Mach-O Programming Topics

General

2009-02-04

Apple Inc.
© 2003, 2009 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

.Mac is a registered service mark of Apple Inc.

Apple, the Apple logo, Carbon, Cocoa, eMac, Finder, Mac, Mac OS, Macintosh, Objective-C, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

Intel and Intel Core are registered trademarks of Intel Corportation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

PowerPC and and the PowerPC logo are trademarks of International Business Machines Corporation, used under license therefrom.

UNIX is a registered trademark of The Open Group

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction 7

Who Should Read This Document 8
Organization of This Document 8
See Also 8

Building Mach-O Files 11

The Tools—Building and Running Mach-O Files 11
The Products—Types of Mach-O Files You Can Build 12
Modules—The Smallest Unit of Code 14
Static Archive Libraries 14

Executing Mach-O Files 17

Launching an Application 17
Forking and Executing the Process 18
Finding Imported Symbols 18
    Binding Symbols 18
    Searching for Symbols 19
Scope and Treatment of Symbol Definitions 21

Loading Code at Runtime 25

Using Shared Libraries and Frameworks 25
    Maintaining Client Application Compatibility 26
    Packaging a Shared Library as a Framework 26
    Packaging Frameworks and Libraries Under an Umbrella Framework 28
Loading Plug-in Code With Bundles 28

Indirect Addressing 31

PowerPC Symbol References 31
IA-32 Symbol References 33
x86-64 Symbol References 34

Position-Independent Code 37

Eliminating Position-Independent Code References 37
Relocating Position-Independent Code 38
Relocations in the x86-64 Environment 39

x86-64 Code Model 43

Document Revision History 47

Index 49

Listings

Loading Code at Runtime 25

Listing 1 Building a framework 27
Listing 2 Building a private framework 27
Listing 3 Building a simple umbrella framework 28

Indirect Addressing 31

Listing 1 C code example for indirect function calls 31
Listing 2 Example of an indirect function call 32
Listing 3 Example of a position-independent, indirect function call 32
Listing 4 C program using an imported symbol 34
Listing 5 IA-32 symbol reference in assembly 34
Listing 6 Generating a stub, helper and lazy pointer 34

Position-Independent Code 37

Listing 1 C source code example for position-independent code 37
Listing 2 Position-independent code generated from the C example (with addresses in the left column) 38
Listing 3 Example assembly instructions and their corresponding relocations 40

x86-64 Code Model 43

Listing 1 C code and the corresponding assembly code 43
Listing 2 The code model for function calls 44

Introduction

Mac OS X supports a number of application environments, each with its own runtime rules, conventions, and file formats. In Mac OS X, kernel extensions, command-line tools, applications, frameworks, and libraries (shared and static) are implemented using Mach-O (Mach object) files.

The Mac OS X runtime architecture dictates how object files are laid out in the filesystem and how programs communicate with the kernel. The object file format used in Mac OS X is Mach-O .

A Mach-O file has the following regions of data (the complete format is described in *Mac OS X ABI Mach-O File Format Reference*):

- **Header:** Specifies the target architecture of the file, such as PPC, PPC64, IA-32, or x86-64.
- **Load commands:** Specify the logical structure of the file and the layout of the file in virtual memory.
- **Raw segment data:** Contains raw data for the segments defined in the load commands.

The following list describes other runtime environments supported in Mac OS X:

- **Classic** is a Mac OS X application that runs Mac OS 9 within its address space and provides bridging services that allow Mac OS X to interact with Mac OS 9 applications. Both classic 68K applications and PowerPC Code Fragment Manager (CFM) applications can run under Mac OS 9 in Classic. (Mac OS 9 does not support the 68K variant of Code Fragment Manager, so you cannot run CFM-68K applications in Mac OS X.)
- **LaunchCFMApp** is a command-line tool that runs programs created for the PowerPC Code Fragment Manager. The file format used by such programs is called **Preferred Executable Format (PEF)**. Carbon provides bridging for Code Fragment Manager applications that allows them to link to Mach-O–based code, but—for ease of debugging if for no other reason—it's generally a good idea to use Mach-O for Carbon applications.
- The HotSpot **Java virtual machine** is a Mac OS X application that executes Java bytecode applications and applets.
- The **Mac OS X kernel** supports kernel extensions (KEXTs), which are static Mach-O executable files that are loaded directly into the address space of the kernel. Because errant code can write directly to memory used by the kernel, kernel extensions have the potential to crash the operating system. You should generally avoid implementing functionality as kernel extensions if possible.

The Code Fragment Manager is documented in *Mac OS Runtime Architectures*, available from the Apple Developer Connection website.

This document discusses how you use the Mach-O file format. It describes what types of programs you can build, how programs are loaded and executed, how you can change the way programs are loaded and executed, how to load code at runtime, and how to load and link code at runtime. If you create or load bundles, shared libraries, or frameworks, you'll probably want to read and understand everything in this document.

Who Should Read This Document

If you write development tools for Mac OS X, you need to understand the information presented in this document.

This document is also useful for developers of shared libraries and frameworks, and for developers of applications that need to load code at runtime.

Organization of This Document

This document contains the following articles:

- "Building Mach-O Files" (page 11) describes how Mac OS X programs are built and describes the types of programs you can develop.
- "Executing Mach-O Files" (page 17) provides an overview of the Mac OS X dynamic loading process.
- "Loading Code at Runtime" (page 25) describes how to use shared libraries and frameworks and how to load plug-ins at runtime.
- "Indirect Addressing" (page 31) explains how a Mach-O file refers to symbols defined in another Mach-O file.
- "Position-Independent Code" (page 37) discusses the method by which the dynamic linker loads a region of code at a nonfixed virtual memory address.
- "x86-64 Code Model" (page 43) describes differences in the Mac OS X x86-64 user-space code model from the System V x86-64 code model.

This document also contains a revision history and an index.

See Also

You can access full reference documentation for the standard command-line development tools using the `man` tool on the command line, or by choosing Open Man Page from the Xcode Help menu.

This document provides information on the Mach-O runtime architecture. It does not address the following:

- Descriptions of the data structures that make up a Mach-O file. You can find this information in *Mac OS X ABI Mach-O File Format Reference*.
- If you are loading code at runtime but cannot or do not wish to use the `CFBundle` opaque type or the `NSBundle` class, you should refer to *Mac OS X ABI Dynamic Loader Reference*.
- The GCC C++ application binary interface—the specification of C++ class member layout, function/method name mangling, and related C++ issues. This information is documented for GCC 3.0 and later at http://www.codesourcery.com/cxx-abi/abi.html.
- The GCC Objective-C data structures and dynamic runtime functions. For this information, see *The Objective-C Programming Language*.

- The runtime environment of the Mac OS X kernel, Darwin. See *Mac OS X Technology Overview* for more information.

Source code from the Darwin project can be downloaded from http://developer.apple.com/darwin/.

You might also find the following books useful in conjunction with this document:

- *Mac OS Runtime Architectures*, Apple Computer, Inc. Available at http://developer.apple.com/tools/mpw-tools/books.html. Documents the classic 68K segment loader architecture, as well as the Code Fragment Manager Preferred Executable executable format used with classic PowerPC applications and with many Carbon applications.
- *Linkers and Loaders*, John R. Levine, Morgan Kaufmann, 2000, ISBN 1-55860-496-0. Describes the workings and operation of standard linkers from the earliest program loaders to the present dynamic link editors. Among the contents of this book are discussions of the classic BSD `a.out` format, the Executable and Linking Format (ELF) preferred by many current operating systems, the IBM System/360 linker output format, and the Microsoft Portable Executable (PE) format.
- *System V Application Binary Interface AMD64 Architecture Processor Supplement*. Found at http://www.x86-64.org/documentation, this document describes the System V x86-64 environment, on which the Mac OS X x86-64 environment is based.

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. Mac 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 OS X programs 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 Mac 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" (page 12)).

