

[Apple silicon](#) / Building a universal macOS binary

Article

Building a universal macOS binary

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

Overview

Native apps run more efficiently than translated apps because the compiler is able to optimize your code for the target architecture. An app that supports only the x86_64 architecture must run under Rosetta translation on Apple silicon. A universal binary runs natively on both Apple silicon and Intel-based Mac computers, because it contains executable code for both architectures.

Turn all of your compiled code into universal binaries, not just apps. The following list includes the most common types of executables to turn into universal binaries. This list is not exhaustive, but you can use it as a starting point to assess your projects.

- Apps
- App extensions
- Plug-ins
- Custom frameworks
- Static libraries
- Dynamic libraries
- Build tools
- Command-line tools
- Daemons and launch agents
- DriverKit extensions
- Kernel extensions

Note

You can build a universal binary on either an Apple silicon or Intel-based Mac computer, but you cannot debug the arm64 slice of your binary on an Intel-based Mac computer. It's possible to debug both slices of a universal binary on Apple silicon, but you must run the x86_64 slice under Rosetta translation.

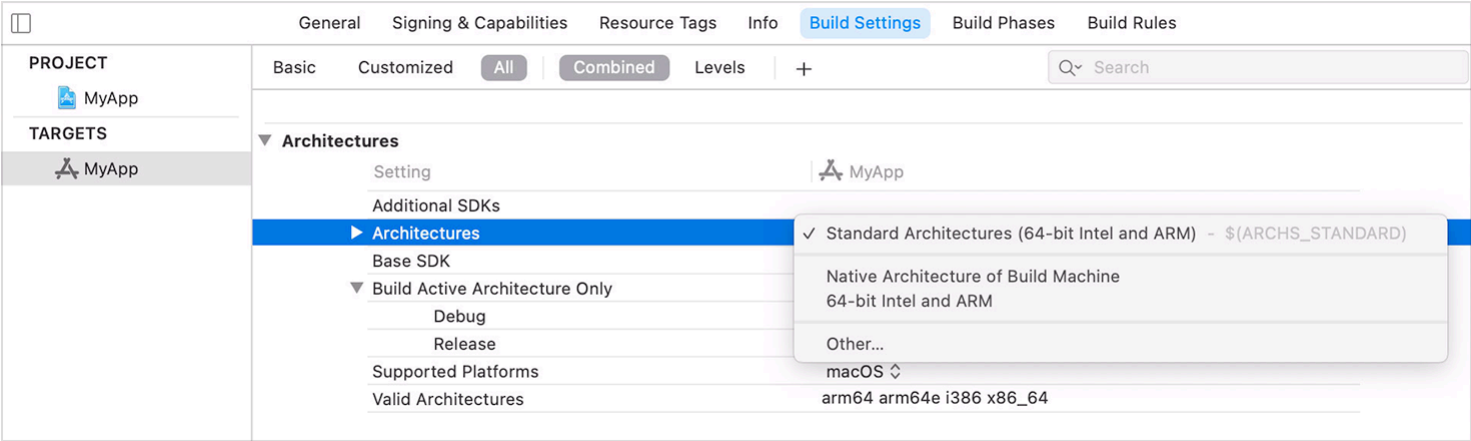
Download and Install Xcode on Your Mac Computer

Xcode 12.2 and later is a requirement for building universal binaries. Earlier versions of Xcode don't contain the support needed to build and test universal versions of your macOS code.

Download the latest public version of Xcode from the App Store. Download prerelease versions of Xcode from developer.apple.com.

Update the Architecture List in Your Xcode Projects

Xcode 12.2 and later automatically adds the arm64 architecture to the list of standard architectures for all macOS binaries, including apps and libraries. During the debugging and testing process, Xcode builds only for the current system architecture by default. However, it automatically builds a universal binary for the release version of your code.



If you customized the Architectures build setting in your Xcode project, remove your customizations and use the Standard Architectures setting instead.

For additional information about Xcode build settings and how to configure them, see [Configure build settings](#).

Update the Architecture List of Custom Makefiles

If you build your project using custom scripts or makefiles, add the `arm64` architecture to the appropriate environment variables. Xcode uses the `ARCHS` environment variable to define the current build architectures. Other build systems may use different environment variables, but with similar purposes. After adding the variable to the appropriate environment variable, build your code and verify that the compiler creates an `arm64` variant of your code. To create a universal binary for your project, merge the resulting executable files into a single executable binary using the `lipo` tool.

For makefiles you create outside of Xcode, pass the appropriate architecture values to the compiler using the `-target` option. The following example shows a makefile that compiles a single-source file twice—once for each architecture. It then creates a universal binary by merging the resulting executable files together with the `lipo` tool.

```
x86_app: main.c
    $(CC) main.c -o x86_app -target x86_64-apple-macos10.12
arm_app: main.c
    $(CC) main.c -o arm_app -target arm64-apple-macos11
universal_app: x86_app arm_app
    lipo -create -output universal_app x86_app arm_app
```

Wrap Platform-Specific Code with Conditional Compilation Macros

When writing code for a specific platform or processor type, isolate that code using the appropriate conditional compilation statements. For C-based code, the system defines a set of macros for you to use in `/usr/include/TargetConditionals.h`. The Swift language also supports conditional compilation using conditional compilation blocks.

