Building Mach-O Files

To create programs, developers convert source code to object files. The object files are then packaged into executable code or static libraries. OS X includes tools to transform source code into a running application or a shared library that can be used by one or more applications.

This article loosely describes how Mac apps are built, and discusses, in depth, the types of programs you can build. It describes the tools involved in the Mach-O file build process, explains the types of Mach-O files you can build, and talks about modules, which are the smallest linkable unit of code and data in the OS X runtime environment. It also describes static archive libraries, which are files that package a set of modules.

The Tools—Building and Running Mach-O Files

To perform the work of actually loading and binding a program at runtime, the kernel uses the *dynamic linker* (a specially marked dynamic shared library located at `/usr/lib/dyld`). The kernel loads the program and the dynamic linker into a new process and executes them.

Throughout this document, the following tools are discussed abstractly:

- A *compiler* is a tool that translates from source code written in a high-level language into intermediate object files that contain machine binary code and data. Unless otherwise specified, this book considers a machine-language assembler to be a compiler.
- A *static linker* is a tool that combines intermediate object files into final products (see The Products—Types of Mach-O Files You Can Build).

The Xcode Tools CD contains several command-line tools (which this document refers to collectively as the *standard tools*) for building and analyzing your application, including compilers and `ld`, the standard static linker. Whether you use the Xcode application, the standard command-line tools, or a third-party tool set to develop your application, understanding the role of each of the following tools can enhance your understanding of the Mach-O runtime architecture and facilitate communication about these topics with other OS X developers. The standard tools include the following:

- The compiler driver, `/usr/bin/gcc`, contains support for compiling, assembling, and linking modules of source code from the C, C++, and Objective-C languages. The compiler driver calls several other tools that implement the actual compiling, assembling, and static linking functionality. The actual compiler tools for each language dialect are normally hidden from view by the compiler driver; their role is to transform input source code into assembly language for input to the assembler.
- The C++ compiler driver, `/usr/bin/c++`, is like `/usr/bin/cc` but automatically links C++ runtime functions into the output file (to support exceptions, runtime type information and other advanced language features).
- The assembler, `/usr/bin/as`, creates intermediate object files from assembly language code. It is primarily used by the compiler driver, which feeds it the assembly language source generated by the actual compiler.
- The static linker, `/usr/bin/ld`, is used by the compiler driver (and as a standalone tool) to combine Mach-O executable files. You can use the static linker to bind programs either statically or dynamically. Statically bound programs are complete systems in and of themselves; they cannot make calls, other than system calls, to frameworks or shared libraries. In OS X, kernel extensions are statically bound, while all other program types are dynamically bound, even traditional UNIX and BSD command-line tools. All calls to the OS X kernel by programs outside the kernel are made through shared libraries, and only dynamically bound programs can access shared libraries.
- The library creation tool, `/usr/bin/libtool`, creates either static archive libraries or dynamic shared libraries, depending on the parameters given. `libtool` supersedes an older tool called `ranlib`, which

was used in conjunction with the `ar` tool to create static libraries. When building shared libraries, `libtool` calls the static linker (`ld`).

> **Note:** There is also a GNU tool named `libtool`, which allows portable source code to build libraries on various UNIX systems. Don't confuse it with OS X `libtool`; while they serve similar purposes, they are not related and they do not accept the same parameters.

Tools for analyzing Mach-O files include the following:

- The `/usr/bin/lipo` tool allows you to create and analyze binaries that contain images for more than one architecture. An example of such a binary is a *universal binary*. Universal binaries can be used in PowerPC-based and Intel-based Macintosh computers. Another example is a *PPC/PPC64 binary*, which can be used in 32-bit PowerPC-based and 64-bit PowerPC-based Macintosh computers.
- The file-type displaying tool, `/usr/bin/file`, shows the type of a file. For multi-architecture files, it shows the type of each of the images that make up the archive.
- The object-file displaying tool, `/usr/bin/otool`, lists the contents of specific sections and segments within a Mach-O file. It includes symbolic disassemblers for each supported architecture and it knows how to format the contents of many common section types.
- The page-analysis tool, `/usr/bin/pagestuff`, displays information on each logical page that compose the image, including the names of the sections and symbols contained in each page. This tool doesn't work on binaries containing images for more than one architecture.
- The symbol table display tool, `/usr/bin/nm`, allows you to view the contents of an object file's symbol table.

