


An Apple Library Primer

 This thread has been locked by a moderator.



 2.4k

Apple's library technology has a long and glorious history, dating all the way back to the origins of Unix. This does, however, mean that it can be a bit confusing to newcomers. This is my attempt to clarify some terminology.

If you have any questions or comments about this, start a new thread and tag it with *Linker* so that I see it.

Share and Enjoy

—
Quinn “The Eskimo!” @ Developer Technical Support @ Apple
`let myEmail = "eskimo" + "1" + "@" + "apple.com"`

An Apple Library Primer

Apple's tools support two related concepts:

- **Platform** — This is the platform itself; macOS, iOS, iOS Simulator, and Mac Catalyst are all platforms.
- **Architecture** — This is a specific CPU architecture used by a platform. `arm64` and `x86_64` are both architectures.

A given architecture might be used by multiple platforms. The most obvious example of this `arm64`, which is used by all of the platforms listed above.

Code built for one platform will not work on another platform, even if both platforms use the same architecture.

Code is usually packaged in either a Mach-O file or a static library. **Mach-O** is used for executables, dynamic libraries, bundles, and object files. These can have a variety of different extensions; the only constant is that `.o` is always used for a Mach-O containing an object file. Use `otool` and `nm` to examine a Mach-O file. Use `vtool` to quickly determine the platform for which it was built. Use `size` to get a summary of its size. Use `dyld_info` to get more details about a dynamic library.

IMPORTANT All the tools mentioned here are documented in man pages; for information on how to access that documentation, see [Reading UNIX Manual Pages](#).

The term **Mach-O image** refers to a Mach-O that can be loaded and executed without further processing. That includes executables, dynamic libraries, and bundles, but not object files.

A **dynamic library** has the extension `.dylib`. You may also see this called a shared library.

A **framework** is a bundle structure with the `.framework` extension that has both compile-time and run-time roles:

- At compile time, the framework combines the library's headers and its stub library (stub libraries are explained below).
- At run time, the framework combines the library's code, as a Mach-O dynamic library, and its associated resources.

The exact structure of a framework varies by platform. For the details, see [Placing Content in a Bundle](#).

macOS supports both frameworks and standalone dynamic libraries. Other Apple platforms support frameworks but not standalone dynamic libraries.

Historically these two roles were combined, that is, the framework included the headers, the dynamic library, and its resources. These days Apple ships different frameworks for each role. That is, the macOS SDK includes the compile-time framework and macOS itself includes the run-time one. Most third-party frameworks continue to combine these roles.

A **static library** is an archive of one or more object files. It has the extension `.a`. Use `ar`, `libtool`, and `ranlib` to inspect and manipulate these archives.

The **static linker**, or just the **linker**, runs at build time. It combines various inputs into a single output. Typically these inputs are object files, static libraries, dynamic libraries, and various configuration items. The output is most commonly a Mach-O image, although it's also possible to output an object file. The linker may also output metadata, such as a link map.

The linker has seen three major implementations:

- `ld` — This dates from the dawn of Mac OS X.
- `ld64` — This was a rewrite started in the 2005 timeframe. Eventually it replaced `ld` completely. If you type `ld`, you get `ld64`.
- `ld_prime` — This was introduced with Xcode 15. This isn't a separate tool. Rather, `ld` now supports the `-ld_classic` and `-ld_new` options to select a specific implementation.

Note During the Xcode 15 beta cycle these options were `-ld64` and `-ld_prime`. I continue to use those names because the definition of *new* changes over time (some of us still think of `ld64` as the new linker ;-).

The **dynamic linker** loads Mach-O images at runtime. It's path is `/usr/lib/dyld`, so it's often referred to as `dyld`, `dyld`, or `DYLD`. Personally I pronounced that *dee-lid*, but some folks say *di-lid* and others say *dee-why-el-dee*.