Note

If you share code across multiple platforms, you can also use compiler-specific macros such as `__arm64__` or `__aarch64__` in conditional compilation statements. These macros don't require you to include the `TargetConditionals.h` header file. For a list of compiler macros, see the compiler documentation.

To distinguish code meant for a specific type of processor, add conditional compilation statements that target the appropriate architecture. Universal macOS apps support the `arm64` and `x86_64` architectures, and the following example shows how to write conditional code for those architectures in Swift:

```
// Swift example
#if arch(arm64)
    // Code meant for the arm64 architecture here.
#elseif arch(x86_64)
    // Code meant for the x86_64 architecture here.
#endif
```

For Objective-C, use the following code:

```
// Objective-C example
#include "TargetConditionals.h"
#if TARGET_CPU_ARM64
    // Code meant for the arm64 architecture here.
#elif TARGET_CPU_X86_64
    // Code meant for the x86_64 architecture here.
#endif
```

If you share code between an iOS and macOS app, don't assume that code intended for the arm64 architecture runs only on an iOS device. That code also runs in your macOS app on an Apple silicon. To isolate code specifically for macOS or iOS, use the conditional compilation statements shown in the following example. You can also combine both platform- and architecture-specific conditional compilation to further refine your code.

```
//Swift example
#if os(macOS)
    // Put CPU-independent macOS code here.
    #if arch(arm64)
        // Put 64-bit arm64 Mac code here.
    #elseif arch(x86_64)
        // Put 64-bit x86_64 Mac code here.
    #endif
#elseif targetEnvironment(macCatalyst)
    // Put Mac Catalyst-specific code here.
#elseif os(iOS)
    // Put iOS-specific code here.
#endif
```

In Objective-C, implement the same code as shown below.

```
// Objective-C example
#include "TargetConditionals.h"
#if TARGET_OS_OSX
    // Put CPU-independent macOS code here.
    #if TARGET_CPU_ARM64
        // Put 64-bit Apple silicon macOS code here.
    #elif TARGET_CPU_X86_64
        // Put 64-bit Intel macOS code here.
    #endif
#elif TARGET_OS_MACCATALYST
    // Put Mac Catalyst-specific code here.
#elif TARGET_OS_IOS
    // Put iOS-specific code here.
#endif
```

For the complete list of conditional compilation macros, see the `/usr/include/TargetConditionals.h` header file in the appropriate SDK. For a list of Swift compilation conditions and arguments, see [Conditional Compilation Block](#) in [The Swift Programming Language](#).

Build Your Target

When you build a debuggable version of your code, Xcode builds only for the current architecture by default. Building for one architecture saves time when you are trying to debug your code and fix problems quickly.

You can create a universal binary with debug symbols on any Mac computer by changing the Build Active Architecture Only build setting of your project. Although you can create this binary on an Intel-based Mac computer, you can't run or debug the `arm64` slice of it. Only a Mac with Apple silicon is capable of running and debugging both slices of your binary; use Rosetta translation to run and debug the `x86_64` slice.

Determine Whether Your Binary Is Universal

To users, a universal binary looks no different than a binary built for a single architecture. When you build a universal binary, Xcode compiles your source files twice—once for each architecture. After linking the binaries for each architecture, Xcode then merges the architecture-specific binaries into a single executable file using the `lipo` tool. If you build the source files yourself, you must call `lipo` as part of your build scripts to merge your architecture-specific binaries into a single universal binary.

To see the architectures present in a built executable file, run the `lipo` or `file` command-line tools. When running either tool, specify the path to the actual executable file, not to any

intermediate directories such as the app bundle. For example, the executable file of a macOS app is in the `Contents/MacOS/` directory of its bundle. When running the `lipo` tool, include the `-archs` parameter to see the architectures. The following example shows how to use `lipo` to view the list of architectures for the Mail app in macOS, and the results when Mail is a universal binary.

```
% lipo -archs /System/Applications/Mail.app/Contents/MacOS/Mail
x86_64 arm64
```

To obtain more information about each architecture, pass the `-detailed_info` argument to `lipo`.

For information about how to determine whether your app is running as a translated binary, see [Determine Whether Your App Is Running as a Translated Binary](#).

Specify the Launch Behavior of Your App

For universal binaries, the system prefers to execute the slice that is native to the current platform. On an Intel-based Mac computer, the system always executes the `x86_64` slice of the binary. On Apple silicon, the system prefers to execute the `arm64` slice when one is present. Users can force the system to run the app under Rosetta translation by enabling the appropriate option from the app's Get Info window in the Finder.

If you never want users to run your app under Rosetta translation, add the `LSRequiresNativeExecution` key to your app's `Info.plist` file. When that key is present and set to YES, the system prevents your app from running under translation. In addition, the system removes the Rosetta translation option from your app's Get Info window. Don't include this key until you verify that your app runs correctly on both Apple silicon and Intel-based Mac computers.

If you want to prioritize one architecture, without preventing users from running your app under translation, add the `LSArchitecturePriority` key to your app's `Info.plist` file. The value of this key is an ordered array of strings, which define the priority order for selecting an architecture.

Note

If an app doesn't contain an executable binary, the system may run it under Rosetta translation as a precautionary measure to prevent potential runtime issues. For example, the system runs script-only apps under Rosetta translation. If you verified that your app runs correctly on both Apple silicon and Intel-based Mac computers, add the `LSArchitecturePriority` key to your app's `Info.plist` file and list the `arm64` architecture first.

See Also

Essentials

☰ 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.