Names: Omar Osama-Mouhab Mahmoud-Abdallah Mahmoud

**Power Management in iOS**

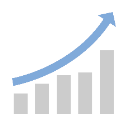
All apps consume energy—whenever they perform networking operations, update the user interface, or run code on the CPU. As users rely increasingly on battery power—and as apps proliferate—energy efficiency becomes integral to the user experience.

**A great user experience requires:**

**Great battery life.** As energy efficiency goes down, so does battery life. Users expect all-day battery life on their iOS devices.



**Awesome speed.** iOS is designed to provide great performance during complex operations—and to make your app fly.



**Responsiveness.** Too many resources being consumed at once can result in a laggy interface that’s slow to respond to user input.



**Cool device.** As more apps use more resources, the system works harder and faster, and the physical temperature of a device gradually rises. When this occurs, the system takes steps to cool down to a more acceptable level.



**iOS Energy-Saving Technologies**

iOS employs advanced energy-saving technologies that help users get the most out of their devices. These features help the system make smart decisions about how to use resources and run code as efficiently as possible.

**Integrated Hardware and Software**

iOS integrates with advanced hardware features such as a power efficient CPU, accelerated graphics, and wireless antennas. Hardware and software work together to deliver an optimized user experience that’s great for battery life.

**Intelligent App Management**

iOS apps have a life cycle that’s managed by the system. When a user finishes interacting with an app, the app is placed into a background state, where activity is throttled and the app may be suspended. Apps generating high CPU usage for extended periods of time while running in the background may be terminated by the system, if necessary.

**Network Operation Deferral**

APIs let you designate criteria that indicate when and how often a network operation should be deferred, how long it can be deferred, and under what circumstances. The system uses this information to defer the operation until an energy efficient time.

**Task Prioritization**

Tasks that affect the user, such as downloading and playing music, take priority over background and discretionary work. Quality of service class APIs allow you to assign priority levels to the work your app performs, giving you fine-grained control over task prioritization.

**Developer Tools**

Xcode and Instruments help you identify and address energy problems as you develop your app, rather than after those problems are encountered by users.

**Your Obligation as a Developer**

Even small inefficiencies in apps add up, significantly affecting battery life, performance, and responsiveness. As an app developer, you have an obligation to make sure your app runs as efficiently as possible. Use recommended APIs so the system can make smart decisions about how best to manage your app and the resources it uses. Whenever possible, batch and reduce network operations, and avoid unnecessary updates to the user interface. Power-intensive operations should be under the user’s control. If a user is playing a graphics-heavy game, for example, the user should not be surprised if the activity consumes power. Strive to make your app absolutely idle when it is not responding to user input.

By adhering to recommended guidelines, you can make big contributions to the overall energy efficiency of the platform and the satisfaction of your users.

There’s no single solution for conserving energy on a device. Numerous technologies and operations influence how energy is used:



**CPU.** The CPU is a major consumer of energy. Periods of high CPU use rapidly drain a user’s battery. Your app uses the CPU for almost everything it does, and it should do so wisely—by doing work only when necessary through batching, scheduling, and prioritizing.



**Device wake.** iOS devices rely on sleep for great battery life. Whenever a device wakes, there is a high overhead cost, as the screen and other resources must be powered up. Your app, especially when operating in the background, should be as idle as possible and avoid waking the device with push notifications or other activity unless absolutely necessary.



**Networking operations.** Most iOS apps perform networking operations. When networking occurs, components such as cellular radios and Wi-Fi power up and use energy. By batching and reducing transactions, compressing data, and appropriately handling errors, your app can make significant contributions to energy conservation.



**Graphics, animations, and video.** Every time your app’s content updates on screen, it uses energy to produce those pixels. Animations and videos can be especially taxing. Unexpected and unnecessary content updates also drain power. Your app should avoid updating content when its interface isn’t visible to the user. Also, follow recommended guidelines under Graphics and Animation in *iOS Human Interface Guidelines*.



**Location.** Many apps make location requests in order to log a user’s physical activity or provide environment-based alerts. Energy use increases with greater precision and longer location requests. Your app should reduce accuracy and duration of location activity whenever possible. Stop location requests when no longer needed.



**Motion** Continuous unwarranted requests for accelerometer, gyroscope, magnetometer, and other motion data waste energy. Request motion updates only when necessary, and stop updates when data is no longer needed.

