Mach-O Runtime Architecture

for Mac OS X version 10.3

2004-08-31

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

Simultaneously published in the United States and Canada.

Contents

Tables, Figures, and Listings

Introduction to Mach-O Runtime Architecture

A **runtime architecture** is a set of rules that define the software environment. Authors of compilers and other development tools must follow the definition of the runtime architecture to guarantee that programs released by different developers work with each other. A runtime architecture typically specifies:

- How to address code and data
- How to load and keep track of portions of program code in memory
- How compilers should generate code
- How to invoke certain system services, such as loading of application plug-ins

Mac OS X supports a number of application environments, each with its own runtime rules, conventions, and file formats. The only executable format the Mac OS X kernel reads directly is the Mach-O file format, which gives the Mach-O runtime architecture its name. In Mac OS X, kernel extensions, command-line tools, applications, frameworks, and libraries (shared and static) are implemented using Mach-O files.

The following list describes other runtime environments supported by 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), 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 describes the dynamic Mach-O runtime environment, including the data formats and calling conventions to which applications must adhere to successfully interoperate with other Mach-O applications and code libraries.

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 describes the Mach-O runtime linking architecture and calling conventions of Mac OS X. The individual chapters discuss the following topics:

- "Overview of the Runtime Architecture" (page 11) describes the basics of the Mac OS X runtime architecture, including detailed conceptual information about how Mach-O executable files are built, linked, and executed. This chapter also explains how dynamic linking works with shared libraries, frameworks, bundles and plug-ins. All programmers who create shared libraries or who dynamically load code at runtime should read this chapter.
- "Runtime Conventions for PowerPC" (page 27) describes the Mac OS X application binary interface for PowerPC microprocessors, which specifies low-level routine calling conventions and data formats. Writers of assembly language code and authors of developer tools should read this chapter.
- "Mach-O File Format Reference" (page 47) describes the layout of the Mach-O file format, used for executables, shared libraries, bundles, and all other native executable machine code in the Mach-O runtime architecture. Authors of developer tools should read this chapter.

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 points out what a developer tool vendor may need to create a Mach-O-based development environment for a procedural language such as C. It does not address the following:

- *Mach-O Runtime Reference* describes the Mach-O low-level programming interface. If you are loading code at runtime but cannot or do not wish to use CFBundle or NSBundle, you should refer to this document.

- 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 Darwin Documentation for more information.

For additional documentation on the standard Mac OS X developer tools, see Tools Documentation.

Source code from the Darwin project can be downloaded from http://developer.apple.com/darwin/. The source code for all of the Mac OS X linking and compiling tools is available in the following Darwin subprojects:

- `gcc3`—The Mac OS X compiler for the C, C++, and Objective-C languages, based on the GNU Compiler Collection version 3.1 (as of this writing). This is the standard compiler for Mac OS X v10.2 and later.
- `gcc`—The Mac OS X compiler for the C, C++, and Objective-C languages, based on the GNU Compiler Collection version 2.95.2. This is the standard compiler for Mac OS X v10.1 and earlier versions.
- `cctools`—The Mac OS X static linker, dynamic linker, and related tools for examining and manipulating Mach-O files and static archive libraries.

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

- *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.
- *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.
- *PowerPC Numerics* in Performance Documentation. Describes the Mac OS X numerics environment.

Overview of the Runtime Architecture

This chapter discusses how you use the Mach-O runtime architecture. It describes the types of programs you can build, how programs are loaded and executed, the ways in which 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 chapter.

The Mach-O file format provides both intermediate (during the build process) and final (after linking the final product) storage of machine code and data. It was designed as a flexible replacement for the BSD `a.out` format, to be used by the compiler and the static linker and to contain statically linked executable code at runtime. Features for dynamic linking were added as the goals of the system evolved, resulting in a single file format for both statically linked and dynamically linked code.

A Mach-O file contains three primary regions of data: A header, a set of load commands, and raw segment data. The header and load commands describe the features, layout, and linking characteristics of the file. The segment data contains raw data for the segments that are defined in the load commands. The complete format is described in "Mach-O File Format Reference" (page 47).

Building Mach-O Files

The following sections loosely describe how Mac OS X programs are built, and discusses, in depth, the types of programs that you can build:

- "The Tools—Building and Running Mach-O Files" (page 12) describes the tools involved in the Mach-O-file build process.
- "The Products—Types of Mach-O Files You Can Build" (page 13) describes the types of Mach-O files that you can build.
- "Modules—The Smallest Unit of Code" (page 14) explains the role of the smallest indivisible unit of code within a Mach-O shared library.

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 dynamic linker into a new process and executes it. The dynamic linker loads the program and all the frameworks and shared libraries that the program uses.

Throughout this book, 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 13)).

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 and facilitate communication about these topics with other Mac OS X developers. The standard tools include the following:

- The compiler driver, `/usr/bin/cc` (or, in Mac OS X v10.2 and later, `/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 Mach-O file analyzer, `/usr/bin/otool`, lists the contents of specific sections and segments within a Mach-O file. It includes symbolic disassemblers for each supported CPU architecture and it knows how to format the contents of many common section types.
- The symbol table display tool, `/usr/bin/nm`, allows you to view the contents of a Mach-O 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 files. All these types of files contain code that conforms to the Mach-O file format and runtime calling conventions.

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 15) for details. Other Mach-O files that contain executable code include:

- **Intermediate object files**are not final products; they are the basic building blocks of larger Mach-O 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, and some development tools may not use the Mach-O format to store intermediate code and data.
- **Dynamic shared libraries** 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 is usable by many applications. See "Using Shared Libraries and Frameworks" (page 21) for more information.
- **Frameworks** are shared libraries that are packaged with associated resources, such as graphics files, developer documentation, and programming interfaces. See "Using Shared Libraries and Frameworks" (page 21) for more information.
- **Umbrella frameworks** 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 21) for more information.
- **Static archive libraries** 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, 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** 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 Mach-O file containing the executable code

- A directory containing the Mach-O file and associated resources. A bundle need not contain a Mach-O file. For more information on bundles, see *Bundles*.

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

See "Loading Plug-In Code With Bundles" (page 25) for more information.

- **Kernel extensions** are statically bound Mach-O object 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 user-space runtime, so it is not covered in this document.

To function properly in Mac OS X, all Mach-O 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 Mach-O 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 Mach-O 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: Again, please note that Mac OS X `libtool` is not GNU `libtool`.

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 "Static Archive Libraries" (page 80).

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 libthings.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.

Executing Mach-O Files

This section 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, then loads and executes the dynamic linker shared library, `/usr/lib/dyld`, in the program's address space. The dynamic linker then loads the program and the libraries it references. This process is described in detail in the next sections:

- "Launching an Application" (page 15) describes the application-launching process.
- "Forking and Executing the Process" (page 16) lists the system calls you can use to run a program and how the dynamic linker sets up the program's runtime environment.
- "Finding Imported Symbols" (page 16) explains how the dynamic linker links the imported symbols of a file with the appropriate library or framework.

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`. `fork` creates a process; `execve` 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 Concepts and Tasks*.