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 Mac 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 Mac 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 Mac 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 Mac 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 Mac 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 (page 17) 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" (page 25) in Loading Code at Runtime (page 25) 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" (page 25) in Loading Code at Runtime (page 25) 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" (page 25) in Loading Code at Runtime (page 25) 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" (page 14) 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 Mac OS X:
    - ❑ The actual object file containing the executable code
    - ❑ A directory containing the object file and associated resources. For example, applications in Mac 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" (page 28) in Loading Code at Runtime (page 25) 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 Mac 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 a Mac 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: Mac OS X `libtool` is not GNU `libtool`, see "The Tools—Building and Running Mach-O Files" (page 11) 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 *Mac 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.

Executing Mach-O Files

To perform their objectives, programs must execute processes and link to dynamic shared libraries. To work with other libraries or modules, your application must define references to symbols in those modules; those references are resolved at runtime. At runtime the symbol names of all the modules your application uses live in a shared namespace, similar to a directory. To allow for future enhancements to applications as well as the libraries they use, application and library developers must ensure the names they choose for their functions and data do not conflict with the names used in other modules.

The two-level namespace feature of Mac OS X v10.1 and later adds the module name as part of the symbol name of the symbols defined within it. This approach ensures a module's symbol names don't conflict with the names used in other modules. To perform special tasks or to provide an enhanced user experience, your application may need to launch other applications or create processes to run command-line tools. To maintain a high degree of interoperability and provide a consistent user experience, your applications should use specific system functions and frameworks to execute processes and launch applications.

This article provides an overview of the Mac OS X dynamic loading process. The process of loading and linking a program in Mac OS X mainly involves two entities: the Mac OS X kernel and the dynamic linker. When you execute a program, the kernel creates a process for the program, and loads the program and the dynamic linker shared library, usually `/usr/lib/dyld`, in the program's address space. The kernel then executes code in the dynamic linker that loads the libraries the program references. This article also describes the visibility symbols in a module get depending on how they are defined and the process of resolving symbol references at runtime.

Launching an Application

When you launch an application from the Finder or the Dock, or when you run a program in a shell, the system ultimately calls two functions on your behalf, `fork` and `execve`. The `fork` function creates a process; the `execve` function loads and executes the program. There are several variant exec functions, such as `execl`, `execv`, and `exect`, each providing a slightly different way of passing arguments and environment variables to the program. In Mac OS X, each of these other exec routines eventually calls the kernel routine `execve`.

When writing a Mac OS X application, you should use the Launch Services framework to launch other applications. Launch Services understands application packages, and you can use it to open both applications and documents. The Finder and the Dock use Launch Services to maintain the database of mappings from document types to the applications that can open them. Cocoa applications can use the class `NSWorkspace` to launch applications and documents; `NSWorkspace` itself uses Launch Services. Launch Services ultimately calls `fork` and `execve` to do the actual work of creating and executing the new process. For more information on Launch Services, see *Launch Services Programming Guide*.

Forking and Executing the Process

To create a process using BSD system calls, your process must call the `fork` system call. The `fork` call creates a logical copy of your process, then returns the ID of the new process to your process. Both the original process and the new process continue executing from the call to `fork`; the only difference is that `fork` returns the ID of the new process to the original process and zero to the new process. (The `fork` function returns `-1` to the original process and sets `errno` to a specific error value if the new process could not be created.)

To run a different executable, your process must call the `execve` system call with a pathname specifying the location of the alternate executable. The `execve` call replaces the program currently in memory with a different executable file.

A Mach-O executable file contains a header consisting of a set of load commands. For programs that use shared libraries or frameworks, one of these commands specifies the location of the linker to be used to load the program. If you use Xcode, this is always `/usr/lib/dyld`, the standard Mac OS X dynamic linker.

When you call the `execve` routine, the kernel first loads the specified program file and examines the `mach_header` structure at the start of the file. The kernel verifies that the file appear to be a valid Mach-O file and interprets the load commands stored in the header. The kernel then loads the dynamic linker specified by the load commands into memory and executes the dynamic linker on the program file.

The dynamic linker loads all the shared libraries that the main program links against (the **dependent libraries**) and binds enough of the symbols to start the program. It then calls the entry point function. At build time, the static linker adds the standard entry point function to the main executable file from the object file `/usr/lib/crt1.o`. This function sets up the runtime environment state for the kernel and calls static initializers for C++ objects, initializes the Objective-C runtime, and then calls the program's `main` function.

Finding Imported Symbols

When the dynamic linker loads a Mach-O file (which, for the purposes of this section, is called the **client program**), it connects the file's imported symbols to their definitions in a shared library or framework. This section describes the process of binding the imported symbols in one Mach-O file to their definitions in other Mach-O files. It also explains the process of finding a symbol. See also "Loading Plug-in Code With Bundles" (page 28) in Loading Code at Runtime (page 25) for information on finding symbols in plug-ins.

Binding Symbols

Binding is the process of resolving a module's references to functions and data in other modules (the **undefined external symbols**, sometimes called **imported symbols**). The modules may be in the same Mach-O file or in different Mach-O files; the semantics are identical in either case. When the application is first loaded, the dynamic linker loads the imported shared libraries into the address space of the program. When binding is performed, the linker replaces each of the program's imported references with the address of the actual definition from one of the shared libraries.

The dynamic linker can bind a program at several stages during loading and execution, depending on the options you specify at build time:

- With **just-in-time binding** (also called lazy binding), the dynamic linker binds a reference (and all the other references in the same module) when the program first uses the reference. The dynamic linker loads the shared libraries the client program depends on when the program is loaded. However, the dynamic linker doesn't bind the program's references to symbols within the shared libraries until the symbols are used.
- With **load-time binding**, the dynamic linker binds all the imported references immediately upon loading the program, or, for bundles, upon loading the bundle. To use load-time binding with the standard tools, specify the `-bind_at_load` option to `ld` to indicate that the dynamic linker must immediately bind all external references when the file is loaded. Without this option, `ld` sets up the output file for just-in-time binding.
- With **prebinding**, a form of load-time binding, the shared libraries referenced by the program are each prebound at a specified address. The static linker sets the address of each undefined reference in the program to default to these addresses. At runtime, the dynamic linker needs only to verify that none of the addresses have changed since the program was built (or since the prebinding was recomputed). If the addresses have changed, the dynamic linker must undo the prebinding by clearing the prebound addresses for all the undefined references and then proceed as if the program had been just-in-time bound. Otherwise, it does not need to perform any action to bind the program.

  Prebinding requires that each framework specify its desired base virtual memory address and that none of the prebound addresses of the loaded frameworks overlap. To prebind a file with the standard tools, specify the `-prebind` option to `ld`.
- **Weak references**, a feature introduced in Mac OS X v10.2, is useful for selectively implementing features that may be available on some systems, but not on others. This mode of binding allows a program to optionally bind to specified shared libraries. If the dynamic linker cannot find definitions for weak references, it sets them to `NULL` and continues to load the program. The program can check at runtime to find out whether or not a reference is null and, if so, avoid using the reference. You can specify both libraries and individual symbols to be weakly referenced.

Note: The Mac OS X weak linking design is derived from the classic Mac OS Code Fragment Manager implementation of weak linking. If you are familiar with the ELF executable format, you may be used to a different meaning for the terms *weak symbol* or *weak linking*, where a weak symbol may be overridden by a nonweak symbol. The equivalent Mac OS X feature is the *weak definition*—see "Scope and Treatment of Symbol Definitions" (page 21) for more information

If no other type of binding is specified for a given library, the static linker sets up the program's undefined references to that library to use just-in-time binding.

Searching for Symbols

A **symbol** is a generic representation of the location of a function, data variable, or constant in an executable file. References to functions and data in a program are references to symbols. To refer to a symbol when using the dynamic linking routines, you usually pass the name of the symbol, although some functions also accept a number representing the ordering of the symbol in the executable file. The name of a symbol representing a function that conforms to standard C calling conventions is the name of the function with an underscore prefix. Thus, the name of the symbol representing the function `main` would be `_main`.

Programs created by the Mac OS X v10.0 development tools add all symbols from all loaded shared libraries into a single global list. Any symbol that your program references can be located in any shared library, as long as that shared library is one of the program's dependent libraries (or one of the dependent libraries of the dependent libraries).