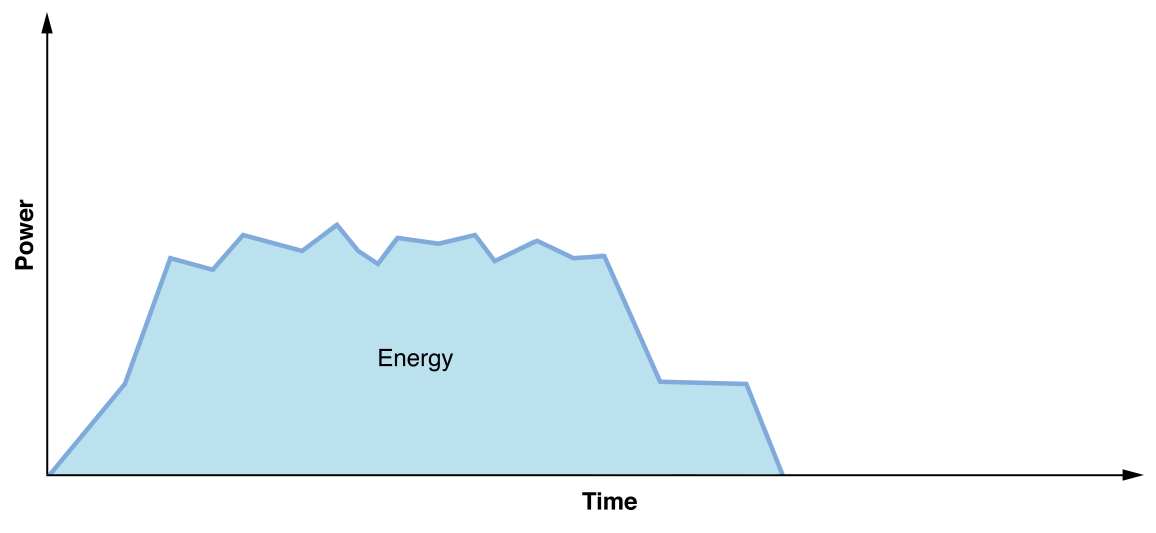


**Bluetooth.** High periods of Bluetooth activity can drain the battery of the iOS device and the Bluetooth device. Whenever possible, batch and buffer Bluetooth activity, and reduce polling for data.

**Energy and Power**

Energy and power are two separate but related concepts. Power is an instantaneous measurement (watts) of energy required at any given point in time (Figure 2-1). Energy is a measurement of power used (joules) over a period of time (watt hours). Energy is finite. It’s stored in the battery and dissipates over time as more power is required.

By being aware of energy and taking it into account while developing your app, you can proactively take measures that make your code more efficient. As more apps improve efficiency, users will have batteries that last much longer, in addition to cooler and quieter devices.

**Figure 2-1**Energy is power over time**CPU Usage and Power Draw**

CPU usage is expensive. As more CPU is used, more power draw occurs, more energy is used, and the device’s battery drains faster. Power draw varies based on the device, processor, resources, and so on, but Table 2-1 provides a rough comparison of varying CPU usage against an idle state.

The majority of techniques and recommendations throughout this document result in less usage of the CPU.

| Idle | 10x greater power draw over sleep |
| --- | --- |
| 1% CPU use | 10% greater power draw over idle |
| 10% CPU use | 2x power draw over idle |
| 100% CPU use | 10x power draw over idle |
| **Table 2-1**Example of idle vs. CPU power draw | |