Forking and Executing the Process

To create a process using BSD system calls, your process must call the fork system call. fork creates a logical copy of your process, then returns the new process ID 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. (fork 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/bin/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 then 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. The settings used to build the client program affect this process, as explained in the following sections:

- "Binding Symbols" (page 17) describes the process of binding the imported symbols in one Mach-O file to their definitions in other Mach-O files. The static linker allows you to specify a number of different ways to perform the binding process.
- "Searching for Symbols" (page 18) describes the process of finding a symbol. For each imported symbol the dynamic linker finds, it uses this process to determine the location of the symbol. Programs can also explicitly find symbols using one of the APIs described in "Loading Plug-In Code With Bundles" (page 25).

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 any particular shared library the first time it binds a reference from that shared library.
- 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 see 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 Mach-O 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 non-weak symbol. The equivalent Mach-O feature is the "weak definition"—see "Scope and Treatment of Symbol Definitions" (page 19) 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 a 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 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 Mac OS X v10.1 static linker defaults to a two-level namespace for all Mach-O files.

Note: The Mach-O 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
```

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 `NSLookupAndBindSymbol` (*Mach-O Runtime Reference*).

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

Scope and Treatment of Symbol Definitions

Symbols in a Mach-O 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 Mach-O file, including functions and data. The following C code defines an external symbol:

```
int x = 0;
```

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 shared library cannot have common symbols. 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 of 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 Mach-O 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` parameter).

You can mark a symbol as private external by using the `__private_extern__` keyword (which works only in C), as in this example:

```
__private_extern__ int x = 0;
```

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 only be used 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, you should use the `weak_import` attribute on a function prototype, as demonstrated by the following code:

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

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.

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 Mach-O equivalent feature.

A **weak definition** is a symbol that is ignored by the linker if an otherwise identical but non-weak 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 remaining weak definitions into non-weak definitions, so this is only a concern for intermediate object files and static libraries that you wish to deploy on older system releases.

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 currently generate debugging symbols in the **stabs** debugging format, which is documented in the GDB debugger internals documentation (see "See Also" (page 8)). Debugging symbols, like other symbols, are stored in the symbol table (see "Symbol Table and Related Data Structures" (page 68)).

Loading Code At Runtime

This section describes how you can load code at runtime:

- "Using Shared Libraries and Frameworks" (page 21) lists the benefits of using shared libraries and explains how they are packaged inside frameworks.
- "Loading Plug-In Code With Bundles" (page 25) describes how the Mach-O runtime 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.

Shared libraries allow the operating system as a whole to use memory more 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.

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.

Client Program Compatibility

This section describes various parameters that affect compatibility with client programs. You can set these parameters at build time.

Shared libraries have two version numbers, which allow you to create new 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 revision 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.

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).

You can pass the `-dynamic` option to `libtool` to create a dynamic shared library. The following command creates a dynamic shared library named `libthing.dylib` from a set of intermediate object files, `thing1.o` and `thing2.o`:

```
libtool -dynamic thing1.o thing2.o-o libthing.dylib
```

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 compile 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 graphics (the standard folder names for these are called `Headers`, `Documentation`, and `Resources`, respectively).

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 build a new, incompatible version of your framework, 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 wish.

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 1-1 build 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 1-1 Building a framework

```
cc -c -o Peace.o Peace.c
cc -c -o Love.o Love.c
mkdir -p Bliss.framework/Versions/A
cc -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 1-2 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 1-2 Building a private framework

```
mkdir -p Bliss.framework/Versions/A
cc -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
```

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 are useful to 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 developers 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 "Mach-O File Format Reference" (page 47). The particular load commands referenced here are documented as `sub_framework_command` (page 66) and `sub_client_command` (page 67), and `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 1-3.