Mac OS X v10.1 introduced the two-level symbol namespace feature. The first level of the two-level namespace is the name of the library that contains the symbol, and the second is the name of the symbol. With the two-level namespace feature enabled, when the static linker records references to imported symbols, it records a reference to the name of the library that contains the symbol and the name of the symbol. Linking your programs with the two level namespace feature offers two benefits over the flat namespace:

- **Enhanced performance when searching for symbols.** With the two-level namespace, the dynamic linker knows exactly where to start looking for the implementation of a symbol. With a flat namespace, the dynamic linker must search all the loaded libraries for the one that contains the symbol.
- **Enhanced forward compatibility.** In the flat namespace, two or more libraries cannot contain symbols with different implementations that share the same name because the dynamic linker cannot know which library contains the preferred implementation. This is not initially a problem, because the static linker catches any such problems when you first build the application. However, if the vendor of one of your dependent shared libraries later releases a new version of the library that contains a symbol with the same name as one in your program or in another dependent shared library, your program will fail to run.

Your application must link directly to the shared library that contains the symbol (or, if the library is part of an umbrella framework, to the umbrella framework that contains it).

When obtaining symbols in a program built with the two-level namespace feature enabled, you must specify a reference to the shared library that contains the symbols.

By default, the static linker in Mac OS X v10.1 and later uses a two-level namespace for all Mach-O files.

Note: The Mac OS X two-level namespace feature is loosely based on the design of the Code Fragment Manager's namespace. A two-level namespace is approximately equivalent to the namespace used to look up symbols in code fragments. Because Code Fragment Manager always requires an explicit reference to the library in which a symbol should be found, there is no Code Fragment Manager equivalent to a flat namespace search.

For programs that do not have a two-level namespace, you can tell the linker to define references to undefined symbols even if the linker cannot find the library that contains them. When you build an executable with such undefined symbols, you are making the assumption that one of the other files loaded as part of the executable file at runtime contains those symbols. Bundles and shared libraries sometimes use this option to reference symbols defined in the main executable. However, this causes you to lose the performance and compatibility benefits of two-level namespaces. It's usually better to explicitly link against an executable that defines the references. However, if you must link with undefined references, you can do it by enabling the flat namespace feature and suppressing undefined reference warnings, using the options `-flat_namespace` and `-undefined suppress` as in the following command line:

```
ld -o my_tool -flat_namespace -undefined suppress peace.o love.o
```

When building executables with a two-level namespace, you can allow the remaining undefined symbols to be looked up by the dynamic linker if the program is targeted for Mac OS X v10.3 and later (the `MACOSX_DEPLOYMENT_TARGET` environment variable is set to `10.3` or higher). To take advantage of this feature, use the `-undefined dynamic_lookup` option.

To build executables with a two-level namespace, the static linker must be able to find the source library for each symbol. This can present difficulties for authors of bundles and dynamic shared libraries that assume a flat, global symbol namespace. To build successfully with the two-level namespace, keep the following points in mind:

- Bundles that need to reference symbols defined in the program's main executable must use the `-bundle_loader` static linker option. The static linker can then search the main executable for the undefined symbols.
- Shared libraries that need to reference symbols defined in the program's main executable must load the symbol dynamically using a function that does not require a library reference, such as `dlsym` or `NSLookupSymbolInImage` (*Mac OS X ABI Dynamic Loader Reference*).

Note: A two-level symbol namespace can be searched using functions for doing flat symbol searches.

Scope and Treatment of Symbol Definitions

Symbols in an object file may exist at several levels of scope. This section describes each of the possible scopes that a symbol may be defined at, and provides samples of C code used to create each symbol type. These samples work with the standard developer tools; a third party tool set may have different conventions.

A **defined external symbol** is any symbol defined in the current object file, including functions and data. The following C code defines external symbols:

```
int x = 0;
double y = 99 __attribute__((visibility("default")));   // GCC 4.0 only
```

An **undefined external symbol** is any symbol defined in a file outside of the current file. The following C code defines two external symbols, a variable and a function:

```
extern int x;
extern void SomeFunction(void);
```

A **common symbol** is a symbol that may appear in multiple intermediate object files. The static linker permits multiple common symbol definitions with the same name in input files, and copies the one with the largest size to the final product. If there is another symbol with the same name as a common symbol, the static linker ignores the common symbol instead.

The standard C compiler generates a common symbol when it sees a **tentative definition**—a global variable that has no initializer and is not marked `extern`. The following line is an example of a tentative definition:

```
int x;
```

A multi-module shared library, which `ld` builds by default, cannot have common symbols. However, you can build a shared library as a single module with the `-single_module` flag. To eliminate common symbols in an existing shared library, you must either explicitly define the symbol (with an initialized value, for example) in one of the modules in the shared library, or pass the `-fno-common` flag to the compiler.

A **private defined symbol** is a symbol that is not visible to other modules. The following C code defines a private symbol:

```
static int x;
```

A **private external symbol** is a defined external symbol that is visible only to other modules within the same object file as the module that contains it. The standard static linker changes private external symbols into private defined symbols unless you specify otherwise (using the `-keep_private_externs` flag).

You can mark a symbol as private external by using the `__private_extern__` keyword (which works only in C) or the `visibility("hidden")` attribute (which works both in C and C++ with GCC 4.0), as in this example:

```
__private_extern__ int x = 0;                       // C only
int y = 99 __attribute__((visibility("hidden")));   // C and C++, GCC 4.0 only
```

A **coalesced symbol** is a symbol that may be defined in multiple object files but that the static linker generates only one copy of in the output file. This can save a lot of memory with certain C++ language features that the compiler must generate for each individual object file, such as virtual function tables, runtime type information (RTTI), and C++ template instantiations. The compiler determines which constructs should be coalesced; no work on your part is required.

A **weak reference** is an undefined external symbol that need not be found in order for the client program to successfully link. If the symbol does not exist, the dynamic linker sets the address of the symbol to zero. Files with weak references can be used only in Mac OS X v10.2 and later. The following C code demonstrates conditionalizing an API call using a weak reference:

```
/* Only call this API if it exists */
if ( SomeNewFunction != NULL )
        SomeNewFunction();
```

To specify that a function should be treated as a weak reference, use the `weak_import` attribute on a function prototype, as demonstrated by the following code:

```
void SomeNewFunction(void) __attribute__((weak_import));
```

A **coalesced weak reference** is an undefined external reference to a symbol defined in multiple object files. In Mac OS X v10.4 and later (with GCC 4.0 and later), you can specify that a symbol be made into a coalesced weak reference by adding the weak attribute to the symbol's declaration. For example:

```
void SomeNewFunction(void) __attribute__((weak));
```

Note: Programmers who use other operating systems may be familiar with the concept of symbols that are marked with a COMDAT flag; a coalesced symbol is the Mac OS X equivalent feature.

A **weak definition** is a symbol that is ignored by the linker if an otherwise identical but nonweak definition exists. This is used by the standard C++ compiler to support C++ template instantiations. The compiler marks implicit—and not explicit—template instantiations as weak definitions. The static linker then prefers any explicit template instantiation to an implicit one for the same symbol, which provides correct C++ linking semantics. As with coalesced symbols, the compiler determines the constructs that require the weak definitions feature; no work on your part is required.

Note: Files with weak definitions can be used only in Mac OS X v10.2 and later. The static linker changes any weak definitions into nonweak definitions, so this is only a concern for intermediate object files and static libraries that you wish to deploy on earlier versions of Mac OS X.

A **debugging symbol** is a symbol generated by the compiler that allows the debugger to map from addresses in machine code to locations in source code. The standard compilers generate debugging symbols using either the Stabs format or the DWARF format (supported in Xcode 2.4 and later). When using the Stabs format, debugging symbols, like other symbols, are stored in the symbol table (see *Mac OS X ABI Mach-O File Format Reference*). But with the DWARF format, debugging symbols are stored in a specialized segment: the `__DWARF` segment. With DWARF you also have the option of storing debugging symbols in a separate debug-information file, which reduces the size of the binary files while permitting a full debugging experience when the corresponding debug-information files are available.