**Fixed Cost and Dynamic Cost**

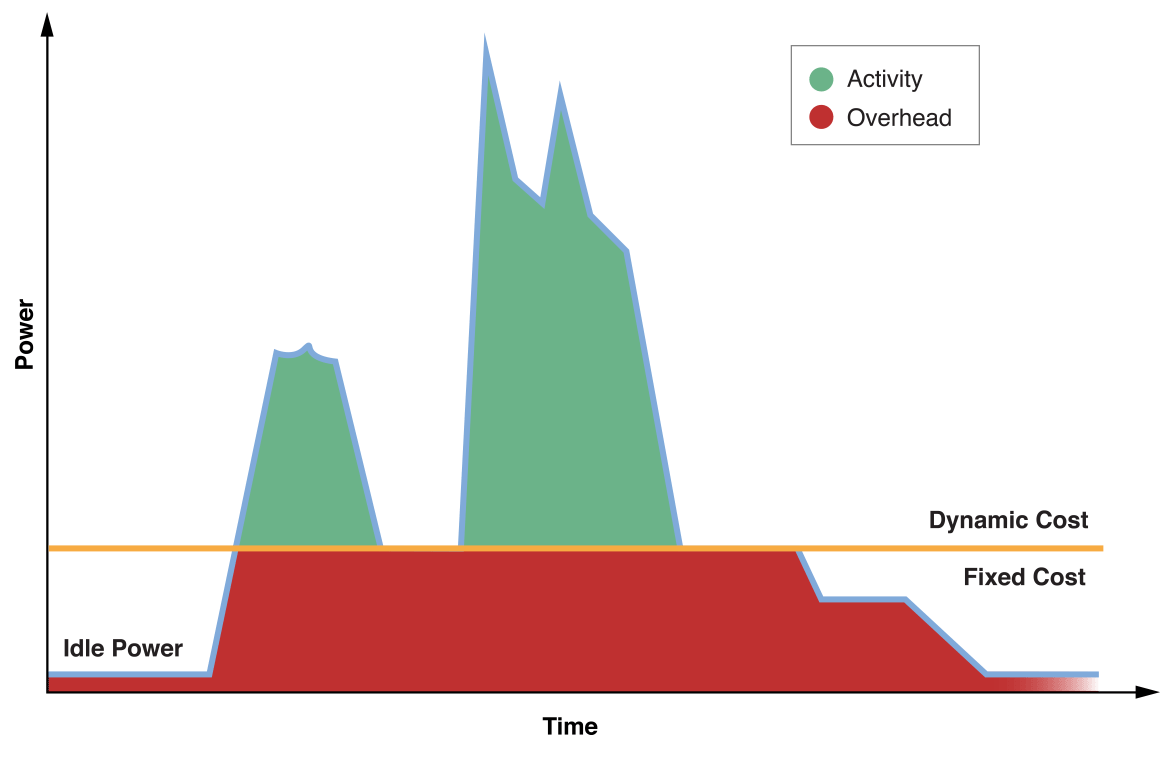
iOS is very good at getting a device into a low power state when it’s not being used. Even at the microsecond scale, such as between keystrokes, the system is able to power down resources that aren’t being used.

At idle, very little power is drawn and energy impact is low. When tasks are actively occurring, system resources are being used and those resources require energy. However, sporadic tasks can cause the device to enter an intermediate state—neither idle nor active—when the device isn’t doing anything. There may not be enough time during these intermediate states for the device to reach absolute idle before the next task executes. When this occurs, energy is wasted and the user’s battery drains faster.

Tasks your app performs have a *dynamic cost*—how much energy your app uses by doing actual work. They also have a *fixed cost*—how much energy is used by bringing the system and various resources up in order for your app to do work, and back down after that work is complete. When lots of sporadic work is occurring, there are dynamic costs and a significant fixed cost too, as resources may never get the chance to reach true idle between the sporadic tasks. This situation results in a lot of energy being used for a relatively small amount of actual work. See Figure 2-2.

IMPORTANT

Networking has a high fixed cost in iOS. Whenever networking occurs, cellular radios and Wi-Fi must power up. In anticipation of additional work, these resources remain running—and consume energy—for prolonged periods, even after your work is complete.

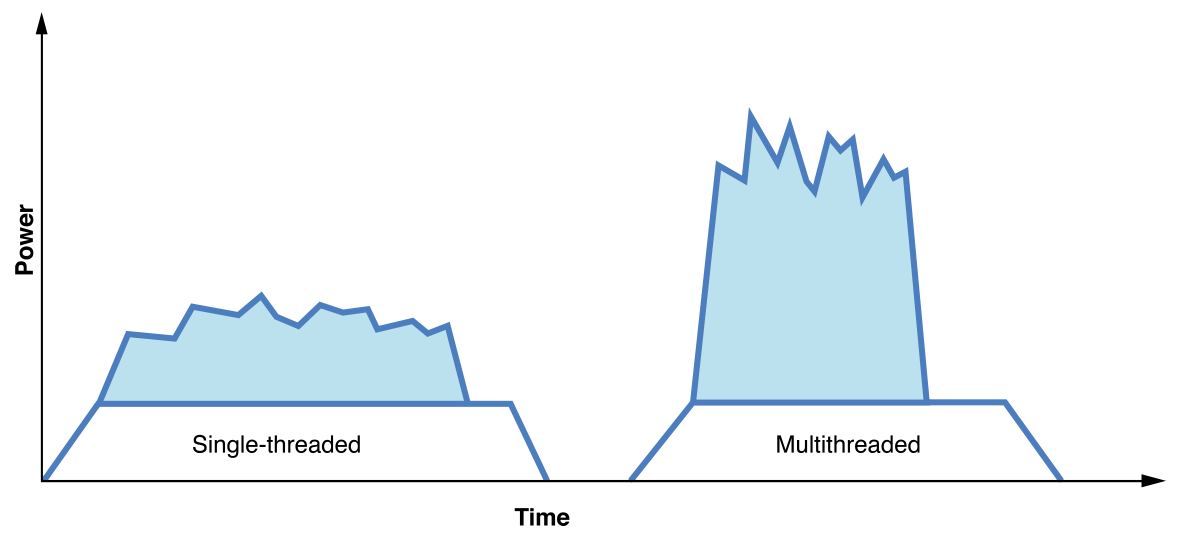
**Figure 2-2**Fixed vs. dynamic energy costs