The dynamic linker has seen 4 major revisions. See WWDC 2017 Session 413 (referenced below) for a discussion of versions 1 through 3. Version 4 is basically a merging of versions 2 and 3.

The `dyld` [man page](#) is chock-full of useful info, including a discussion of how it finds images at runtime.

One of the most common points of confusion with dynamic linker is the way that the dynamic linker identifies dynamic libraries. There are two standard approaches to this, as described in [Dynamic Library Identification](#).

There is no such thing as a **static framework**. Well, you might hear this term used by non-Apple people, but it's not something that Apple has ever supported. DTS spends a lot of time explaining this to folks who are having mysterious build problems.

Xcode 15 introduced the concept of a **mergeable library**. This a dynamic library with extra metadata that allows the linker to embed it into the output Mach-O image, much like a static library. Mergeable libraries have many benefits. For all the backstory, see WWDC 2023 Session 10268 [Meet mergeable libraries](#). For instructions on how to set this up, see [Configuring your project to use mergeable libraries](#).

A **universal binary** is a file that contains multiple architectures for the same platform. Universal binaries always use the **universal binary format**. Use the `file` command to learn what architectures are within a universal binary. Use the `lipo` command to manipulate universal binaries.

A universal binary's architectures are either all in Mach-O format or all in the static library archive format. The latter is called a **universal static library**.

A universal binary has the same extension as its non-universal equivalent. That means a `.a` file might be a static library or a universal static library.

Most tools work on a single architecture within a universal binary. They default to the architecture of the current machine. To override this, pass the architecture in using a command-line option, typically `-arch` or `--arch`.

An **XCFramework** is a single document package that includes libraries for any combination of platforms and architectures. It has the extension `.xcframework`. An XCFramework holds either a framework, a dynamic library, or a static library. All the elements must be the same type. Use `xcodebuild` to create an XCFramework. For specific instructions, see [Xcode Help > Distribute binary frameworks > Create an XCFramework](#).

Historically there was no need to code sign libraries in SDKs. If you shipped an SDK to another developer, they were responsible for re-signing all the code as part of their distribution process. Xcode 15 changes this. You should sign your SDK so that a developer using it can verify this dependency. For more details, see WWDC 2023 Session 10061 [Verify app dependencies with digital signatures](#) and [Verifying the origin of your XCFrameworks](#).

A **stub library** is a compact description of the contents of a dynamic library. It has the extension `.tbd`, which stands for *text-based description* (TBD). Apple's SDKs include stub libraries to minimise their size; for the backstory, read [this post](#). Stub libraries currently use YAML format, a fact that's relevant when you try to [interpret linker errors](#). Use the `tapi` tool to create and manipulate these files. In this context *TAPI* stands for a *text-based API*, an alternative name for TBD. Oh, and on the subject of `tapi`, I'd be remiss if I didn't mention `tapi-analyze`!

Mach-O uses a **two-level namespace**. When a Mach-O image imports a symbol, it references the symbol name *and* the library where it expects to find that symbol. This improves both performance and reliability but it precludes certain techniques that might work on other platforms. For example, you can't define a function called `printf` and expect it to 'see' calls from other dynamic libraries because those libraries import the version of `printf` from `libSystem`.

To help folks who rely on techniques like this, macOS supports a **flat namespace** compatibility mode. This has numerous sharp edges — for an example, see the posts on [this thread](#) — and it's best to avoid it where you can. If you're enabling the flat namespace as part of a developer tool, search the 'net for *dyld interpose* to learn about an alternative technique.

WARNING Dynamic linker interposing is not documented as API. While it's a useful technique for developer tools, do not use it in products you ship to end users.

Apple platforms use **DWARF**. When you compile a file, the compiler puts the debug info into the resulting object file. When you link a set of object files into a executable, dynamic library, or bundle for distribution, the linker does not include this debug info. Rather, debug info is stored in a separate **debug symbols** document package. This has the extension `.dSYM` and is created using `dysymutil`. Use `symbols` to learn about the symbols in a file. Use `dwarfdump` to get detailed information about DWARF debug info. Use `atos` to map an address to its corresponding symbol name.