Loading Code at Runtime

You may need to use **dynamic shared libraries**, which store reusable code, in your applications to take advantage of functionality used by more than one application. For example, when you develop Cocoa applications, at a minimum your application links against the Foundation and Application Kit frameworks. Through this practice, your program can automatically take advantage of improvements in those frameworks as your application's users update the system software in their computers. See "The Products—Types of Mach-O Files You Can Build" (page 12) for more information on dynamic shared libraries.

If you develop shared libraries and distribute them as framework bundles to be used by other developers in their applications, you should ensure changes you make to the libraries (to implement new features, for example) don't break current versions of the applications that use them. You maintain compatibility by exposing the same programming interface to client applications between upgrades of the shared libraries.

When a change to your framework's API is required to implement a feature, you should make available in the same framework bundle the last version of the framework that exposes the API current client applications expect, in addition to the version of the framework that exposes the new, incompatible API. If you follow this guideline, developers of client applications don't have to revise them every time your framework's API changes. And developers that choose to update their applications can take advantage of the features you added to the framework. To ensure that earlier versions of your framework's client applications don't break when the framework is updated, you must package the shared libraries and their resources within the framework bundle in a way that allows earlier versions of client applications to continue using the versions of the framework that they understand.

This article describes how you can load code at runtime. It shows the benefits of using shared libraries and explains how they are packaged inside frameworks to maintain client-application compatibility across updates to the frameworks. It also describes how the Mac OS X runtime environment takes advantage of bundles to allow you to load plug-in code at runtime.

Using Shared Libraries and Frameworks

Programmers often refer to dynamic shared libraries using different names, such as *dynamically linked shared libraries*, *dynamic libraries*, *DLLs*, *dylibs*, or just *shared libraries*. In Mac OS X, all these names refer to the same thing: a library of code dynamically loaded into a process at runtime.

Dynamic shared libraries allow the operating system as a whole to use memory efficiently. Each process in Mac OS X has its own virtual address space. The Mac OS X kernel allows regions of logical memory to be mapped into multiple processes at different addresses. The dynamic linker takes advantage of this feature by mapping the same read-only copy of the shared library code into the address space of each process. The result is that only one physical copy of a shared library is in memory at any time, even though many processes may use it at the same time. Data, such as variables and constants, contained by a shared library is mapped into each client process using the kernel's copy-on-write optimization capability. With copy-on-write, the data is shared among processes until one of the processes attempts to change the data. At that point, the kernel creates a writable copy of the data private to that process. The other processes continue to use the read-only shared copy. Thus, additional memory for data is allocated only when absolutely necessary.

Dynamic shared libraries also provide a way for programs to seamlessly benefit from system upgrades. When the system is upgraded, the shared libraries are updated, but the programs need not be. Since they are dynamically bound to the shared libraries, the programs can continue to call the same functions and the updated implementation of the shared libraries is executed. For more information on dynamic-library development and usage, see *Dynamic Library Programming Topics*.

Maintaining Client Application Compatibility

This is an overview of various parameters that affect compatibility with client applications. You can set these parameters at build time. For a more detailed discussion of this topic, see Dynamic Library Design Guidelines in *Dynamic Library Programming Topics*.

Shared libraries have two version numbers, which allow you to create versions of a shared library that are **binary compatible** (that is, they do not require client programs to be recompiled) with the functions exported by the earlier versions of a library:

- The **current version** of the library specifies the current version number of the library's implementation. A client program can examine this version number to find out the exact version of the library, which can be useful for checking for bug fixes and feature additions. The shared library can also examine the version number that the client program originally linked against, which can be useful for maintaining backwards compatibility.
- The **compatibility version** of the library specifies the version of the library's API that the shared library claims to be backward-compatible with. If the compatibility version of the shared library is more recent than the version recorded with the client program, the program fails to link and an error occurs.

The **install name** is the pathname used by the dynamic linker to find a shared library at runtime. The install name is defined by the shared library and recorded into the client program by the static linker.

Packaging a Shared Library as a Framework

A **framework** is a shared library packaged with associated resources, such as headers, localized strings, and documentation, installed in a standard folder hierarchy, usually in a standard location in the file system. The folders usually contain related header files, documentation in any format, and resource files. A framework may contain multiple versions of itself, and each version may have its own set of resources, documentation, and header files.

From a tools perspective, a framework is a shared library whose install name ends in the form *`frameworkName`*`.framework/Versions/`*`versionName`*`/`*`frameworkName`* or the form *`frameworkName`*`.framework/`*`frameworkName`*.

You create a framework by building a normal dynamic shared library into a folder with the same name and the `.framework` extension. For example, to create a framework named Chaos, place a dynamic shared library named `Chaos` in a folder called `Chaos.framework`. You can create other folders inside this folder to store related resources, such as header files, documentation, and images (the standard folder names for these are called, respectively, `Headers`, `Documentation`, and `Resources`).

You can locate private frameworks and shared libraries in an application package using a relative-path install name beginning with `@executable_path`, such as `@executable_path/../Frameworks/MyFramework.framework`. This is useful for sharing functionality with plug-ins (bundles).

Apple follows a standard framework versioning convention, different from the shared library version numbering system. By versioning your framework, you can ship older versions of your framework alongside newer versions, to allow older clients to continue functioning, while still allowing you to advance the design of the framework in ways not compatible with older clients.

To version your framework, create a parent folder inside the framework called `Versions`, create a subfolder in `Versions` using a naming scheme of your choice, and build the framework shared library and other folders in this subfolder. Then create symbolic links in the framework's root folder to point to the shared library and folders. When you need to create a version of your framework that is not compatible with the previous version, build it into a new directory in the versions directory and update the symlinks to point to the new version. When a client links to a versioned framework, the install name recorded in the client executable includes the full path to the shared library executable, and the dynamic linker, thus, loads only that version.

For example, a client links to a framework called `Peace.framework`, and the symlinks in `Peace.framework` point to the latest version, which is named `B`.The install name of the framework ends with `Peace.framework/Versions/B/Peace`. The static linker records this install name in the client. When the client is loaded, the dynamic linker attempts to load the shared library with this install name. Note that, while frameworks that ship with the system usually name successive versions with consecutive letters of the English alphabet (A through Z), you can use any name you want.

A framework developer can build a simple, versioned framework in four steps:

1. Create the framework version directory.
2. Compile the framework executable into the framework version directory.
3. Create a symbolic link named `Current` that points to the framework version directory.
4. Create a symbolic link to the framework executable in the parent framework directory.

The shell commands in Listing 1 builds a framework named `Bliss` from the C source files `Peace.c` and `Love.c`. The resulting framework has the install name `Bliss.framework/Versions/A/Bliss`.

Listing 1 Building a framework

```
gcc -c -o Peace.o Peace.c
gcc -c -o Love.o Love.c
mkdir -p Bliss.framework/Versions/A
gcc -dynamiclib -o Bliss.framework/Versions/A/Bliss Peace.o Love.o
cd ./Bliss.framework/Versions && ln -sf A Current
cd ./Bliss.framework && ln -sf Versions/Current/Bliss Bliss
```

Listing 2 demonstrates how to create a **private framework**—that is, a framework located in an application package. Specify the install name explicitly during the linking phase and prefix it with `@executable_path`. The install name of the resulting framework is `@executable_path/../Frameworks/Bliss.framework/Versions/A/Bliss`.

Listing 2 Building a private framework

```
mkdir -p Bliss.framework/Versions/A
gcc -c Peace.c Love.c
libtool -dynamic
  -install_name @executable_path/../Frameworks/Bliss.framework/Versions/A/Bliss
  -o Bliss.framework/Versions/A/Bliss Peace.o Love.o
  -framework System
```

For detailed information about designing and using frameworks, see *Framework Programming Guide*.

Packaging Frameworks and Libraries Under an Umbrella Framework

An **umbrella framework** is a framework that serves as the "parent" of a group of frameworks and shared libraries that implement related functionality. Umbrella frameworks help manage extremely large development projects with complex interdependencies, such as subsystems of Mac OS X itself. For all other projects, a single framework should suffice (and is better for load-time performance).