**Trading Dynamic Cost for Fixed Cost**

Your app can avoid sporadic work by batching tasks and performing them less frequently. For example, instead of performing a series of sequential tasks on the same thread, distribute those same tasks simultaneously across multiple threads, as shown in Figure 2-3. Each time the CPU is accessed, memory, caches, buses, and so forth must be powered up. By batching activity, components can be powered up once and used over a shorter period of time.

This strategy incurs a greater up-front dynamic cost—more work is done at a given time, requiring more power. In exchange, you get a dramatic reduction in fixed cost, which results in tremendous energy savings over time. Your app draws more power, but it does so more efficiently and over less time. This lets the CPU get back to idle and other components to power down much more quickly.

As you develop your app, think holistically about its behavior, and try to reduce fixed costs wherever possible.

**Figure 2-3**Use multithreading to trade power for energy

When the user isn’t actively using your app, the system places it into a background state. The system may eventually suspend your app if it’s not performing important work, such as finishing a task the user initiated or running in a specially declared background execution mode.

IMPORTANT

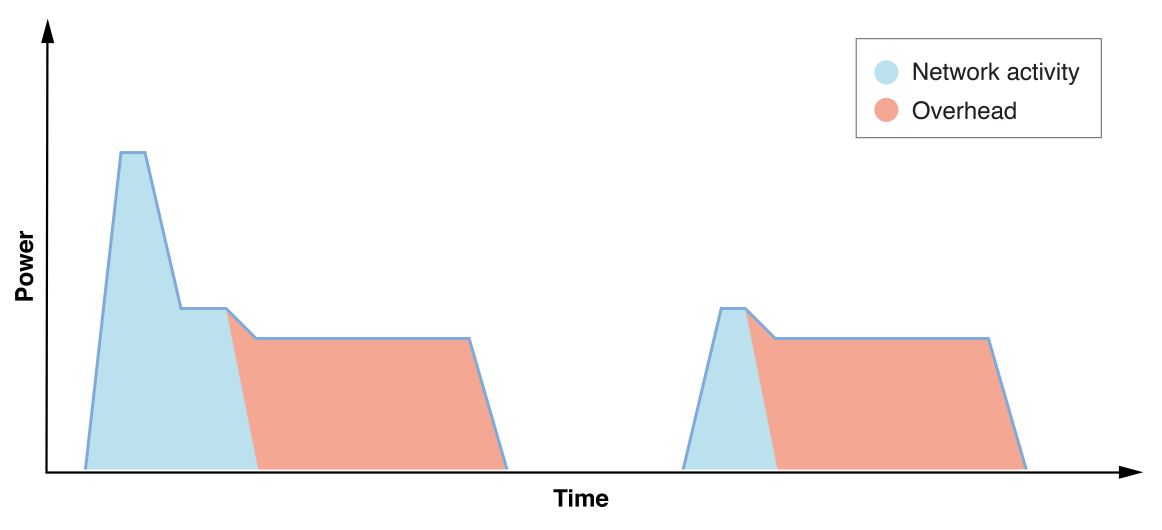
Your app shouldn’t wait to be suspended by the system. It should begin winding down activity immediately once notified that state has changed. When your app completes any remaining tasks, it should notify the system that background activity is complete. Failing to do so causes the app to remain active and draw energy unnecessarily.

You can use Energy organizer to view crash and energy reports that are generated from log information collected automatically from TestFlight users and with permission from users of the App Store versions of your apps.

Almost all iOS apps perform network operations of some kind, and it’s essential that networking be employed efficiently. This means eliminating overhead cost whenever possible by reducing and scheduling transactions and using efficient APIs.

**Device Networking Overhead**

Whenever your app performs network operations, there is substantial overhead cost involved. Networking hardware, such as cellular and Wi-Fi radios, are powered down by default to conserve battery. These resources must be powered up to perform activity. Then, they remain up for the duration of the activity and for an additional period of time in anticipation of more work. Sporadic network transactions result in high overhead and can quickly deplete the device’s battery. See Figure 8-1.

**Figure 8-1** Example of overhead due to recurring network activity

Networking Variable Effect on Energy

A variety of factors affect the amount of energy required to perform network operations on a device.