Listing 1-3 Building a simple umbrella framework

```
mkdir Foundation.framework
ld -dylib -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.

Because an umbrella framework is a framework, you can use the directory-based versioning strategy described in "Packaging a Shared Library as a Framework" (page 22).

Loading Plug-In Code With Bundles

Bundles provide the Mach-O 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 NSBundle, 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** to load code fragments updated for Carbon from PEF files. For more information, see the Code Fragment Manager documentation ("See Also" (page 8)).
- For simpler needs, use the `dyld` library **object file image** functions and the `dyld`**low-level functions** to load and link bundle files. When porting UNIX tools that support plug-ins to Mac OS X, you usually want to use these two sets of functions. See "Object File Image Functions" and "Low-Level Dynamic Linking Functions" in *Mach-O Runtime 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. So, by default, the static linker creates bundles that are compatible with Mac OS X v10.0 by not including the two-level namespace hint table. You can use the `-twolevel_namespace_hints` option to ask the static linker to include the two-level hint table. The resulting bundle can be used only with Mac OS X v10.1 and later.

CFBundle and CFPlugin can both be used from Carbon applications running in both Mac OS 9 and Mac OS X. Both NSBundle and CFPlugin 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 on Mac OS X, CFPlugin uses the object file image `dyld` library functions.

For more information on NSBundle, see "Bundles" in *Loading Resources*. For more information on Code Fragment Manager, see *Mac OS Runtime Architectures*. For more information on CFPlugin and COM, see Core Foundation Documentation.

Runtime Conventions for PowerPC

This chapter covers specific low-level details of the Mac OS X PowerPC runtime architecture.

Together, these details specify the Mach-O PowerPC Application Binary Interface (ABI). If you are writing PowerPC assembly code or creating Mac OS X development tools, you should understand and conform to these conventions to ensure compatibility with the other programs running in the Mach-O runtime architecture.

Data Types

Table 2-1 lists the scalar binary data types and their sizes in the Mach-O PowerPC runtime environment.

Table 2-1 Scalar data types in the Mach-O PowerPC runtime environment

C or C++ type	Size (in bytes)	Value range
`unsigned char`	1	0 to 255
`char signed char`	1	–128 to 127
`unsigned short`	2	0 to 65,535
`signed short`	2	–32,768 to 32,767
`_Bool bool`	4	0 or 1 (false or true)
`unsigned int` `unsigned long`	4	0 to 4,294,967,295
`int signed int` `signed long`	4	-2,147,483,648 to 2,147,483,647
`unsigned long long`	8	0 to 18,446,744,073,709, 551,615
`signed long long`	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

C or C++ type	Size (in bytes)	Value range
`float`	4	See *PowerPC Numerics* ("See Also" (page 8))
`double`	8	See *PowerPC Numerics* ("See Also" (page 8))
`long double`	See notes below	
pointer	4	0 to 0xFFFFFFFF

Table 2-2 lists the binary vector types available in the Mach-O PowerPC runtime environment with Velocity Engine (AltiVec).

Table 2-2 Vector data types in the Mach-O PowerPC runtime environment

AltiVec C or C++ type	Size (in bytes)	Value range for each unit
`vector unsigned char`	16 (1 byte each)	0 to 255
`vector char` `vector signed char`	16 (1 byte each)	–128 to 127
`vector unsigned short`	16 (2 bytes each)	0 to 65,535
`vector signed short`	16 (2 bytes each)	–32,768 to 32,767
`vector unsigned int`	16 (4 bytes each)	0 to 4,294,967,295
`vector int` `vector signed int`	16 (4 bytes each)	-2,147,483,648 to 2,147,483,647
`vector bool char`	16 (1 bytes each)	0 (false), 1 (true)
`vector bool short`	16 (2 bytes each)	0 (false), 1 (true)
`vector bool int`	16 (4 bytes each)	0 (false), 1 (true)
`vector float`	16 (4 bytes each)	See *PPC Numerics*
`vector pixel`	16 (2 bytes each)	1/5/5/5 pixel format

Here are some things to note about PowerPC data types:

- As in most CPU architectures, a byte is 8 bits long, and a null pointer has a value of zero.
- All floating point types conform to the IEEE-754 standard representation. For the value range and precise format of floating point data types, see *PowerPC Numerics* ("See Also" (page 8)).
- PowerPC processors use the big-endian format to store numeric and pointer data types—most significant bytes first, then least significant bytes.
- PowerPC processors use two's-complement binary representation for signed integer types.

- On 32-bit PowerPC processors, arithmetic for the 64-bit integer data types (`long long`) must be implemented by the compiler (using math library routines), since the CPU itself does not implement 64-bit integer math operations.
- The `long double` extended-precision type is 16 bytes on classic Mac OS, but GCC 3.3 for PowerPC treats it as 8 bytes, equivalent to a `double`. A future revision of the compiler may extend `long double` to 128 bytes. For this reason, it is not currently recommended that you use `long double` on Mac OS X.
- Vector types are available only on CPUs that implement AltiVec execution units.

Data Alignment

The PowerPC runtime environment supports multiple data alignment modes. Alignment of data types falls into two categories:

- The **natural alignment**, which is the alignment of a data type when allocated in memory or assigned a memory address
- The **embedding alignment**, which is the alignment of a data type within a composite data structure

For example, the alignment of an `unsigned short` variable on the stack may differ from that of an `unsigned short` data item embedded in a data structure.

Note: Data items passed as parameters in a function call have their own special alignment rules. See "Routine Calls" (page 34), for more information.

The natural alignment of a data type is the size of the type; Table 2-1 (page 27) shows the size of each data type supported by the PowerPC runtime architecture.

In data structures, you can specify an embedding alignment that varies depending on the alignment mode selected. You can typically select the alignment mode using compiler options or pragmas. Your particular choice of mode is determined by compatibility and performance concerns, as detailed for each mode in the following list:

- **Power alignment mode** is derived from the alignment rules used by the IBM XLC compiler for the AIX operating system. It is the default alignment mode for Apple's PPCC and MrC compilers for classic Mac OS, as well as the default for the PowerPC version of GCC used on AIX and Mac OS X. Because this mode is most likely to be compatible between PowerPC compilers from different vendors, you typically use it with data structures that are shared between different programs. The rules for power alignment are:
  - The embedding alignment of the first element in a data structure is equal to the element's natural alignment.
  - For subsequent elements with a natural alignment less than 4, the embedding alignment of each element is equal to its natural alignment.
  - For subsequent elements that have a natural alignment greater than 4 bytes, the embedding alignment is 4, unless the element is a `vector` data type.
  - The embedding alignment for `vector` data types is always 16 bytes.

- The embedding alignment of a composite type (array or data structure) is determined by the largest embedding alignment of its members.
- The total size of a composite type is rounded up to a multiple of its embedding alignment, and is padded with null bytes.

Be careful when defining data structures with `double` and `long long` data types in power alignment mode. Because these types have natural alignments greater than 4 bytes, they may not be appropriately aligned, which impairs performance when such data members are accessed. If you use these data types for any element after the first element, be sure to place padding in the data structure to align these elements to their natural alignment, or use natural alignment mode instead.

- **Mac68k alignment mode** is derived from the alignment rules used by the MPW compilers for classic Mac OS. This alignment mode is usually used with legacy data structures inherited from classic Mac OS. New code should not need to use this alignment mode except to preserve compatibility with older data structures. The rules for mac68k alignment are:
  - The embedding alignment of a `char` type is 1 byte.
  - The embedding alignment of all other types other than vector types is 2 bytes.
  - The embedding alignment for vector data types is 16 bytes.
  - The total size of a composite data type is rounded up to a multiple of 2 bytes.
- **Natural alignment mode** uses the natural alignment of each data type as its embedding alignment. Use this mode for highest performance when working with `double`, `long long`, and `long double` data types.
- **Packed alignment mode** contains no alignment padding between elements (the embedding alignment for all elements is 1 byte). Use this mode when you need a data structure to be compressed as small as possible in memory.

Table 2-3 compares the embedding alignment for each data type in each of the alignment modes.

Table 2-3 Embedded alignment modes (in bytes)

C data type	Power	68K	Packed	Natural
`char`	1	1	1	1
`short`	2	2	1	2
`long`	4	2	1	4
`_Bool`	4	2	1	4
`float`	4	2	1	4
`double`	4 or 8	2	1	8
`long long`	8	2	1	8
All `vector` types	16	16	1	16

C data type	Power	68K	Packed	Natural
Composite (data structure or array)	4, 8, or 16	2	1	1, 2, 4, 8, or 16

With the standard C compiler, you can control data structure alignment by adding pragma statements to your source code, or by using parameters on the command line. The power alignment mode is used if you do not specify otherwise.

To change the default alignment at the command line, you can pass the options `-malign-power`, `-malign-mac68k`, and `-malign-natural` to the compiler. To enable a particular alignment mode for a data structure, place an alignment pragma statement of the following form before the data structure:

```
#pragma option align=mode
```

where *mode* is `power`, `mac68k`, `natural`, or `packed`.

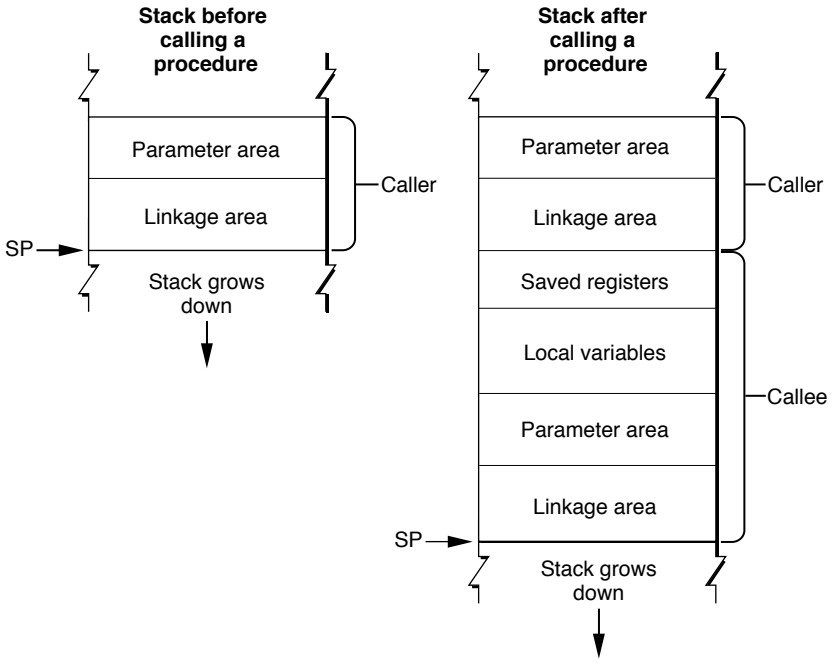
Alignment modes are nested. To restore the previous alignment mode, use `reset`, as follows:

```
#pragma option align=reset
```

Stack Structure

The PowerPC runtime environment uses a grow-down stack that contains linkage information, local variables, and a routine's parameter information, as shown in Figure 2-1.

Figure 2-1 The PowerPC stack



The PowerPC stack conventions use only a stack pointer (held in register GPR1) and no frame pointer. This configuration assumes a fixed stack frame size, which is known at compile time. Parameters are not passed by pushing them onto the stack.

The calling routine's stack frame includes a parameter area and some linkage information. The **parameter area** has space for the parameters of any routines the caller calls (not the parameters of the caller itself). Since the calling routine might call several routines, the parameter area must be large enough to accommodate the largest parameter list of all the routines the caller calls. It is the calling routine's responsibility to set up the parameter area before each call to some other routine, and the called routine's responsibility for accessing the parameters placed within it.

The calling routine's **linkage area** holds a number of values, some of which are saved by the calling routine and some by the called routine. The elements within the linkage area are:

- **The Link Register (LR)** value is saved at 8(SP) by the called routine if it chooses to do so.
- **The Condition Register (CR)** value may be saved at 4(SP) by the called routine. As with the Link Register value, the called routine is not required to save this value.
- **The stack pointer** is always saved by the calling routine as part of its stack frame.

Note that the linkage area is at the top of the stack, adjacent to the stack pointer. This positioning is necessary so the calling routine can find and restore the values stored there and also to enable the called routine to find the caller's parameter area. This placement means that a routine cannot push and pop parameters from the stack once the stack frame is set up.

The stack frame also includes space for the called routine's local variables. In general, the general-purpose registers GPR13 through GPR31, the floating-point registers FPR14 through FPR31, and vector registers v0, v1, and v14 through v31 are reserved for the routine's local variables. However, if the routine contains more local variables than would fit in the registers, it uses additional space on the stack. The size of the local variable area is determined at compile time. Once a stack frame is allocated, the size of the local variable area cannot change.

Prologs and Epilogs

The called routine is responsible for allocating its own stack frame, making sure to preserve 16-byte alignment on the stack. This action is accomplished by a section of code called the **prolog,** which the compiler places before the body of the routine. After the body of the routine, the compiler generates an **epilog** to restore the processor to the state it was prior to the prolog.

The compiler-generated prolog code does the following:

- Decrements the stack pointer to account for the new stack frame.
- Writes the previous value of the stack pointer to its own linkage area. This procedure ensures the stack can be restored to its original state after returning from the call.
- Saves all nonvolatile general-purpose and floating-point registers into the saved-registers area. Note that if the called routine does not change a particular nonvolatile register, it does not save it.
- Saves the Link Register and Condition Register values in the caller's linkage area, if needed.

These actions need not be executed in any particular order. Listing 2-1 shows an example of a routine prolog. Note that the order of these actions differs from the order previously described.

Listing 2-1 Example PowerPC assembler prolog code

```
linkageArea: set 24                     # size in PowerPC environment
params: set 32                          # callee parameter area
localVars: set 0                        # callee local variables
numGPRs: set 0                          # volatile GPRs used by callee
numFPRs: set 0                          # volatile FPRs used by callee)

spaceToSave: set linkageArea + params + localVars
spaceToSave: set spaceToSave + 4*numGPRs + 8*numFPRs

.functionName:                        # PROLOG
    mflr        r0,                     # extract return address
    stw         r0,8(SP)                # save the return address
    stwu        SP, -spaceToSave(SP)    # skip over caller save
```

At the end of the function, the compiler-generated epilog does the following:

- Restores the nonvolatile general-purpose and floating-point registers that were saved in the stack frame
- Restores the Condition Register and Link Register values that were stored in the linkage area
- Restores the stack pointer to its previous value
- Returns to the calling routine using the address stored in the Link Register

Again, these actions need not be executed in any particular order. Listing 2-2 shows an example PowerPC routine epilog.

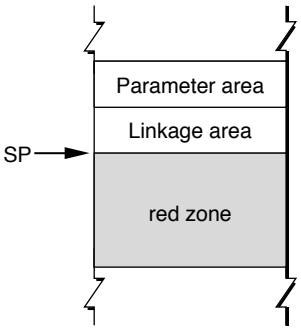
Listing 2-2 Example PowerPC routine epilog

```
                                  ; EPILOG
    lwz         r0,spaceToSave(SP)+8    # get the return address
    mtlr        R0                      #   into the Link Register
    addic       SP,SP,spaceToSave       # restore stack pointer
    blr                                 # and branch to the return address
```

The Red Zone

The space beneath the stack pointer, where a new stack frame would normally be allocated, is called the **red zone**. This area, as shown in Figure 2-2, may be used for any purpose as long as a new stack frame does not need to be added to the stack.

Figure 2-2 The red zone



For example, the red zone may be used by a **leaf procedure**. A leaf procedure is a routine that does not call any other routines. Since it does not call any other routines, it does not need to allocate a parameter area on the stack. Furthermore, if it does not need to use the stack to store local variables, it need save and restore only the nonvolatile registers that it uses for local variables. Since by definition no more than one leaf procedure is active at any time, there is no possibility of multiple leaf procedures competing for the same red zone space.

A leaf procedure neither allocates a stack frame nor decrements the stack pointer. Instead, it stores the Link Register and Condition Register values in the linkage area of the routine that calls it (if necessary) and stores the values of any nonvolatile registers it uses in the red zone. This streamlining means that a leaf procedure's prolog and epilog do only minimal work; they do not have to set up and take down a stack frame.

Note: The value of 224 bytes is the space occupied by nineteen 32-bit general-purpose registers plus eighteen 64-bit floating-point registers, rounded up to the nearest 16-byte boundary. If a leaf procedure's red zone usage would exceed 224 bytes, it must set up a stack frame just like routines that call other routines.

Routine Calls

This section details the process of passing parameters to a routine in the PowerPC runtime environment. See "Dynamic Code Generation" (page 41) for information about generating indirect calls and position-independent code.

Note: These parameter-passing conventions are part of Apple's standard for procedural interfaces. Object-oriented languages may use different rules for their own method calls. For example, the conventions for C++ virtual function calls may be different from those for C functions.

Parameter Passing

A routine can have a fixed or variable number of arguments. In an ANSI-style C syntax definition, a routine with a variable number of arguments typically appears with ellipsis points (…) at the end of its parameter list.

A variable-argument routine may have several required (that is, fixed) parameters preceding the variable parameter portion. For example, the routine definition:

```
(void) fooColor(int number, ...)
```

gives no restriction on the number of arguments after `number`, but you precede them with the `number` argument. Therefore, `number` is a fixed parameter.

Typically, the calling routine passes parameters in registers. However, the compiler generates a parameter area in the caller's stack frame that is large enough to hold all parameters passed to the called routine, regardless of how many of the parameters are actually passed in registers. There are several reasons for this scheme:

- It provides the callee with space to store a register-based parameter if it wants to use one of the parameter registers for some other purpose (for instance, to pass parameters to a subroutine).
- Routines with variable-length parameter lists must often access their parameters from RAM, not from registers. Such routines must reserve eight registers (32 bytes) in the parameter area to hold the parameter values.
- To simplify debugging, some compilers may write parameters from the parameter registers into the parameter area in the stack frame. This allows you to see all the parameters by looking only at that parameter area.

You can think of the parameter area as a data structure that has space to hold all the parameters in a given call. The parameters are conceptually assigned a location in the structure from left to right according to these rules:

- All non-vector parameters are aligned on 4-byte (word) boundaries.
- Noncomposite parameters (that is, parameters that are not arrays or data structures) smaller than 4 bytes occupy the high-order bytes of their word.
- Composite parameters (arrays or data structures) 1 or 2 bytes in size are preceded by padding to 4 bytes.

  This rule is inconsistent with other PowerPC ABIs. In AIX and classic Mac OS, padding bytes always follow the structure data even in the case of composite parameters smaller than 4 bytes.
- Composite parameters 3 bytes or larger in size are followed by padding to make a multiple of 4 bytes, with the padding bytes being undefined. (You should use zero, however.)

The caller doesn't necessarily place any parameters in the first 8 words of the parameter area, except composite parameters whose size is 3 bytes or more and not divisible by 4 (for example, 3, 5, 6, 7, 9, and so on). The callee assumes this behavior and obtains such parameters from the stack instead of the registers. If the callee needs to have other parameters in the parameter area, it must place them there itself.

For a routine with fixed parameters, the first 8 words (32 bytes) of the parameters, no matter the size of the individual parameters, are passed in registers according to the following rules:

- The first 8 words are placed in GPR3 through GPR10 unless a floating-point parameter is encountered.
- Floating-point parameters are placed in the floating-point registers FPR1 through FPR13.

- If a floating-point parameter appears before all the general-purpose registers are filled, the corresponding GPRs that match the size of the floating-point parameter are skipped. For example, a `float` item causes one (4-byte) GPR to be skipped; an item of type `double` causes two GPRs to be skipped.
- Structures (values of type `struct`) with only one noncomposite member are placed in GPRs or FPRs (depending on whether they're integers or floating-point values). For example, a structure composed only of a `float` is placed in an FPR, not a GPR.
- If the number of parameters exceeds the number of usable registers, the calling routine places the excess parameters in the parameter area of its stack frame.
- The caller places `vector` parameters in vector registers v2 through v13. For routines with a fixed number of parameters, the presence of vectors does not affect the allocation of GPRs and FPRs. The caller should not allocate space for vector register values in the parameter area of the stack, unless the number of vector parameters exceeds the number of available vector registers.

For example, consider a routine `fooFunc` with this declaration:

```
void fooFunc (SInt32 i1, float f1, double d1, SInt16 s1, double d2,
                        UInt8 c1, UInt16 s2, float f2, SInt32 i2);
```

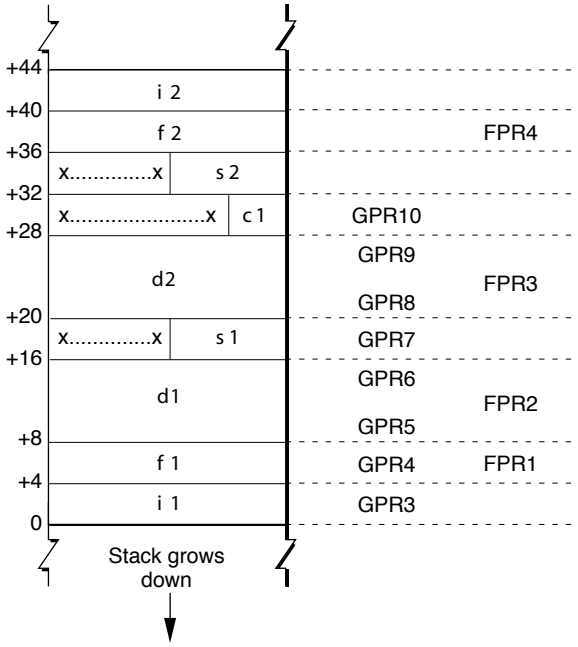
To see how the parameters of `fooFunc` are arranged in the parameter area on the stack, first convert the parameter list into the following structure:

```
struct params {
    SInt32                      p_i1;
    float                       p_f1;
    double                      p_d1;
    SInt16                      p_s1;
    double                      p_d2;
    UInt8                       p_c1;
    UInt16                      p_s2;
    float                       p_f2;
    SInt32                      p_i2;
};
```

This structure serves as a template for constructing the parameter area on the stack. (Remember that, in actual practice, many of these variables are passed in registers; nonetheless, the compiler still allocates space for all of them on the stack, for the reasons just mentioned.)

The "top" position on the stack is for the field `pi_1` (the structure field corresponding to parameter `i1`). The floating-point field `p_f1` is assigned to the next word in the parameter area. The 64-bit double field `p_d1` is assigned to the next two words in the parameter area. Next, the short integer field `p_s1` is placed in the following 32-bit word; the original value of `p_s1` is in the lower half of the word, and the padding is in the upper half. The remaining fields of the `params` structure are assigned space on the stack in exactly the same way, with unsigned values being extended to fill each field to make it a 32-bit word. The final arrangement of the stack is illustrated in Figure 2-3. (Because the stack grows down, it makes the fields of the params structure appear upside down.)

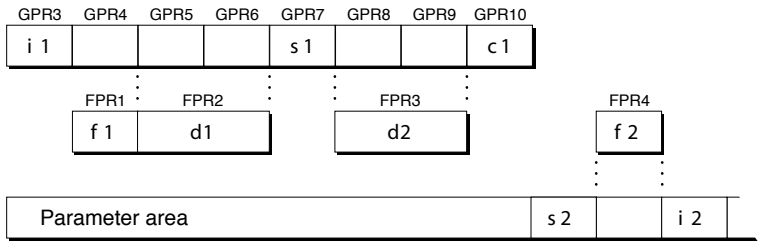
Figure 2-3 The organization of the parameter area of the stack



To see which parameters are passed in registers and which are passed on the stack, you need to map the stack, as illustrated in Figure 2-3, to the available general-purpose and floating-point registers. Therefore, the parameter `i1` is passed in GPR3, the first available general-purpose register. The floating-point parameter `f1` is passed in FPR1, the first available floating-point register. This action causes GPR4 to be skipped.

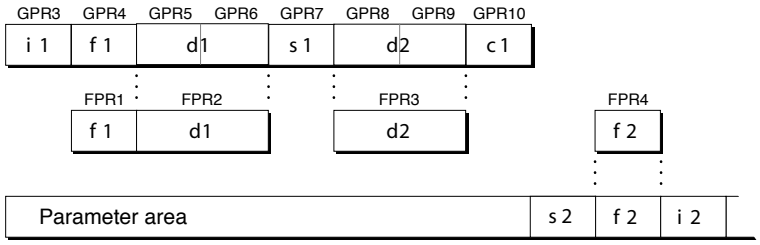
The parameter `d1` is placed into FPR2 and the corresponding general-purpose registers GPR5 and GPR6 are unused. The parameter `s1` is placed into the next available general-purpose register, GPR7. Parameter `d2` is placed into FPR3, with GPR8 and GPR9 masked out. Parameter `c1` is placed into GPR10, which fills out the first 8 words of the data structure. Parameter `s2` is then passed in the parameter area of the stack. Parameter `f2` is passed in FPR4, since there are floating-point registers available. Finally, parameter `i2` is placed on the stack. Figure 2-4 shows the final layout of the parameters in the registers and the parameter area.

Figure 2-4 Parameter layout in registers and the parameter area



If you have a C routine with a variable number of parameters (that is, one that does not have a fixed prototype), the compiler cannot know whether to pass a parameter in the variable portion of the routine, in the general-purpose (that is, fixed-point) registers, or in the floating-point registers. Therefore, the compiler passes the parameter in both the floating-point and the general-purpose registers, as shown in Figure 2-5.

Figure 2-5 Passing a variable number of parameters



The called routine can access parameters in the fixed portion of the routine definition as usual. However, in the variable-argument portion of the routine, the callee must copy the GPRs to the parameter area and access the values from there. Listing 2-3 shows a routine that accesses values by walking through the stack.

Listing 2-3 A variable-argument routine