To create an umbrella framework, you can take a normal framework and designate a subset of its imported frameworks as subframeworks. The subframeworks themselves need not be aware that they are part of the umbrella. With `ld`, you can use the `-sub_umbrella` option to designate a subframework.

When your program links against an umbrella framework, it also implicitly links against all the subframeworks. Symbols located in subframeworks of umbrella frameworks are recorded in the client program as if they were implemented directly in the umbrella framework. This feature allows the contents of the umbrella framework to change over time while preserving compatibility with older client programs.

To ensure that clients link to the "parent" umbrella framework and not one of the subframeworks, the subframework can be built with a special load command to prevent unauthorized linking. When a client tries to link directly to such a subframework, the static linker produces an error. However, the subframework can authorize specific clients to link against it, and all subframeworks of an umbrella framework are implicitly authorized to link against each other. (Load commands are explained in *Mac OS X ABI Mach-O File Format Reference*. The particular load commands referenced here are documented as `sub_framework_command` and `sub_client_command`; `ld` generates them if given the `-sub_framework <parent_umbrella_name>` and `-sub_client <client_name>` options.) Note that these conventions are enforced at build time by the static linker but ignored by the dynamic linker at runtime.

You can also include libraries in umbrella frameworks. For example the Foundation framework includes both the Objective-C runtime library (`libobjc`) as a sublibrary and the Core Foundation framework as a subframework. You may build Foundation using a variation on the commands listed in Listing 3.

Listing 3 Building a simple umbrella framework

```
mkdir Foundation.framework
gcc -dynamiclib -o Foundation.framework/Foundation -sub_umbrella CoreFoundation
   -sub_library libobjc -framework CoreFoundation -lobjc Foundation.o
```

By convention, subframeworks of an umbrella framework live within the `Frameworks` directory in the root directory of the umbrella framework, although this is obviously not a technical requirement. For example, the Cocoa framework is an umbrella framework that includes the AppKit framework; the AppKit framework is itself an umbrella framework that includes Foundation and Application Services as subframeworks.

Loading Plug-in Code With Bundles

Bundles provide the Mac OS X mechanism for loading extension (or plug-in) code into an application at runtime. Typically, a bundle links against the application binary to gain access to the application's exported API. Bundles can be—but are not required to be—packaged with resources, using the same folder hierarchy as that of an application package. In some cases (depending on the code in the bundle), bundles can also be unloaded.

Mac OS X supports several schemes that allow third-party developers to extend the capabilities of your application by writing plug-in code that your program can load at runtime. Although you can use any one of these plug-in schemes in any type of application, some are more suited to particular situations than others. For example:

- To load Objective-C classes at runtime, use the Foundation framework class `NSBundle`. `NSBundle` provides general services for referring to a packaged program, whether the program is an application or a plug-in.
- To load C functions at runtime, use the Core Foundation framework object `CFBundle`, which, like the `NSBundle` class, provides general services for referring to a packaged program, whether the program is an application or a plug-in.
- The Core Foundation framework object `CFPlugin` implements a small subset of the Microsoft Component Object Model (COM) standard. COM allows you to instantiate C functions and data in an object-oriented manner at runtime.
- Carbon developers can also use **Code Fragment Manager (CFM)** to load code fragments updated for Carbon from PEF files. For more information, see the Code Fragment Manager documentation.
- In general, for applications or libraries targeted for Mac OS X v10.4 or later, use the dynamic loader compatibility functions, defined in `/usr/include/dlfcn.h`, to load and link bundle files. These functions are the preferred way to load code at runtime. They are particularly helpful when porting UNIX tools that support plug-ins to Mac OS X. See “Dynamic Loader Compatibility Functions” in *Mac OS X ABI Dynamic Loader Reference* for more information.

Note: The dynamic linker in Mac OS X v10.0 causes your program to crash if you ask it to load programs that are built with a two-level namespace hint table. By default, the static linker in Mach OS X v10.0 creates bundles that do not include the two-level namespace hint table. If you want your program to run in Mac OS X 10.0 and are developing in Mac OS X v10.1 or later, use the `-flat_namespace` flag to ask the static linker to create the program using a flat namespace.

The `CFBundle` and `CFPlugin` objects can both be used from Carbon applications running in both Mac OS 9 and Mac OS X. Both the `NSBundle` class and the `CFPlugin` object allow you to package plug-in code with the resources associated with the plug-in (such as graphics files and documentation), similar to the packaging for an application. To load COM objects in Mac OS 9, `CFPlugin` uses Code Fragment Manager, and in Mac OS X, `CFPlugin` uses the object file image `dyld` library functions.

For more information on loading resources using the `NSBundle` class, see *Resource Programming Guide*. For more information on Code Fragment Manager, see *Mac OS Runtime Architectures*. For more information on `CFPlugin` and COM, see *Plug-ins*.

Indirect Addressing

Indirect addressing is the name of the code generation technique that allows symbols defined in one file to be referenced from another file, without requiring the referencing file to have explicit knowledge of the layout of the file that defines the symbol. Therefore, the defining file can be modified independently of the referencing file. Indirect addressing minimizes the number of locations that must be modified by the dynamic linker, which facilitates code sharing and improves performance.

When a file uses data that is defined in another file, it creates symbol references. A **symbol reference** identifies the file from which a symbol is imported and the referenced symbol. There are two types of symbol references: nonlazy and lazy.

- **Nonlazy symbol references** are resolved (bound to their definitions) by the dynamic linker when a module is loaded.

  A nonlazy symbol reference is essentially a **symbol pointer**—a pointer-sized piece of data. The compiler generates nonlazy symbol references for data symbols or function addresses.
- **Lazy symbol references** are resolved by the dynamic linker the first time they are used (not at load time). Subsequent calls to the referenced symbol jump directly to the symbol's definition.

  Lazy symbol references are made up of a symbol pointer and a **symbol stub**, a small amount of code that directly dereferences and jumps through the symbol pointer. The compiler generates lazy symbol references when it encounters a call to a function defined in another file.

The following sections describe how symbol references are implemented for the PowerPC and IA-32 architectures. For detailed information on the PowerPC and IA-32 symbol stubs, see *Mac OS X Assembler Reference*.

PowerPC Symbol References

In the PowerPC architecture, when generating calls to functions that are defined in other files, the compiler creates a symbol stub and a lazy symbol pointer. The **lazy symbol pointer** is an address that is initially set to glue code that calls the linker glue function `dyld_stub_binding_helper`. This glue function calls the dynamic linker function that performs the actual work of binding the stub. On return from `dyld_stub_binding_helper`, the lazy pointer points to the actual address of the external function.

The simple code example in Listing 1 might produce two different types of symbol stubs, depending on whether it is compiled with position-independent code generation. Listing 2 shows indirect addressing without position-independent code, and Listing 3 (page 32) shows both indirect addressing and position-independent code.

Listing 1 C code example for indirect function calls

```
extern void bar(void);
void foo(void)
{
    bar();
}
```

Listing 2 Example of an indirect function call

```
.text
        ; The function foo
        .align 2
        .globl _foo
_foo:
        mflr r0         ; move the link register into r0
        stw r0,8(r1)    ; save the link register value on the stack
        stwu r1,-64(r1) ; set up the frame on the stack
        bl L_bar$stub   ; branch and link to the symbol stub for _bar
        lwz r0,72(r1)   ; load the link register value from the stack
        addi r1,r1,64   ; removed the frame from the stack
        mtlr r0         ; restore the link register
        blr             ; branch to the link register to return

.symbol_stub            ; the standard symbol stub section
L_bar$stub:
        .indirect_symbol _bar               ; identify this symbol stub for the
                                            ;  symbol _bar
        lis r11,ha16(L_bar$lazy_ptr)        ; load r11 with the high 16 bits of the
                                            ;  address of bar’s lazy pointer
        lwz r12,lo16(L_bar$lazy_ptr)(r11)   ; load the value of bar’s lazy pointer
                                            ;  into r12
        mtctr r12                           ; move r12 to the count register
        addi r11,r11,lo16(L_bar$lazy_ptr)   ; load r11 with the address of bars lazy
                                            ; pointer
        bctr                                ; jump to the value in bar’s lazy pointer

.lazy_symbol_pointer    ; the lazy pointer section
L_bar$lazy_ptr:
        .indirect_symbol _bar               ; identify this lazy pointer for symbol
                                            ;  _bar
        .long dyld_stub_binding_helper      ; initialize the lazy pointer to the stub
                                            ;  binding helper address
```