The Products—Types of Mach-O Files You Can Build

In OS X, a typical application executes code that originates from many types of files. The main executable file usually contains the core logic of the program, including the entry point `main` function. The primary functionality of a program is usually implemented in the main executable file's code. See Executing Mach-O Files for details. Other files that contain executable code include:

- *Intermediate object files.* These files are not final products; they are the basic building blocks of larger object files. Usually, a compiler creates one intermediate object file on output for the code and data generated from each input source code file. You can then use the static linker to combine the object files into dynamic linkers. Integrated development environments such as Xcode usually hide this level of detail.
- *Dynamic shared libraries.* These are files that contain modules of reusable executable code that your application references dynamically and that are loaded by the dynamic linker when the application is launched. Shared libraries are typically used to store large amounts of code that are usable by many applications. See Using Shared Libraries and Frameworks in Loading Code at Runtime for more information.
- *Frameworks.* These are directories that contain shared libraries and associated resources, such as graphics files, developer documentation, and programming interfaces. See Using Shared Libraries and Frameworks in Loading Code at Runtime for more information.
- *Umbrella frameworks.* These are special types of frameworks that themselves contain more than one subframework. For example, the Cocoa umbrella framework contains the Application Kit (user interface classes) framework, and the Foundation (non-user-interface classes) framework. See Using Shared Libraries and Frameworks in Loading Code at Runtime for more information.
- *Static archive libraries.* These files contain modules of reusable code that the static linker can add to your application at build time. Static archive libraries generally contain very small amounts of code

that are usable only to a few applications, or code that is difficult to maintain in a shared library for some reason. See Static Archive Libraries for more information.

- *Bundles.* These are executable files that your program can load at runtime using dynamic linking functions. Bundles implement plug-in functionality, such as file format importers for a word processor. The term *bundle* has two related meanings in OS X:
  - The actual object file containing the executable code
  - A directory containing the object file and associated resources. For example, applications in OS X are packaged as bundles. And, because these bundles are displayed in the Finder as a single file instead of as a directory, application bundles are also known as *application packages*. A bundle need not contain an object file. For more information on bundles, see *Bundle Programming Guide*.

  The latter usage is the more common. However, unless otherwise specified, this document refers to the former.

  See Loading Plug-in Code With Bundles in Loading Code at Runtime for more information.
- *Kernel extensions* are statically bound Mach-O files that are packaged similarly to bundles. Kernel extensions are loaded into the kernel address space and must therefore be built differently than other Mach-O file types; see the kernel documentation for more information. The kernel's runtime environment is very different from the userspace runtime, so it is not covered in this document.

To function properly in OS X, all object files except kernel extensions must be *dynamically bound*—that is, built with code that allows dynamic references to shared libraries.

By default, the static linker searches for frameworks and umbrella frameworks in `/System/Library/Frameworks` and for shared libraries and static archive libraries in `/usr/lib`. Bundles are usually located in the `Resources` directory of an application package. However, you can specify the pathname for a different location at link time (and, for development purposes, at runtime as well).

Modules—The Smallest Unit of Code

At the highest level, you can view an OS X shared library as a collection of modules. A *module* is the smallest unit of machine code and data that can be linked independently of other units of code. Usually, a module is an object file generated by compiling a single C source file. For example, given the source files `main.c`, `thing.c`, and `foo.c`, the compiler might generate the object files `main.o`, `thing.o`, and `foo.o`. Each of these output object files is one module. When the static linker is used to combine all three files into a dynamic shared library, each of the object files is retained as an individual unit of code and data. When linking applications and bundles, the static linker always combines all the object files into one module.

The static linker can also reduce several input modules into a single module. When building most dynamic shared libraries, it's usually a good idea to do this before creating the final shared library because function calls between modules are subject to a small amount of additional overhead. With `ld`, you can perform this optimization by using the command line as follows:

```
ld -r -o things.o thing1.o thing2.o thing3.o
```

Xcode performs this optimization by default.

Static Archive Libraries

To group a set of modules, you can use a *static archive library*, which is an archive file with a table of contents entry. The format is that used by the `ar` command. You can use the `libtool` command to build a static archive library, and you can use the `ar` command to manipulate individual modules in the library.

> **Note:** OS X `libtool` is not GNU `libtool`, see The Tools—Building and Running Mach-O Files for details.

In addition to Mach-O files, the static linker and other development tools accept static archive libraries as input. You might use a static archive library to distribute a set of modules that you do not want to include in a shared library but that you want to make available to multiple programs.

Although an `ar` archive can contain any type of file, the typical purpose is to group several object files together with a table of contents, forming a static archive library. The static linker can link the object files stored in a static archive library into a Mach-O executable or dynamic library. Note that you must use the `libtool` command to create the static library table of contents before an archive can be used as a static archive library.

> **Note:** For historical reasons, the `tar` file format is different from the `ar` file format. The two formats are not interchangeable.

The `ar` archive file format is described in *OS X ABI Mach-O File Format Reference*.

With the standard tools, you can pass the `-static` option to `libtool` to create a static archive library. The following command creates a static archive library named `libthing.a` from a set of intermediate object files, `thing1.o` and `thing2.o`:

```
libtool -static thing1.o thing2 -o libthing.a
```

Note that if you pass neither `-static` nor `-dynamic`, `libtool` assumes `-static`. It is, however, considered good style to explicitly pass `-static` when creating static archive libraries.

For more information, see the `libtool` and `ar` man pages.