```
double dsum (int count, ...)
{
    double sum = 0.0;
    double * arg = (double *) (&count + 1 /* pointer arithmetic */);
    while (count > 0 ) {
        sum += *arg;
        arg += 1;                    /* pointer arithmetic */
        count -= 1;
    }
return sum;
}
```

Vector parameters in the fixed portion of the routine definition are passed in v2 through v13. For functions with variable arguments only, they are also shadowed on the stack, where they must be aligned to a 16-byte boundary. Vector parameters that appear in the variable-argument portion of the routine must also be shadowed in the GPRs.

Function Return

In the PowerPC runtime environment, functions return floating-point values in register FPR1. Other values are returned as follows:

- Functions returning simple values smaller than 4 bytes (such as the GCC C compiler types `char` and `short`) place the return value in the least significant byte or bytes of GPR3. The most significant bytes in GPR3 are undefined.
- Functions returning 4-byte values (such as pointers, including array pointers, or GCC C compiler types `long` and `int`) return them normally in GPR3.

- Functions returning `long long` values place the return value in GPR3 (the 4 high-order bytes) and GPR4 (the 4 low-order bytes).
- If a function returns a composite value (for example, a `struct` or `union` data type) or a value larger than 4 bytes, a pointer must be passed as an implicit left-most parameter before passing all the user-visible arguments (that is, the address is passed in GPR3, and the actual parameters begin with GPR4). The address of the pointer must be a memory location large enough to hold the function return value. Since GPR3 is treated as a parameter in this case, its value is not guaranteed on return. Note that Mac OS X differs from PowerPC Linux in how they handle 64-bit composite values. In PowerPC Linux, those values are stored in GPR3 and GPR4.

Register Preservation

Table 2-4 lists registers used in the PowerPC runtime environment and their volatility in routine calls. Registers that retain their value after a routine call are called **nonvolatile**. All registers are 4 bytes long.

Table 2-4 Volatile and nonvolatile registers

Types	Register	Preserved by a routine call (nonvolatile)	Notes
General-purpose register	GPR0	No	
	GPR1	Yes	Used as the stack pointer to store parameters and other temporary data items.
	GPR2	No	On other PowerPC platforms, including Mac OS 9, GPR2 is usually a pointer to the current TOC. Mac OS X uses a different indirect addressing scheme, and GPR2 is thus considered a volatile register available for general use.
	GPR3	No	The caller passes parameter values to the called function in GPR3 through GPR10. The callee should place the return value, if any, in GPR3.
	GPR4–GPR10	No	Used to pass parameter values in routine calls (see notes for GPR3).
	GPR11	Yes for nested functions. No for nonnested functions	In nested functions, the caller passes its stack frame to the nested function in this register. In nonnested functions, the register is available. For details on nested functions, see GCC documentation.

Types	Register	Preserved by a routine call (nonvolatile)	Notes
	GPR12	No	Set to the address of the branch target before an indirect call for dynamic code generation. This register is not set for a routine that has been called directly, so routines that may be called directly should not depend on this register being set up correctly. See "Indirect Addressing" (page 43) for more information.
	GPR13–GPR31	Yes	
Floating- point register	FPR0	No	
	FPR1–FPR13	No	Used to pass floating- point parameters in routine calls.
	FPR14–FPR31	Yes	
Vector register	v0–v19	No	The caller passes vector parameters in v2 to v13 during a routine call.
	v20–31	Yes	
Vector special purpose	VRSAVE	Yes	32-bit special purpose register. Set bits for each vector register that must be saved during a thread or process context switch.
Link Register	LR	No	Stores the return address of the calling routine during a routine call.
Count Register	CTR	No	
Fixed-point exception register	XER	No	
Condition Registers	CR0–CR1	No	
	CR2–CR4	Yes	
	CR5–CR7	No	

Dynamic Code Generation

To support dynamically bound shared libraries, applications, bundles, the compiler tools, and the dynamic linker support two features: Position-independent code (abbreviated PIC) and indirect addressing.

Position-Independent Code

Position-independent code, or **PIC**, is the name of the code generation technique that allows the dynamic linker to load a region of code at a different virtual memory addresses. 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 linker, 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 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.

Position-independent code is typically required for shared libraries and bundles to allow the dynamic linker 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. Apple's version of GCC 3.1 introduces a new option, called `-mdynamic-no-pic`, to reduce the code size of application executables by eliminating position-independent code references, while preserving indirect calls to shared libraries. 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 2-7 (page 44).

Note: Dynamic code generation without PIC was introduced in GCC 3.1, the standard compiler shipped with Mac OS X v10.2. However, executables generated with this option run on earlier versions of Mac OS X, as long as they do not rely on incompatible features of the Mac OS X v10.2 tools (such as weak references).

Listing 2-5 shows an example of the position-independent code generated for the C code in Listing 2-4.

Listing 2-4 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 2-5 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 "Relocation Data Structures" (page 76).

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` (page 77)):

`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

Indirect Addressing

Indirect addressing is the name of the code generation technique, separate from position-independent code, that allows symbols defined in one file to be referenced from another file, without requiring the first file to have explicit knowledge of the layout of the second. This allows the second file to be modified independently of the first.

When generating calls to functions that are defined in other files, the compiler creates a symbol stub and a lazy symbol pointer. The **symbol stub** is a small amount of code that directly dereferences and jumps to the 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`. `dyld_stub_binding_helper` 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 example code in Listing 2-6 might produce two different types of symbol stubs, depending on whether it is compiled with position-independent code generation. Listing 2-7 shows indirect addressing without position-independent code, while Listing 2-8 (page 45) shows both indirect addressing and position-independent code.

Listing 2-6 Example C code for indirect function calls

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

Listing 2-7 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$lazy_ptr)(r11)   ; load the value of bar's lazy pointer
                                            ;  into r12
        mtctr r12                           ; move r2 to the count register
        addi r11,r11,lo16(L_bar$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 pointerp to the stub
                                            ;  binding helper address
```

Listing 2-8 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 linkn 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
                                                ;  L1$_bar + distance
        addis r11,r11,ha16(L_bar$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 2-8 (page 45) resembles the position-independent code generated for Listing 2-5 (page 42). 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 the GPR12 register. Standardizing the register 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.