Listing 3 Example of a position-independent, indirect function call

```
.text
        ; The function foo
        .align 2
        .globl _foo
_foo:
        mflr r0         ; move the link register into r0
        stw r0,8(r1)    ; save the link register value on the stack
        stwu r1,-80(r1) ; set up the frame on the stack
        bl L_bar$stub   ; branch and link to the symbol stub for _bar
        lwz r0,88(r1)   ; load the link register value from the stack
        addi r1,r1,80   ; removed the frame from the stack
        mtlr r0         ; restore the link register
        blr             ; branch to the link register to return

.picsymbol_stub         ; the standard pic symbol stub section
L_bar$stub:
        .indirect_symbol _bar       ; identify this symbol stub for the symbol _bar
        mflr r0                     ; save the link register (LR)
        bcl 20,31,L0$_bar           ; Use the branch-always instruction that does not
                                    ;  affect the link register stack to get the
                                    ;  address of L0$_bar into the LR.
L0$_bar:
        mflr r11                                ; then move LR to r11
                                                ; bar's lazy pointer is located at
                                                ;  L0$_bar + distance
        addis r11,r11,ha16(L_bar$lazy_ptr-L0$_bar); L0$_bar plus high 16 bits of
                                                ;  distance
        mtlr r0                                 ; restore the previous LR
        lwz r12,lo16(L_bar$lazy_ptr-L0$_bar)(r11); ...plus low 16 of distance
        mtctr r12                               ; move r12 to the count register
        addi r11,r11,lo16(L_bar$lazy_ptr-L0$_bar); load r11 with the address of bar's
                                                ; lazy pointer
        bctr                                    ; jump to the value in bar's lazy
                                                ;  pointer

.lazy_symbol_pointer    ; the lazy pointer section
L_bar$lazy_ptr:
        .indirect_symbol _bar           ; identify this lazy pointer for symbol bar
        .long dyld_stub_binding_helper  ; initialize the lazy pointer to the stub
                                        ;  binding helper address.
```

As you can see, the `__picsymbol_stub` code in Listing 3 (page 32) resembles the position-independent code generated for Listing 2 (page 38). For any position-independent Mach-O file, symbol stubs must obviously be position independent, too.

The static linker performs two optimizations when writing output files:

- It removes symbol stubs for references to symbols that are defined in the same module, modifying branch instructions that were calling through stubs to branch directly to the call.
- It removes duplicates of the same symbol stub, updating branch instructions as necessary.

Note that a routine that branches indirectly to another routine must store the target of the call in GPR11 or GPR12. Standardizing the registers used by the compiler to store the target address makes it possible to optimize dynamic code generation. Because the target address needs to be stored in a register in any event, this convention standardizes what register to use. Routines that may have been called directly should not depend on the value of GR12 because, in the case of a direct call, its value is not defined.

IA-32 Symbol References

In the IA-32 architecture, symbol references are implemented as a symbol stub and a lazy symbol pointer combined into one `JMP` instruction. Initially, such instructions point to the dynamic linker. When the dynamic linker encounters such an instruction, it locates the referenced symbol and modifies the `JMP` instruction to point directly to this symbol. Therefore, subsequent executions of the `JMP` instruction, jump directly to the referenced symbol.

Listing 4 and Listing 5 show a simple C program and the IA-32 assembly generated highlighting the symbol stub and nonlazy pointer for an imported symbol.

Listing 4 C program using an imported symbol

```
#include <stdio.h>
main( int arc, char *argv[])
{
  fprintf(stdout, "hello, world!\n") ;
}
```

Listing 5 IA-32 symbol reference in assembly

```
        .cstring
LC0:
        .ascii "hello, world!\12\0"
        .text
.globl _main
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    L___sF$non_lazy_ptr, %eax
        addl    $88, %eax
        movl    %eax, 12(%esp)
        movl    $14, 8(%esp)
        movl    $1, 4(%esp)
        movl    $LC0, (%esp)
        call    L_fwrite$stub                        ; call to imported symbol
        leave
        ret
        .section
__IMPORT,__jump_table,symbol_stubs,self_modifying_code+pure_instructions,5
L_fwrite$stub:                                        ; symbol stub
        .indirect_symbol _fwrite
        hlt ; hlt ; hlt ; hlt ; hlt
        .section __IMPORT,__pointers,non_lazy_symbol_pointers
L___sF$non_lazy_ptr:                                  ; nonlazy pointer
        .indirect_symbol ___sF
        .long   0
        .subsections_via_symbols
```

x86-64 Symbol References

This section describes deviations from the System V x85-64 environment in the area of symbol references.

Note: The Mac OS X x86-64 environment uses Mach-O (not ELF) as its executable file format.

The static linker is responsible for generating all stub functions, stub helper functions, lazy and nonlazy pointers, as well as the indirect symbol table needed by the dynamic loader (`dyld`).

For reference, Listing 6 shows how a a stub, helper, and lazy pointer are generated.

Listing 6 Generating a stub, helper and lazy pointer

```
_foo$stub:            jmp     *_foo$lazy_pointer(%rip)
```

```
_foo$stub_helper:      leaq     _foo$lazy_pointer(%rip),%r11
                       jmp      dyld_stub_binding_helper
_foo$lazy_pointer:     .quad    _foo$stub_helper
```

Position-Independent Code

Position-independent code, or **PIC**, is the name of the code generation technique used in the PowerPC environments that allows the dynamic linker to load a region of code at a nonfixed virtual memory address. Without some form of position-independent code generation, the operating system would need to place all code you wanted to be shared at fixed addresses in virtual memory, which would make maintenance of the operating system remarkably difficult. For example, it would be nearly impossible to support shared libraries and frameworks because each one would need to be preassigned an address that could never change.

Mach-O position-independent code design is based on the observation that the `__DATA` segment is always located at a constant offset from the `__TEXT` segment. That is, the dynamic loader, when loading any Mach-O file, never moves a file's `__TEXT` segment relative to its `__DATA` segment. Therefore, a function can use its own current address plus a fixed offset to determine the location of the data it wishes to access. All segments of a Mach-O file, not only the `__TEXT` and `__DATA` segments, are at fixed offsets relative to the other segments.

Note: If you are familiar with the Executable and Linking Format (ELF), you may note that Mach-O position-independent code is similar to the GOT (global offset table) scheme. The primary difference is that Mach-O code references data using a direct offset, while ELF indirects all data access through the global offset table.

Eliminating Position-Independent Code References

Position-independent code is typically required for shared libraries and bundles to allow the dynamic loader to relocate them to different addresses at load time. However, it is not required for applications that typically reside at the same address in virtual memory. GCC 3.1 introduces a new option, called `-mdynamic-no-pic`. This option both reduces the code size of application executables and improves their performance by eliminating position-independent code references, while preserving indirect calls to shared libraries and indirection to undefined symbols. If you use Xcode to create your application, this option is enabled by default. For an example of dynamic code generated without PIC, see Listing 2 (page 32).

Listing 2 shows an example of the position-independent code generated for the C code in Listing 1.

