Article

Adapting iOS code to run in the macOS environment

Support modern iOS features that result in a better user experience when running on Apple silicon.

Overview

Most iOS apps require no modifications to run on Apple silicon. In macOS, the system maps iOS behaviors to appropriate macOS behaviors using infrastructure already in place to support Mac Catalyst. However, you might still want to update your code to account for differences between the iOS and macOS environments. For example, while the system matches touch events to mouse events, macOS supports only one such event at a time, whereas iOS supports multiple simultaneous touches. Making minor adjustments for environment differences, and providing alternatives for missing capabilities, ensures your app runs well on both platforms.

For general information about how to create iOS apps that run in macOS, see Running your iOS apps in macOS.

Check device properties and hardware availability dynamically

macOS provides versions of all iOS frameworks for your app to link with, but some features of those frameworks may be unavailable in macOS. In particular, frameworks that interact with iOS-specific hardware, such as Core Motion and ARKit, return errors when you try to use features that aren't available. Similarly, macOS doesn't support frameworks like HomeKit, and you receive errors when you use them.

Before you attempt to use any sensor or feature in your iOS app, always check whether it's available. The following list includes some of the more common APIs to call:

- Check the `isAccelerometerAvailable`, `isGyroAvailable`, and `isMagnetometerAvailable` properties of the `CMMotionManager` object to determine the availability of those sensors.
- Check for appropriate camera support in ARKit using the `isSupported` property of the appropriate `ARConfiguration` subclass.
- Identify the available cameras on the device using the `AVCaptureDevice.DiscoverySession` class. Don't assume a device contains a front- or rear-facing camera.
- Use Core Location to fetch location data as you do in iOS. Core Location doesn't require GPS hardware to get the current location. However, the locations you receive may not be as precise when GPS is unavailable.
- Call the `isHealthDataAvailable()` method of `HKHealthStore` to determine whether health-related data is available.
- Check the `authorizationStatus` property of `HMHomeManager` to see whether your app is able to access HomeKit.

Handle unknown device types gracefully

Some system APIs provide information about the underlying type of device, but such information is rarely the best way to make decisions. A decision based on a device type creates problems whenever your app runs on an unknown device, as might be the case for some lower-level system APIs. The better alternative is to use hard data that relates directly to the current task. For example, use size classes to determine whether to display an expanded or minimal version of your app's interface.

When you do need to know the device type, use higher-level APIs whenever possible. In macOS, the `UITraitCollection` and `UIDevice` classes report the following iOS idiom types instead of `UIUserInterfaceIdiom.mac`:

- For an app deployable on iPad, the idiom type is always `UIUserInterfaceIdiom.pad`.
- For an app deployable only on iPhone, and not on iPad, the idiom type is `UIUserInterfaceIdiom.phone`.

Any time you use device-specific information to make decisions, provide reasonable default behavior for unrecognized values. Although UIKit returns expected iOS values, other system APIs may not. In particular, lower-level APIs might return values that are appropriate for Mac devices, but may not be common on iOS devices. Run your app in macOS to see what kind of values you get for hardware-specific information, and use that information to determine a reasonable approach.

Support multiple windows using scenes

The macOS desktop makes it easy for an app to display multiple windows simultaneously, and many apps use that support to display different documents side-by-side. Most iOS apps display a single window, but the introduction of scenes in iOS 13 and iPadOS 13 provides an equivalent type of multiwindow support to iOS apps. Because supporting multiple windows requires you to manage your data differently, you must tell the system that your app supports multiple scenes. If you opt in, your app gains automatic support for multiple windows in macOS.

To enable multiple scenes in iOS, add the `UIApplicationSupportsMultipleScenes` key with the value of `true` to your app's `Info.plist` file. The presence of this key tells macOS to enable multiple window support for your app. For information about how to add scene support to your app, see Supporting multiple windows on iPad.

Make your windows resizable using iPad multitasking

In iPadOS, iPad multitasking lets a person display multiple apps onscreen at once, with each app occupying a specific portion of the available screen space. When they initiate this feature, or change the relative size of the apps, the system resizes the app's window accordingly. These resize events cause the app to lay out its content using the new window size. macOS leverages this same infrastructure to support resizable windows for your apps.