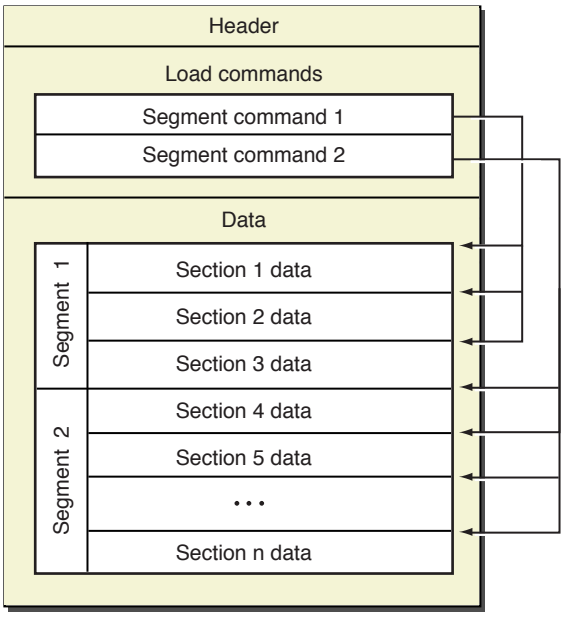
Mach-O File Format Reference

This chapter describes the structure of the Mach-O executable file format, which is the standard used to store programs on disk in the Mach-O runtime architecture. To understand how the Xcode tools work with Mach-O files, and to perform low-level debugging tasks, you need to understand this information.

A Mach-O file contains three major regions (as shown in Figure 3-1):

- At the beginning of every Mach-O file is a **header structure** that identifies the file as a Mach-O executable file. The header also contains other basic file type information, indicates the target CPU architecture, and contains flags specifying options that affect the interpretation of the rest of the file.
- Directly following the header are a series of variable-size **load commands** that specify the layout and linkage characteristics of the Mach-O file. Among other information, the load commands can specify:
  - The initial layout of the file in virtual memory
  - The location of the symbol table (used for both dynamic linking and debugging information)
  - The initial execution state of the main thread of the program
  - The names of shared libraries that contain definitions for the main executable's imported symbols
- Following the load commands, all Mach-O files contain the data of one or more segments. Each **segment** contains zero or more sections. Each **section** of a segment contains code or data of some particular type. Each segment defines a region of virtual memory that the dynamic linker maps into the address space of the process. The exact number and layout of segments and sections is specified by the load commands and the file type.

Figure 3-1 Mach-O file format basic structure



Various tables within a Mach-O file refer to sections by number. Section numbering begins at 1 (not zero) and continues across segment boundaries. Thus, the first segment in a file may contain sections 1 and 2 and the second segment may contain sections 3 and 4.

A Mach-O file contains code and data for one CPU architecture. The header structure of a Mach-O file specifies the target CPU architecture, which allows the kernel to ensure that, for example, binary machine code intended for PowerPC processors is not executed on an x86 processor. You can store Mach-O files for multiple CPU architectures in one file using the format described in "Multi-CPU Architecture Files" (page 82).

Segments and sections are normally accessed by name. Segments, by convention, are named using all uppercase letters preceded by two underscores (for example, `__TEXT`); sections should be named using all lowercase letters preceded by two underscores (for example, `__text`). This naming convention is standard, although not required for the tools to operate correctly.

A segment defines a range of bytes in a Mach-O file and the addresses and memory protection attributes at which those bytes are mapped into virtual memory when the dynamic linker loads the application. As such, segments are always virtual memory page aligned.

Segments that require more memory at runtime than they do at build time can specify a larger in-memory size than they actually have on disk. For example, the `__PAGEZERO` segment generated by the linker for PowerPC executable files has a virtual memory size of one page but an on-disk size of zero. Because `__PAGEZERO` contains no data, there is no need for it to occupy any space in the executable file.

A segment contains zero or more sections. For performance reasons, sections that are to be filled with zeros should always be placed at the end of the segment.

For compactness, an intermediate object file contains only one segment. This segment has no name; it contains all the sections destined ultimately for different segments in the final Mach-O file. The data structure that defines a `section` (page 56) contains the name of the segment the section is intended for, and the static linker places each section in the final Mach-O file accordingly.

For best performance, segments should be aligned on virtual memory page boundaries—4096 bytes for PowerPC processors and 8192 bytes for x86 processors. To calculate the size of a segment, add up the size of each section, then round up the sum to the next virtual memory page boundary (4096 bytes, or 4 kilobytes). Using this algorithm, the minimum size of a segment is 4 kilobytes, and thereafter it is sized at 4 kilobyte increments.

The header and load commands are considered part of the first segment of the file for paging purposes. In an executable file, this generally means that the headers and load commands live at the start of the `__TEXT` segment because that is the first segment that contains data. The `__PAGEZERO` segment contains no data on disk, so it's ignored for this purpose.

The standard Mac OS X development tools add five segment types to a typical Mac OS X executable:

- The static linker creates a `__PAGEZERO` segment as the first segment of an executable file. This segment is located at virtual memory location zero and has no protection rights assigned, the combination of which causes accesses to `NULL`, a common C programming error, to immediately crash. The `__PAGEZERO` segment is the size of one full VM page for the current CPU architecture (for x86 and PowerPC, this is 4096 bytes or 0x1000 in hexadecimal). Because there is no data in the `__PAGEZERO` segment, it occupies no space in the file (the file size in the segment command is zero).
- The `__TEXT` segment contains executable code and other read-only data. To allow the kernel to map it directly from the executable into sharable memory, the static linker sets this segment's virtual memory permissions to disallow writing. When the segment is mapped into memory, it can be shared among all processes interested in its contents. (This is primarily used with frameworks, bundles, and shared libraries, but it is possible to run multiple copies of the same executable in Mac OS X, and this applies in that case as well.) The read-only attribute also means that the pages that make up the `__TEXT` segment never need to be written back to disk. When the kernel needs to free up physical memory, it can simply discard one or more `__TEXT` pages and re-read them from disk when they are next needed.
- The `__DATA` segment contains writable data. The static linker sets the virtual memory permissions of this segment to allow both reading and writing. Because it is writable, the `__DATA` segment of a framework or other shared library is logically copied for each process linking with the library. When memory pages such as those making up the `__DATA` segment are readable and writable, the kernel marks them copy-on-write; therefore when a process writes to one of these pages, that process receives its own private copy of the page.
- The `__OBJC` segment contains data used by the Objective-C language runtime support library.
- The `__LINKEDIT` segment contains raw data used by the dynamic linker, such as symbol, string, and relocation table entries.