Listing 1 C source code example for position-independent code

```
struct s { int member1; int member2; };

struct s bar = {1,2};

int foo(void)
{
    return bar.member2;
}
```

Listing 2 Position-independent code generated from the C example (with addresses in the left column)

```
                    .text
                    ; The function foo
                    .align 2
                    .globl _foo
0x0     _foo:       mflr r0                     ; save the link register (LR)
0x4                 bcl 20,31,L1$pb             ; Use the branch always instruction
                                                ;  that does not affect the link
                                                ;  register stack to get the address
                                                ;  of L1$pb into the LR.
0x8     L1$pb:      mflr r10                    ; then move LR to r10
0xc                 mtlr r0                     ; restore the previous LR
                                                ; bar is located at L1$pc + distance
0x10                addis r9,r10,ha16(_bar-L1$pb); L1$pb plus high 16 bits of distance
0x14                la r9,lo16(_bar-L1$pb)(r9)  ; plus low 16 of distance
                                                ; => r9 now contains address of bar
0x18                lwz r3,4(r9)                ; return bar.member2
0x1c                blr
.data
                    ; The initialized structure bar
                    .align 2
                    .globl _bar
0x20    _bar:       .long 1                     ; member1’s initialized value
0x24                .long 2                     ; member2’s initialized value
```

To calculate the address of `_bar`, the generated code adds the address of the `L1$pb` symbol (`0x8`) to the distance to `bar`. The distance to `bar` from the address of `L1$pb` is the value of the expression `_bar - L1$pb`, which is `0x18` (`0x20 - 0x8`).

Relocating Position-Independent Code

To support relocation of code in intermediate object files, Mach-O supports a section difference relocation entry format. Relocation entries are described in *Mac OS X ABI Mach-O File Format Reference*.

Each of the add-immediate instructions is represented by two relocation entries. For the `addis` instruction (at address `0x10` in the example) the following tables list the two relocation entries. The fields of the first relocation entry (of type `scattered_relocation_info`) are:

`r_scattered`	`1`—true
`r_pcrel`	`0`—false
`r_length`	`2`—indicating 4 bytes
`r_type`	`PPC_RELOC_HA16_SECTDIFF`
`r_address`	`0x10`—the address of the `addis` instruction
`r_value`	`0x20`—the address of the symbol `_bar`

The values of the second relocation entry are:

`r_scattered`	`1`—true
`r_pcrel`	`0`—false
`r_length`	`2`—indicating 4 bytes
`r_type`	`PPC_RELOC_PAIR`
`r_address`	`0x18`—the low 16 bits of the expression (`_bar - L1$pb`)
`r_value`	`0x8`—the address of the symbol `L1$pb`

The first relocation entry for the `la` instruction (at address `0x14` in the example) is:

`r_scattered`	`1`—true
`r_pcrel`	`0`—false
`r_length`	`2`—indicating 4 bytes
`r_type`	`PPC_RELOC_LO16_SECTDIFF`
`r_address`	`0x14`—the address of the `addi` instruction
`r_value`	`0x20`—the address of the symbol `_bar`

The values of the second relocation entry are:

`r_scattered`	`1`—true
`r_pcrel`	`0`—false
`r_length`	`2`—indicating 4 bytes
`r_type`	`PPC_RELOC_PAIR`
`r_address`	`0x0`—the high 16 bits of the expression (`_bar - L1$pb`)
`r_value`	`0x8`—the address of the symbol `L1$pb`

Relocations in the x86-64 Environment

Relocations in the Mac OS X x86-64 environment are different than relocations in other Mac OS X environments and System V x86-64 (http://www.x86-64.org/documentation). The main differences are:.

- Scattered relocations are not used
- Compiler-generated code uses mostly external relocations
- Mach Object (Mach-O), not Executable and Linkable Format (ELF), is used as the executable file format

This section describes how relocations are implemented in the Mac OS X x86-64 environment.

When the assembler generates relocations, if the target label is a local label (it begins with `L`), the previous nonlocal label in the same section is used as the target of the external relocation. An addend (that is, the `4` in `_foo + 4`) is used with the distance from that nonlocal label to the target label. The assembler uses an internal relocation only when there is no previous nonlocal label in the section.

The addend is encoded in the instruction (Mach-O does not have `RELA` relocations). For PC-relative relocations, the addend is stored in the instruction. This practice is different than in other Mac OS X environments, which encode the addend minus the current section offset. The x86-64 relocation types are described in *Mac OS X ABI Mach-O File Format Reference*.

Listing 3 shows assembly instructions and the relocation and section content that they generate.

Listing 3 Example assembly instructions and their corresponding relocations

```
call    _foo
r_type=X86_64_RELOC_BRANCH, r_length=2, r_extern=1, r_pcrel=1, r_symbolnum=_foo
E8 00 00 00 00

call    _foo+4
r_type=X86_64_RELOC_BRANCH, r_length=2, r_extern=1, r_pcrel=1, r_symbolnum=_foo
E8 04 00 00 00

movq _foo@GOTPCREL(%rip), %rax
r_type=X86_64_RELOC_GOT_LOAD, r_length=2, r_extern=1, r_pcrel=1, r_symbolnum=_foo
48 8B 05 00 00 00 00

pushq _foo@GOTPCREL(%rip)
r_type=X86_64_RELOC_GOT, r_length=2, r_extern=1, r_pcrel=1, r_symbolnum=_foo
FF 35 00 00 00 00

movl _foo(%rip), %eax
r_type=X86_64_RELOC_SIGNED, r_length=2, r_extern=1, r_pcrel=1, r_symbolnum=_foo
8B 05 00 00 00 00

movl _foo+4(%rip), %eax
r_type=X86_64_RELOC_SIGNED, r_length=2, r_extern=1, r_pcrel=1, r_symbolnum=_foo
8B 05 04 00 00 00

movb  $0x12, _foo(%rip)
r_type=X86_64_RELOC_SIGNED, r_length=2, r_extern=1, r_pcrel=1, r_symbolnum=_foo
C6 05 FF FF FF FF 12

movl  $0x12345678, _foo(%rip)
r_type=X86_64_RELOC_SIGNED, r_length=2, r_extern=1, r_pcrel=1, r_symbolnum=_foo
C7 05 FC FF FF FF 78 56 34 12

.quad _foo
r_type=X86_64_RELOC_UNSIGNED,r_length=3, r_extern=1,r_pcrel=0, r_symbolnum=_foo
00 00 00 00 00 00 00 00

.quad _foo+4
r_type=X86_64_RELOC_UNSIGNED,r_length=3,r_extern=1,r_pcrel=0,r_symbolnum=_foo
04 00 00 00 00 00 00 00

.quad _foo - _bar
```

```
r_type=X86_64_RELOC_SUBTRACTOR,r_length=3,r_extern=1, r_pcrel=0,r_symbolnum=_bar
r_type=X86_64_RELOC_UNSIGNED,r_length=3,r_extern=1, r_pcrel=0,r_symbolnum=_foo
00 00 00 00 00 00 00 00

.quad _foo - _bar + 4
r_type=X86_64_RELOC_SUBTRACTOR,r_length=3, r_extern=1,r_pcrel=0,r_symbolnum=_bar
r_type=X86_64_RELOC_UNSIGNED,r_length=3, r_extern=1,r_pcrel=0,r_symbolnum=_foo
04 00 00 00 00 00 00 00

.long _foo - _bar
r_type=X86_64_RELOC_SUBTRACTOR,r_length=2,r_extern=1,r_pcrel=0,r_symbolnum=_bar
r_type=X86_64_RELOC_UNSIGNED,r_length=2,r_extern=1,r_pcrel=0,r_symbolnum=_foo
00 00 00 00

lea L1(%rip), %rax
r_type=X86_64_RELOC_SIGNED, r_length=2, r_extern=1, r_pcrel=1, r_symbolnum=_prev
48 8d 05 12 00 00 00
// Assumes that _prev is the first nonlocal label 0x12 bytes before L1.

lea L0(%rip), %rax
r_type=X86_64_RELOC_SIGNED, r_length=2, r_extern=0, r_pcrel=1, r_symbolnum=3
48 8d 05 56 00 00 00
// Assumes that  L0 is in third section, and has an address of 0x00000056
// in .o file, and no previous nonlocal label.

.quad L1
r_type=X86_64_RELOC_UNSIGNED,r_length=3,r_extern=1,r_pcrel=0, r_symbolnum= _prev
12 00 00 00 00 00 00 00
// Assumes that _prev is the first nonlocal label 0x12 bytes before L1.

.quad L0
r_type=X86_64_RELOC_UNSIGNED,r_length=3, r_extern=0, r_pcrel=0, r_symbolnum= 3
56 00 00 00 00 00 00 00
// Assumes that L0 is in third section, and has address of 0x00000056
// in .o file, and no previous nonlocal label.

.quad _foo - .
r_type=X86_64_RELOC_SUBTRACTOR,r_length=3,r_extern=1,r_pcrel=0,r_symbolnum=_prev
r_type=X86_64_RELOC_UNSIGNED,r_length=3,r_extern=1,r_pcrel=0,r_symbolnum=_foo
EE FF FF FF FF FF FF FF
// Assumes that _prev is the first nonlocal label 0x12 bytes
// before this .quad

.quad _foo - L1
r_type=X86_64_RELOC_SUBTRACTOR,r_length=3,r_extern=1,r_pcrel=0,r_symbolnum=_prev
r_type=X86_64_RELOC_UNSIGNED,r_length=3,r_extern=1,r_pcrel=0,r_symbolnum=_foo
EE FF FF FF FF FF FF FF
// Assumes that  _prev is the first nonlocal label 0x12 bytes before L1.

.quad L1 - _prev
// No relocations. This is an assembly time constant.
12 00 00 00 00 00 00 00
// Assumes that _prev is the first nonlocal label 0x12 bytes before L
```