If your app already supports iPad multitasking, you don't need to do any extra work to enable resizable windows in macOS. If your app doesn't support iPad multitasking, add support by doing the following:

- Implement view controllers that support both compact and regular size classes. Size classes help you adapt your interface to particularly large or small window sizes.
- Lay out your views to fit the current window size. Optimize your Auto Layout constraints to improve resizing speeds.
- Support all device orientations.
- Don't include the `UIRequiresFullScreen` key in your app's `Info.plist` file.

Important

An app that you build to deploy only on iPhone always runs in a fixed-size window.

For more information about iPad multitasking, see Multitasking on iPad, Mac, and Apple Vision Pro.

Support native resolution of a Mac display in full-screen mode

If you choose to not support iPad multitasking in your app — for example, your iPad app is a game designed for full-screen mode — you can provide a pixel-perfect, edge-to-edge, full-screen experience on a Mac by including the `UILaunchToFullScreenByDefaultOnMac` and `UISupportsTrueScreenSizeOnMac` keys in your app's `Info.plist` file.

Don't make assumptions about system-provided UI

UIKit and other frameworks provide standard system alerts and view controllers for you to display from your apps. These interfaces include assorted pickers, such as file and color pickers, and view controllers that display standard interfaces, such as the activity view controller. At display time, macOS may present these interfaces differently than iOS does. For example, macOS uses its standard Open and Save panels for picking files.

When displaying a system interface from your app, use only the provided APIs to display and interact with the interface. The implementation of system interfaces may change from release to release, and from platform to platform. Don't make assumptions about how the system presents the interface, or about the size, position, or number of views in that interface.

Provide keyboard shortcuts for your app's features

Consider adding keyboard shortcuts to trigger common behaviors and features of your iOS app. Keyboard input is a common way to initiate actions in macOS, and it's also a useful way to trigger actions on iOS devices with a connected keyboard. In addition, keyboard input is a way to work around the lack of custom multifinger gestures in macOS. Use your keyboard code to trigger the same action as your custom gesture recognizer.

You can add keyboard support to any iOS app, not just apps you build with Mac Catalyst. To handle specific key combinations, override the appropriate press-event methods in your `UIResponder` object. For more information about how to add keyboard support to your app, see Handling key presses made on a physical keyboard.

Look for files in well-known directories

Be aware of differences between the iOS and macOS file systems, and take those differences into account when accessing files and directories:

- A person using iOS has limited access to files, but a person using macOS has access to all files and directories in the Finder or Terminal. They may directly examine the contents of any directory, including your app's data container.

- On a Mac, a person may move apps anywhere.

Don't assume that your app or data container is always at a specific location. When you write code to access the file system, always use Foundation APIs to get the URL for your app and data directories. The `url(for:in:appropriateFor:create:)` method of the default `FileManager` object returns a URL that's appropriate for the current platform.

Migrate away from deprecated technologies

macOS supports nearly all iOS frameworks, including many that are deprecated. However, deprecated frameworks don't receive active updates and bug fixes, and might not behave as expected in macOS. Therefore, migrate away from deprecated frameworks whenever possible and use modern replacements instead.

The following frameworks and features are deprecated with appropriate replacements:

- Address Book — Use Contacts instead.
- Address Book UI — Use Contacts UI instead.
- Assets Library — Use PhotoKit instead.
- OpenGLES — Use Metal instead.
- GLKit — Use MetalKit instead.
- Socket-based VoIP support — Use PushKit instead.

Xcode issues warnings when you build your code with deprecated APIs. The documentation and headers also include information about which frameworks and APIs are deprecated, and what the appropriate replacements are. Update your iOS app to use the designated replacements as soon as possible.

See Also

iOS apps on Mac

Running your iOS apps in macOS

Modernize the iOS apps you choose to run on a Mac with Apple silicon, or opt out of running on a Mac altogether.

{} Providing touch gesture equivalents using Touch Alternatives

Enable Touch Alternatives to provide keyboard, mouse, and trackpad equivalents to your iOS app when it runs on a Mac with Apple silicon.

{} Providing an edge-to-edge, full-screen experience in your iPad app running on a Mac

Take advantage of the true native resolution of a Mac display when running your iPad app in full-screen mode on a Mac.