* Cellular network activity requires significantly more energy than activity over Wi-Fi.
* Poor or fluctuating signal conditions may result in slower or problematic transactions, which must be retried.
* Low network throughput (bandwidth) means radios have to stay on longer to perform transactions.
* Even geographic location and choice of cellular provider can impact energy consumption, as signal conditions and throughput can vary.

As a result, a device’s technical specs typically provide battery life estimates for a variety of scenarios. For example, the specs for iPhone 6s indicate Internet use of up to 10 hours on 3G and LTE, and 11 hours on Wi-Fi. App usage ultimately determines how much network traffic occurs. The more efficient the network use by apps, the longer the device battery life.

### Reduce Media Quality and Size

If your app uploads, downloads, or streams media content, lower quality and smaller sizes reduce the amount of data being sent and received. Some apps let the user specify the quality and size. For example, when emailing a photo, Mail lets the user send a scaled version of the image at small, medium, or large size. The smallest size is the most energy efficient.

### Compress Data

Use compression algorithms to compact data as much as possible before sending or receiving it.

### Avoid Redundant Transfers

Your app shouldn’t repeatedly download the same data.

### Cache Data

Use caching to store infrequently updated data locally. Redownload the data only when it has changed or the user requests it. The [NSURLCache](https://developer.apple.com/documentation/foundation/nsurlcache) and [NSURLSession](https://developer.apple.com/documentation/foundation/urlsession) APIs can be used to implement in-memory and on-disk caches for URL request data.

### Use Pausable and Resumable Transactions

Network conditions fluctuate, and signal loss can be a regular occurrence. Be prepared to resume interrupted transactions so the same content isn’t downloaded multiple times. In some cases, it makes sense to let users pause downloads and resume them later. For example, iOS lets users pause app downloads by tapping on a partially downloaded app icon.

The [NSURLSession](https://developer.apple.com/documentation/foundation/urlsession) API lets you implement pause and resume functionality, without implementing caching.

### Handle Errors

Don’t attempt to perform network operations when the network is unavailable.

## **Avoid Extraneous Graphics and Animations**

If your app uses only standard windows and controls, you probably don’t need to worry much about extraneous content updates, as the system APIs are designed to maximize energy efficiency. However, if you have custom windows and controls, be sure your drawing code performs efficiently. Your app shouldn’t refresh content unnecessarily, such as in obscured areas on screen, or through excessive use of animations.

Every time your app updates (or “draws”) content to screen, it requires the CPU, GPU, and screen to be active. Extraneous or inefficient drawing can pull system resources out of low-power states or prevent them from powering down altogether, resulting in significant energy use.

Follow these guidelines to optimize content refreshes:

* Reduce the number of views your app uses.
* Reduce the use of opacity, such as in views that exhibit a translucent blur. If you need to use opacity, avoid using it over content that changes frequently. Otherwise, energy cost is magnified, as both the background view and the translucent view must be updated whenever content changes.
* Eliminate drawing when your app or its content is not visible, such as when your app's content is obscured by other views, clipped, or offscreen.
* Use lower frame rates for animations whenever possible. For example, a high frame rate may make sense during game play, but a lower frame rate may be sufficient for a menu screen. Use a high frame rate only when the user experience calls for it.
* Use a consistent frame rate when performing an animation. For example, if your app displays 60 frames per second, maintain that frame rate throughout the lifetime of the animation.
* Avoid using multiple frame rates at once on screen. For example, don’t have a character in your game moving at 60 frames per second, while the clouds in the sky are moving at 30 frames per second. Use the same frame rate for both, even if it means raising one of the frame rates.
* Use recommended frameworks when developing games. These frameworks are optimized to provide great performance and optimal energy efficiency:
  + Use SpriteKit for 2D games. See *[SpriteKit Programming Guide](https://developer.apple.com/library/archive/documentation/GraphicsAnimation/Conceptual/SpriteKit_PG/Introduction/Introduction.html" \l "//apple_ref/doc/uid/TP40013043" \t "_self)* and *[SpriteKit Framework Reference](https://developer.apple.com/documentation/spritekit" \t "_self)*.
  + Use SceneKit for casual 3D games. See [*Scene Kit Framework Reference*](https://developer.apple.com/documentation/scenekit) and Scene Kit Functions Reference.
  + Use Metal for highly immersive games. See [*Metal Programming Guide*](https://developer.apple.com/library/archive/documentation/Miscellaneous/Conceptual/MetalProgrammingGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40014221), [*Metal Framework Reference*](https://developer.apple.com/documentation/metal), and Metal Functions Reference.