x86-64 Code Model

This article describes differences in the Mac OS X x86-64 user-space code model from the code model described in *System V Application Binary Interface AMD64 Architecture Processor Supplement*, at http://www.x86-64.org/documentation.

The x86-64 environment in Mac OS X has only one code model for user-space code. It's most similar to the small PIC model defined by the x86-64 System V ABI. Under Mach-O, all static initialized storage (both code and data) must fit within a 4GB Mach-O file. Uninitialized (zero-fill) data may be any size, although there is a practical limit imposed by Mac OS X.

All local and small data is accessed directly using addressing that's relative to the instruction pointer (RIP-relative addressing). All large or possibly nonlocal data is accessed indirectly through a global offset table (GOT) entry. The GOT entry is accessed directly using RIP-relative addressing.

Listing 1 shows sample C code and corresponding assemble code.

Listing 1 C code and the corresponding assembly code

```
extern int src[];
extern int dst[];
extern int* ptr;

static int lsrc[500];
static int ldst[500];
static int bsrc[500000];
static int bdst[500000];
static int* lptr;

dst[0] = src[0];        movq _src@GOTPCREL(%rip), %rax
                        movl (%rax), %edx
                        movq _dst@GOTPCREL(%rip), %rax
                        movl %edx, (%rax)

ptr = dst;              movq _dst@GOTPCREL(%rip), %rdx
                        movq _ptr@GOTPCREL(%rip), %rax
                        movq %rdx, (%rax)
                        ret

*ptr = src[0];          movq  _ptr@GOTPCREL(%rip), %rax
                        movq   (%rax), %rdx
                        movq   _src@GOTPCREL(%rip), %rax
                        movl   (%rax), %eax
                        movl   %eax, (%rdx)
                        ret

ldst[0] = lsrc[0];      movl _lsrc(%rip), %eax
                        movl %eax, _ldst(%rip)

lptr = ldst;            lea  _ldst(%rip), %rax
```

```
                        movq %rax, _lptr(%rip)

*lptr = lsrc[0];        movl _lsrc(%rip), %edx
                        movq _lptr(%rip), %rax
                        movl %edx, (%rax)

bdst[0] = bsrc[0];      movq _bsrc@GOTPCREL(%rip), %rax
                        movl (%rax), %edx
                        movq _bdst@GOTPCREL(%rip), %rax
                        movl %edx, (%rax)

lptr = bdst;            movq _bdst@GOTPCREL(%rip), %rax
                        movq %rax, _lptr(%rip)

*lptr = bsrc[0];        movq _bsrc@GOTPCREL(%rip), %rdx
                        movq _lptr(%rip), %rax
                        movl (%rdx), %edx
```

The Mac OS X x86-64 code-generation model accesses large local data through the GOT, which is different from the way the small PIC model works in the System V x86-64 environment. Indirection through the GOT obviates the need for a medium code model. This behavior stems from the fact that, when the linker lays out data, it places data that is accessed directly (small local data and the GOT itself) within 2 GB of the code. Other data can be placed farther away because these data are accessed only through the GOT. This behavior enables large (greater than 4 GB) and small executables to be built using the same code model.

Note: It is acceptable for the compiler to access static data through a GOT entry. The linker preserves the static semantics of the symbol.

The code model for function calls is very simple, as shown in Listing 2.

Listing 2 The code model for function calls

```
extern int foo();
static int bar();

foo();                   call _foo

bar();                   call _bar
```

All direct function calls are made using the `CALL rel32` instruction.

The linker is responsible for creating GOT entries (also known as nonlazy pointers) as well as stub functions and lazy pointers (also known as program load table entries, or PLT entries) for calls to another linkage unit. Since the linker must create these entries, it can also choose not to create them when it sees the opportunity. The linker has a complicated set of rules that dictate which symbols must be accessed indirectly (depending on flat versus two-level namespace, weak versus nonweak definitions, symbol visibility, distance from code, and so on). But ultimately there are many symbols that can be accessed directly (not through GOT or PLT entries). For these symbols the linker makes the following optimization:

1. A `CALL` or `JMP` instruction performs a direct, PC-relative branch to the target.
2. A load instruction performed on a GOT entry (for example, `movq _foo@GOTPCREL(%rip), %rxx`) is transformed into a `LEA` calculation (for example, `leaq _foo(%rip), %rxx`). This transformation removes one GOT entry and saves one memory load.

In both cases special relocations are used that allow the linker to perform this optimization.

Document Revision History

This table describes the changes to *Mach-O Programming Topics*.

Date	Notes
2009-02-04	Made minor changes.
2006-11-28	Added details about the Mac OS X x86-64 environment.
	Added details about the user-space code model in "x86-64 Code Model" (page 43).
	Added details about symbol references to "x86-64 Symbol References" (page 34).
	Added details about symbol relocations in "Relocations in the x86-64 Environment" (page 39).
2006-11-07	Added information on the IA-32 symbol stubs and the DWARF debugging format.
	Added information about the DWARF debugging format to "Scope and Treatment of Symbol Definitions" (page 21).
	Added information on the stubs used for indirect addressing in the IA-32 environment in "Indirect Addressing" (page 31).
2005-11-09	Added the "Dynamic Code Generation" article from content previously published in "PowerPC Runtime Architecture Guide."
2005-08-11	Clarified terminology for binaries that contain object files for more than one architecture.
2005-06-04	Updated for Mac OS X v10.4.
2005-04-29	New document that describes basic concepts about the Mac OS X runtime environment. Replaces information that was published previously in "Mach-O Runtime Architecture."
2004-08-31	Corrected sample of a private external symbol in "Scope and Treatment of Symbol Definitions" (page 21) in Executing Mach-O Files (page 17).
	Corrected framework-building example in Listing 1 (page 27).

Index

A

application package 13

B

binding 18
bundles 13, 28

C

CFPlugin object 29
Classic runtime environment 7
coalesced symbol 22
Cocoa framework 17
Code Fragment Manager 7
Code Fragment Manager, using in Mac OS X 7
Code Fragment Manager, using with Carbon 29
COM objects 29

D

dependent libraries 18
DLL. *See* dynamic shared libraries
dyld tool 17
dynamic linker 17, 18
dynamic shared libraries 25

E

errno variable 18
execve function 17
external symbol 21

F

-flat_namespace flag 29
fork function 17
frameworks 13

H

HotSpot Java virtual machine 7

I

install name 26
intermediate object files 12

J

Java virtual machine 7
just-in-time binding 19

L

LaunchCFMApp tool 7
lazy binding 19
load commands 18
load-time binding 19

M

mach_header data structure 18
main function 18
module 14

N

NSBundle class 29

O

object file image functions 29

P

plug-in. *See* bundles
prebinding 19
Preferred Executable Format (PEF) 7, 29
private defined symbol 21

S

shared libraries 13
shared libraries, and versioning 26
static archive libraries 13
static archive library 14
symbol 19

T

tentative symbol 21
two-level namespace hint table 20
two-level symbol namespace 20

U

umbrella framework 28
umbrella frameworks 13
/usr/lib/crt1.o 18

W

weak binding 19