as possible when it comes to failure handling. It is perfectly reasonable to implement handlers that respond to changes in network conditions, but do not be surprised if those handlers are not called consistently. For example, the Bonjour networking callbacks may not always be called immediately in response to the disappearance of a network service. The Bonjour system service immediately invokes browsing callbacks when it receives a notification that a service is going away, but network services can disappear without notification. This situation might occur if the device providing the network service unexpectedly loses network connectivity or the notification is lost in transit.

Using Wi-Fi

If your app accesses the network using the Wi-Fi radios, you must notify the system of that fact by including the `UIRequiresPersistentWiFi` key in the app's `Info.plist` file. The inclusion of this key lets the system know that it should display the network selection dialog if it detects any active Wi-Fi hot spots. It also lets the system know that it should not attempt to shut down the Wi-Fi hardware while your app is running.

To prevent the Wi-Fi hardware from using too much power, iOS has a built-in timer that turns off the hardware completely after 30 minutes if no running app has requested its use through the `UIRequiresPersistentWiFi` key. If the user launches an app that includes the key, iOS effectively disables the timer for the duration of the app's life cycle. As soon as that app quits or is suspended, however, the system reenables the timer.

Note: Note that even when `UIRequiresPersistentWiFi` has a value of `true`, it has no effect when the device is idle (that is, screen-locked). The app is considered inactive, and although it

may function on some levels, it has no Wi-Fi connection.

For more information on the `UIRequiresPersistentWiFi` key and the keys of the `Info.plist` file, see The Information Property List File.

The Airplane Mode Alert

If your app launches while the device is in airplane mode, the system may display an alert to notify the user of that fact. The system displays this alert only when all of the following conditions are met:

- Your app's information property list (`Info.plist`) file contains the `UIRequiresPersistentWiFi` key and the value of that key is set to `true`.
- Your app launches while the device is currently in airplane mode.
- Wi-Fi on the device has not been manually reenabled after the switch to airplane mode.

Improve Your File Management

Minimize the amount of data you write to the disk. File operations are relatively slow and involve writing to the flash drive, which has a limited lifespan. Some specific tips to help you minimize file-related operations include:

- Write only the portions of the file that changed, and aggregate changes when you can. Avoid writing out the entire file just to change a few bytes.
- When defining your file format, group frequently modified content together to minimize the overall number of blocks that need to be written to disk each time.
- If your data consists of structured content that is randomly accessed, store it in a Core Data persistent store or a SQLite database, especially if the amount of data you are manipulating could grow to more than a few megabytes.

Avoid writing cache files to disk. The only exception to this rule is when your app quits and you need to write state information that can be used to put your app back into the same state when it is next launched.

Make App Backups More Efficient

Backups occur wirelessly via iCloud or when the user syncs the device with iTunes. During backups, files are transferred from the device to the user's computer or iCloud account. The location of files in your app sandbox determines whether or not those files are backed up and restored. If your application creates many large files that change regularly and puts them in a location that is backed up, backups could be slowed down as a result. As you write your file-management code, you need to be mindful of this fact.

App Backup Best Practices

You do not have to prepare your app in any way for backup and restore operations. Devices with an active iCloud account have their app data backed up to iCloud at appropriate times. For devices that are plugged into a computer, iTunes performs an incremental backup of the app's data files. However, iCloud and iTunes do not back up the contents of the following directories:

- *<Application_Home>*`/`*AppName*`.app`
- *<Application_Data>*`/Library/Caches`
- *<Application_Data>*`/tmp`

To prevent the syncing process from taking a long time, be selective about where you place files inside your app's home directory. Apps that store large files can slow down the process of backing up to iTunes or iCloud. These apps can also consume a large amount of a user's available storage, which may encourage the user to delete the app or disable backup of that app's data to iCloud. With this in mind, you should store app data according to the following guidelines:

- Critical data should be stored in the *<Application_Data>*`/Documents` directory. Critical data is any data that cannot be recreated by your app, such as user documents and other user-generated content.
- Support files include files your application downloads or generates and that your application can recreate as needed. The location for storing your application's support files depends on the current iOS version.
  - In iOS 5.1 and later, store support files in the *<Application_Data>*`/Library/Application Support` directory and add the `NSURLIsExcludedFromBackupKey` attribute to the corresponding `NSURL` object using the `setResourceValue:forKey:error:` method. (If you are using Core Foundation, add the `kCFURLIsExcludedFromBackupKey` key to your `CFURLRef` object using the `CFURLSetResourcePropertyForKey` function.) Applying this attribute prevents the files from being backed up to iTunes or iCloud. If you have a large number of support files, you may store them in a custom subdirectory and apply the extended attribute to just the directory.
  - In iOS 5.0 and earlier, store support files in the *<Application_Data>*`/Library/Caches` directory to prevent them from being backed up. If you are targeting iOS 5.0.1, see *How do I prevent files from being backed up to iCloud and iTunes?* for information about how to exclude files from backups.
- Cached data should be stored in the *<Application_Data>*`/Library/Caches` directory. Examples of files you should put in the `Caches` directory include (but are not limited to) database cache files and downloadable content, such as that used by magazine, newspaper, and map apps. Your app should be able to gracefully handle situations where cached data is deleted by the system to free up disk space.
- Temporary data should be stored in the *<Application_Data>*`/tmp` directory. Temporary data comprises any data that you do not need to persist for an extended period of time. Remember to delete those files when you are done with them so that they do not continue to consume space on the user's device.

Although iTunes backs up the app bundle itself, it does not do this during every sync operation. Apps purchased directly from a device are backed up when that device is next synced with iTunes. Apps are not backed up during subsequent sync operations, though, unless the app bundle itself has changed (because the app was updated, for example).

For additional guidance about how you should use the directories in your app, see *File System Programming Guide*.

Files Saved During App Updates

When a user downloads an app update, iTunes installs the update in a new app directory. It then moves the user's data files from the old installation over to the new app directory before deleting the old installation. Files in the following directories are guaranteed to be preserved during the update process:

- *<Application_Data>*`/Documents`
- *<Application_Data>*`/Library`

Although files in other user directories may also be moved over, you should not rely on them being present after an update.

Move Work off the Main Thread

Be sure to limit the type of work you do on the main thread of your app. The main thread is where

your app handles touch events and other user input. To ensure that your app is always responsive to the user, you should never use the main thread to perform long-running or potentially unbounded tasks, such as tasks that access the network. Instead, you should always move those tasks onto background threads. The preferred way to do so is to use Grand Central Dispatch (GCD) or `NSOperation` objects to perform tasks asynchronously.

Moving tasks into the background leaves your main thread free to continue processing user input, which is especially important when your app is starting up or quitting. During these times, your app is expected to respond to events in a timely manner. If your app's main thread is blocked at launch time, the system could kill the app before it even finishes launching. If the main thread is blocked at quitting time, the system could similarly kill the app before it has a chance to write out crucial user data.

For more information about using GCD, operation objects, and threads, see *Concurrency Programming Guide*.
