An Apple Library Primer

This thread has been locked by a moderator.

89

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 a Mach-O was built. 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.

A **framework** is a Mach-O dynamic library embedded in a bundle structure. It always has the extension `.framework`. The exact structure varies by platform. For the details, see Placing Content in a Bundle.

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.

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.

A **universal binary** is a file that contains multiple architectures for the same platfrom. 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`.

Apple recently introduced the **XCFramework** format, a single document package that includes libraries for any combination of platfoms 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.

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*. 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.

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 `dsymutil`. 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.

Revision History

- **2022-09-29** Added info about `.dSYM` files. Added a few more links to WWDC sessions.
- **2022-09-21** First posted.

Linker

Reply

Posted 3 weeks ago by eskimo

Add a Comment