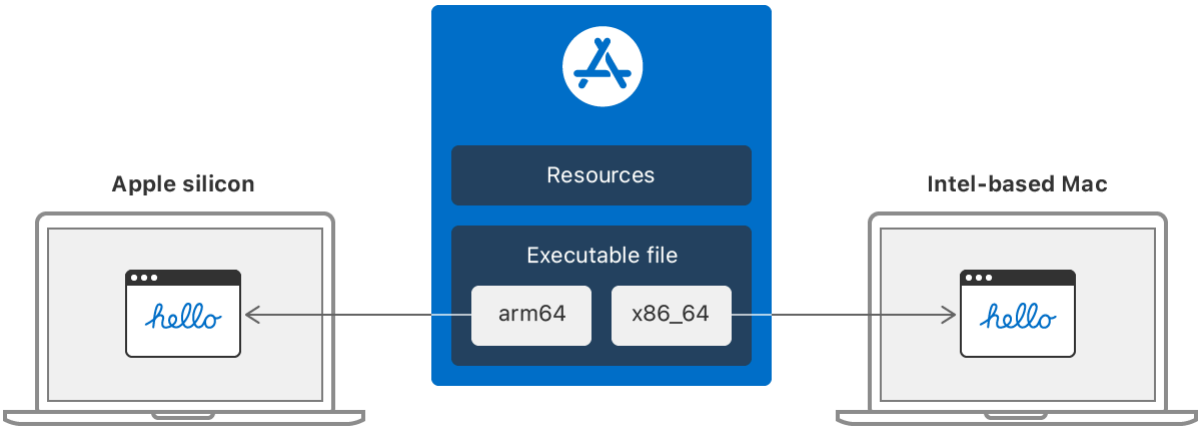
Porting your macOS apps to Apple silicon

Create a version of your macOS app that runs on both Apple silicon and Intel-based Mac computers.

Overview

Port your existing macOS app to Apple silicon by creating a universal binary and modifying your code to handle architectural differences. A universal binary looks no different than a regular app, but its executable file contains two versions of your compiled code. One version runs natively on Apple silicon, and the other runs natively on Intel-based Mac computers. At runtime, the system automatically chooses which version to run on the current platform.



To build a universal binary, you need Xcode 12 or a later version, which adds `arm64` to the standard list of build architectures for macOS binaries. When you open your project and do a clean build, Xcode creates a universal binary automatically if your project uses the standard architectures. If you use custom makefiles or build scripts, add the `arm64` architecture to your build system.

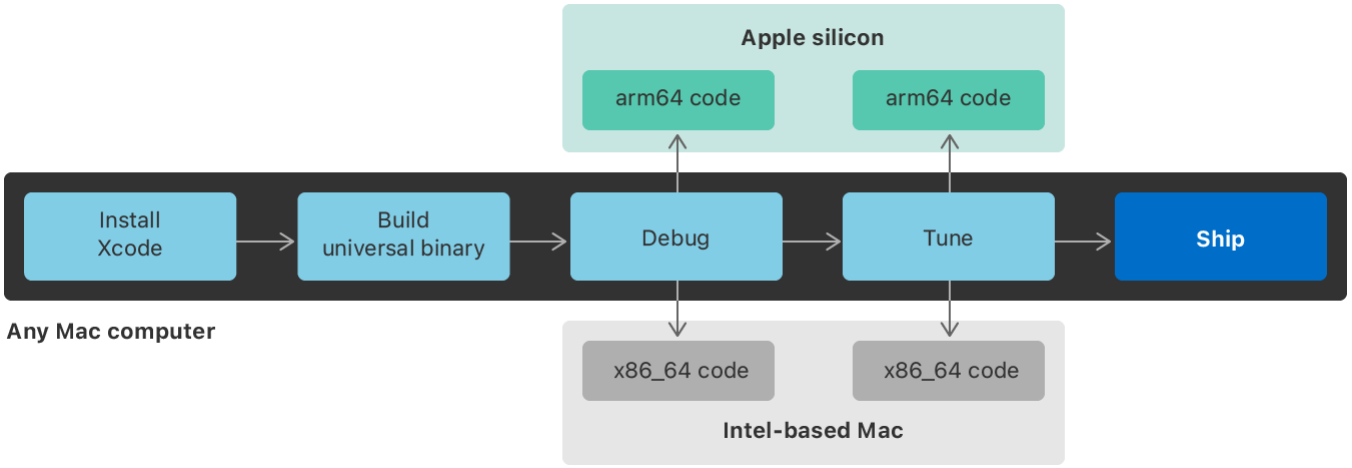
After you create a universal binary, test it on both architectures and determine whether you need to make additional changes. macOS frameworks shield apps from most architectural differences between platforms, but some differences may still require you to change your code. In addition, architectural differences may affect your app's performance and require further changes.