Over the years there have been some *really* good talks about linking and libraries at WWDC, including:

- WWDC 2022 Session 110362 [Link fast: Improve build and launch times](#)
- WWDC 2022 Session 110370 [Debug Swift debugging with LLDB](#)
- WWDC 2021 Session 10211 [Symbolication: Beyond the basics](#)
- WWDC 2019 Session 416 [Binary Frameworks in Swift](#) — Despite the name, this covers XCFrameworks in depth.
- WWDC 2018 Session 415 [Behind the Scenes of the Xcode Build Process](#)
- WWDC 2017 Session 413 [App Startup Time: Past, Present, and Future](#)
- WWDC 2016 Session 406 [Optimizing App Startup Time](#)

Note The older talks are no longer available from Apple, but you may be able to find transcripts out there on the 'net.

Historically Apple published a document, *Mac OS X ABI Mach-O File Format Reference*, or some variant thereof, that acted as the definitive reference to the Mach-O file format. This document is no longer available from Apple. If you're doing serious work with Mach-O, I recommend that you find an old copy. It's definitely out of date, but there's no better place to get a high-level introduction to the concepts. The [Mach-O Wikipedia page](#) has a link to an archived version of the document.

For the most up-to-date information about Mach-O, see the declarations and doc comments in `<mach-o/loader.h>`.

Revision History

- **2023-09-20** Added a link to [Dynamic Library Identification](#). Updated the names for the static linker implementations (`-ld_prime` is no more!). Removed the *beta* epithet from Xcode 15.
- **2023-06-13** Defined the term *Mach-O image*. Added sections for both the static and dynamic linkers. Described the two big new features in Xcode 15: mergeable libraries and dependency verification.
- **2023-06-01** Add a reference to `tapi-analyze`.
- **2023-05-29** Added a discussion of the two-level namespace.
- **2023-04-27** Added a mention of the `size` tool.
- **2023-01-23** Explained the compile-time and run-time roles of a framework. Made other minor editorial changes.
- **2022-11-17** Added an explanation of TAPI.
- **2022-10-12** Added links to Mach-O documentation.
- **2022-09-29** Added info about `.dSYM` files. Added a few more links to WWDC sessions.
- **2022-09-21** First posted.


Linker

Reply

Posted 1 year ago by  eskimo 

Add a Comment

This site contains user submitted content, comments and opinions and is for informational purposes only. Apple disclaims any and all liability for the acts, omissions and conduct of any third parties in connection with or related to your use of the site. All postings and use of the content on this site are subject to the [Apple Developer Forums Participation Agreement](#).

 > Developer > Forums

Platforms	Topics & Technologies	Resources	Programs
iOS	Accessibility	Documentation	Apple Developer Program
iPadOS	Accessories	Curriculum	Apple Developer Enterprise Program
macOS	App Extensions	Downloads	App Store Small Business Program
tvOS	App Store	Forums	MFI Program
watchOS	Audio & Video	Videos	News Partner Program
visionOS	Augmented Reality		Video Partner Program
	Business	Support	Security Bounty Program
Tools	Design	Support Articles	Security Research Device Program
Swift	Distribution	Contact Us	
SwiftUI	Education	Bug Reporting	
SF Symbols	Fonts	System Status	Events
Swift Playgrounds	Games		App Accelerators
TestFlight	Health & Fitness	Account	App Store Awards
Xcode	In-App Purchase	Apple Developer	Apple Design Awards
Xcode Cloud	Localization	App Store Connect	Apple Developer Academies
	Maps & Location	Certificates, IDs, & Profiles	Entrepreneur Camp
	Machine Learning	Feedback Assistant	Tech Talks
	Security		WWDC
	Safari & Web		

To view the latest developer news, visit

[News and Updates](#)