> **Note**
>
> On Apple silicon, apps built for the `x86_64` architecture run under the Rosetta translation environment. For more information, see About the Rosetta translation environment.

To learn how to build a universal binary, see Building a universal macOS binary.

Create a Porting Plan

Early in the porting process, identify the workflow you'll use to build and test your code. Xcode runs on all Mac computers, so build your code on either an Apple silicon or Intel-based Mac computer and do your initial testing there. However, always test, tune, and validate your code on both computer types to uncover issues specific to that architecture.



In addition to a workflow plan, identify potential areas to investigate during the porting process. The porting effort for `arm64` depends on how much you rely on hardware-specific features. If you rely mostly on Apple frameworks and technologies, your porting effort may be small. If you tuned your code specifically for the `x86_64` architecture and hardware capabilities, porting to `arm64` may require additional effort.

To start your investigation, make a note of any code that does the following:

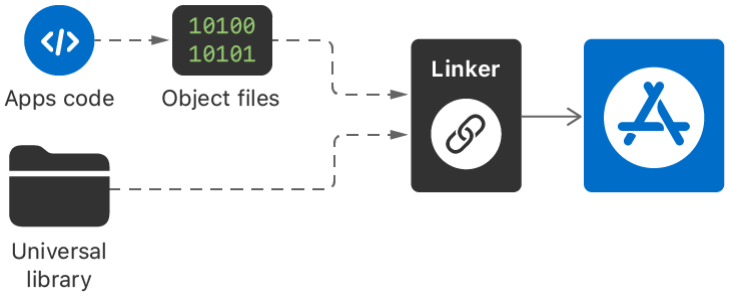
- Interacts with third-party libraries you don't own.
- Interacts with the kernel or hardware.
- Relies on specific GPU behaviors.
- Contains assembly instructions.
- Manages threads or optimizes your app's multithreaded behavior.
- Contains hardware-specific assumptions or performance optimizations.

The list above is not exhaustive, but it offers a starting point for your investigation. Hardware and architectural differences may introduce bugs or performance issues to your code on Apple silicon. Identifying potential problem areas early will save you time later.

Always have a well-defined test plan, ideally with a set of automated test suites you can run at build time. In addition to testing your code's correctness, gather metrics on your app's performance. Examine your app's memory usage, and measure how long it takes to execute specific tasks on both Apple silicon and Intel-based Mac computers. Use that information to identify additional areas to investigate.

Obtain Universal Versions of Linked Libraries

If your project depends on any third-party libraries, contact the original vendors and ask them to provide you with universal versions of those libraries. All code running in the same process must support the same architecture. You cannot produce a universal version of your binary without universal versions of all linked libraries. If one or more libraries is not universal, the linker reports errors.

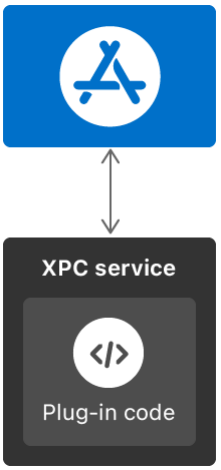


To learn how to create your own universal binaries, see Building a universal macOS binary.

Update Plug-Ins to Universal Binaries

A universal plug-in runs natively on any Mac computer. If your app supports a plug-in model, create universal versions of the plug-ins that you manage. If your company allows external developers to contribute plug-ins, encourage those developers to create universal versions of their plug-ins.

Universal plug-ins are essential if your app loads those plug-ins directly into its process space. Code running in the same process must support the same architecture. If your app attempts to load a plug-in with an incompatible architecture, the system reports an error at load time.



Plug-ins that run out-of-process using an XPC service may run using a different architecture than the app itself. To give your developers time to update their plug-ins, provide two non-universal XPC services—one to run `arm64` plug-ins and one to run `x86_64` plug-ins. A single XPC service can manage either native or translated plug-ins, but not both at the same time. When creating the services, give each one a unique bundle identifier so they may run simultaneously.

For information about how to communicate with out-of-process plug-ins using XPC, see XPC.

Address Architectural Differences

Apart from large-scale changes to the processor and graphics hardware, subtle architectural differences exist between Apple silicon and Intel-based Mac computers. During the porting process, audit your code to identify fixes for any potential architectural issues. For example, look for places where your code relies on specific hardware features or configurations.

The following list identifies several known architectural differences between Apple silicon and Intel-based Mac computers. Update code that relies on any of the following:

- Virtual memory-page sizes
- Cache line sizes
- Variadic functions
- Memory that is simultaneously writable and executable
- Just-in-time compilers
- Realtime threads
- Explicit thread priorities
- Hardware-specific details
- Assembly-language instructions or builtin intrinsics
- Vector unit instructions

- C++ ABI details

> **Note**
>
> Both Apple silicon and Intel-based Mac computers use the little-endian format for data, so you don't need to make endian conversions in your code. However, continue to minimize the need for endian conversions in custom data formats that you create.

For additional information about architectural differences, see Addressing architectural differences in your macOS code.

Update GPU-Specific Code

Metal on Apple silicon supports the features of both Intel-based Mac computers and iOS devices. If your app adopts Metal features that are found only on Intel-based Mac computers, consider also adopting the iOS-specific features in your `arm64` code. Adopting these features can lead to performance improvements for many apps.

If your app uses Metal, OpenGL, or OpenCL, be aware of the following differences:

- The GPU and CPU on Apple silicon share memory.
- OpenGL is deprecated, but is available on Apple silicon.
- OpenCL is deprecated, but is available on Apple silicon when targeting the GPU. The OpenCL CPU device is not available to arm64 apps.

For information about how to update your graphics code, see Porting your Metal code to Apple silicon.

Update Drivers, System Extensions, and Kernel Extensions

When porting code to macOS 11, be aware of the following requirements for code that interacts with the kernel:

- Implement hardware drivers using DriverKit. macOS 11 requires you to use a DriverKit extension when support for one is available. Most driver types now support DriverKit, and only a few still require the creation of a kernel extension.
- Kernel extensions must support the native architecture. Kernel extensions run in the kernel, and the kernel always runs as a native process. You cannot run kernel extensions using Rosetta translation.

- The installation and uninstallation of kernel extensions requires a reboot. When you install a kernel extension, the system doesn't load your extension until after a reboot.

For more information about kernel extension and driver changes, see Implementing drivers, system extensions, and kexts.

Migrate Away from Specific Technologies

macOS includes a few technologies that are currently deprecated or discouraged for active development. If your app uses one of the following technologies, migrate to an appropriate replacement as soon as possible:

- OpenGL—Use Metal instead.
- OpenCL—Use Metal instead.
- AddressBook—Use the Contacts framework instead.
- Carbon APIs—Migrate to AppKit, Foundation, and other modern APIs.
- IOKit kernel extensions—Migrate to DriverKit where appropriate; see DriverKit framework.

Apple silicon still provides support for the preceding technologies, and you may continue to use them in macOS 11. However, this support may be removed in a future version of macOS, so migration to newer technologies is recommended.

Debug and Test Your Code

Apple silicon supports all debugging and testing tools found on Intel-based Mac computers. Use the Xcode IDE to set and monitor breakpoints and monitor other aspects of your app's behavior. Use `lldb` from the command line to perform similar tasks outside of the Xcode interface.

For more information about how to debug and test your code, see Xcode.

Tune Your App's Performance

Apple silicon runs all performance tools found on Intel-based Mac computers. Use Instruments and other performance tools to gather different types of metrics for your app, including information about its memory usage, speed, energy usage, and more. You can also use command-line tools such as `leaks`, `heap`, `top`, `fs_usage`, `sc_usage`, `vm_stat`, `otool`, `sample`, `malloc_history`, and `vmmap` to identify potential performance issues.

Architectural differences between `arm64` and `x86_64` mean that techniques that work well on one system might not work well on the other. For example:

- Don’t assume a discrete GPU means better performance. The integrated GPU in Apple processors is optimized for high performance graphics tasks. See Porting your Metal code to Apple silicon.
- Don’t assume that all processor cores are equal. The processors on Apple silicon contain a mixture of performance cores (P-cores) and efficiency cores (E-cores), which execute tasks with different performance characteristics. Use Quality-of-Service (QoS) classes to help the system schedule your tasks on the right type of core.

During the porting process, measure your app’s performance on both Apple silicon and Intel-based Mac computers and investigate any discrepancies. Tasks that take longer to run on one platform may require additional tuning.

For specific tips on tuning universal binaries, see Tuning your code’s performance for Apple silicon.

Topics

Additional Porting Tips

- Addressing architectural differences in your macOS code

  Fix problems that stem from architectural differences between Apple silicon and Intel-based Mac computers.

- Porting your audio code to Apple silicon

  Eliminate issues in your audio-specific code when running on Apple silicon Mac computers.

- Porting just-in-time compilers to Apple silicon

  Update your just-in-time (JIT) compiler to work with the Hardened Runtime capability, and with Apple silicon.

See Also

Essentials

- Building a universal macOS binary

Create macOS apps and other executables that run natively on both Apple silicon and Intel-based Mac computers.