The `__TEXT` and `__DATA` segments may contain a number of standard sections, listed in Table 3-1. The `__OBJC` segment contains a number of sections that are private to the Objective-C compiler. Note that the static linker and file analysis tools typically use the section type and attributes (instead of the section name) to determine how they should treat the section. The section name, type and attributes are explained further in the description of the `section` (page 56) data type.

Table 3-1 Typical sections in a Mach-O file

Segment and section name	Contents
`__TEXT,__text`	Executable machine code. The compiler places only executable code in this section; no tables or data of any sort are stored here.

Segment and section name	Contents
__TEXT,__cstring	Constant C strings. A C string is a sequence of non-null bytes that ends with a null byte ('\0'). The static linker coalesces constant C string values, removing duplicates, when building the final product.
__TEXT,__picsymbol_stub	Position-independent indirect symbol stubs. See "Indirect Addressing" (page 43) for more information.
__TEXT,__symbol_stub	Indirect symbol stubs. See "Indirect Addressing" (page 43) for more information.
__TEXT,__const	Initialized constant variables. The compiler places all data declared const in this section.
__TEXT,__literal4	4-byte literal values. The compiler places single-precision floating point constants in this section. The static linker coalesces these values, removing duplicates, when building the final product. With some CPU architectures, it's more efficient for the compiler to use immediate load instructions rather than adding to this section.
__TEXT,__literal8	8-byte literal values. The compiler places double-precision floating point constants in this section. The static linker coalesces these values, removing duplicates, when building the final product. With some CPU architectures, it's more efficient for the compiler to use immediate load instructions rather than adding to this section.
__DATA,__data	Initialized mutable variables, such as writable C strings and data arrays.
__DATA,__la_symbol_ptr	Lazy symbol pointers, which are indirect references to functions imported from a different file. See "Indirect Addressing" (page 43) for more information.
__DATA,__nl_symbol_ptr	Non-lazy symbol pointers, which are indirect references to data items imported from a different file. See "Indirect Addressing" (page 43) for more information.
__DATA,__dyld	Placeholder section used by the dynamic linker.
__DATA,__const	Unintialized constant variables.
__DATA,__mod_init_func	Module initialization functions. The C++ compiler places static constructors here.
__DATA,__mod_term_func	Module termination functions.
__DATA,__bss	Data for uninitialized static variables (for example, static int i;).
__DATA,__common	Uninitialized imported symbol definitions (for example, int i;) located in the global scope (outside of a function declaration).

Note: Compilers or any tools that create Mach-O files are free to define additional section names, which do not appear in Table 3-1.

Mach-O Types and Data Structures

This section describes the data types that compose a Mach-O file. Values for integer types in all Mach-O data structures are written using the host CPU's byte ordering scheme, except for `fat_header` (page 82) and `fat_arch` (page 83), which are written in big-endian byte order. All these data types can be found in `/usr/include/mach-o/loader.h`, unless otherwise specified in the description.

Mach-O Header Data Structure

`mach_header`

Specifies general attributes of the file.

```
struct mach_header
{
    unsigned long magic;
    cpu_type_t cputype;
    cpu_subtype_t cpusubtype;
    unsigned long filetype;
    unsigned long ncmds;
    unsigned long sizeofcmds;
    unsigned long flags;
};
```

Field Descriptions

`magic`

An integer containing a value identifying this file as a Mach-O executable file. Use the constant `MH_MAGIC` if the file is intended for use on a CPU with the same endianness as the computer on which the compiler is running. The constant `MH_CIGAM` can be used when the byte ordering scheme of the target machine is the reverse of the host CPU.

`cputype`

An integer indicating the CPU architecture you intend to use the file on. Appropriate values include:

- `CPU_TYPE_POWERPC` for PowerPC-architecture CPUs
- `CPU_TYPE_I386` for x86-architecture CPUs

`cpusubtype`

An integer specifying the exact model of the CPU. To run on all PowerPC or x86 processors supported by the Mac OS X kernel, this should be set to `CPU_SUBTYPE_POWERPC_ALL` or `CPU_SUBTYPE_I386_ALL`.

`filetype`

An integer indicating the usage and alignment of the file. Valid values for this field include:

- The `MH_OBJECT` file type is the format used for intermediate object files. It is a very compact format containing all its sections in one segment. The compiler and assembler usually create one `MH_OBJECT` file for each source code file. By convention, the file name extension for this format is `.o`.
- The `MH_EXECUTE` file type is the format used by standard executable programs.
- The `MH_BUNDLE` file type is the type typically used by code that you load at runtime (typically called bundles or plug-ins). By convention, the file name extension for this format is `.bundle`.
- The `MH_DYLIB` file type is for dynamic shared libraries. It contains some additional tables to support multiple modules. By convention, the file name extension for this format is `.dylib`, except for the main shared library of a framework, which does not usually have a file name extension.
- The `MH_PRELOAD` file type is an executable format used for special-purpose programs that are not loaded by the Mac OS X kernel, such as programs burned into programmable ROM chips. Do not confuse this file type with the `MH_PREBOUND` flag, which is a flag that the static linker sets in the header structure to mark a prebound image.
- The `MH_CORE` file type is used to store core files, which are traditionally created when a program crashes. Core files store the entire address space of a process at the time it crashed. You can later run `gdb` on the core file to figure out why the crash occurred.
- The `MH_DYLINKER` file type is the type of a dynamic linker shared library. This is the type that `dyld` is constructed from.

`ncmds`

An integer indicating the number of load commands following the header structure.

`sizeofcmds`

An integer indicating the number of bytes occupied by the load commands following the header structure.

flags

An integer containing a set of bit flags that indicate the state of certain optional features of the Mach-O file format. These are the masks you can use to manipulate this field:

- MH_NOUNDEFS—The object file contained no undefined references when it was built.
- MH_INCRLINK—The object file is the output of an incremental link against a base file and cannot be linked again.
- MH_DYLDLINK—The file is input for the dynamic linker and cannot be statically linked again.
- MH_BINDATLOAD—The dynamic linker should bind the undefined references when the file is loaded.
- MH_PREBOUND—The file's undefined references are prebound.
- MH_SPLIT_SEGS—The file has its read-only and read-write segments split.
- MH_TWOLEVEL—The image is using two-level namespace bindings.
- MH_FORCE_FLAT—The executable is forcing all images to use flat namespace bindings.
- MH_SUBSECTIONS_VIA_SYMBOLS—The sections of the object file can be divided into individual blocks. These blocks are dead-stripped if they are not used by other code. See "Dead-Code Stripping" in *Xcode Build System* for details.

Special Considerations

For all file types, except MH_OBJECT, segments must be aligned on page boundaries for the given CPU architecture: 4096 bytes for PowerPC processors and 8192 bytes for x86 processors. This allows the kernel to page virtual memory directly from the segment into the address space of the process. The header and load commands must be aligned as part of the data of the first segment stored on disk (which would be the __TEXT segment, in the file types described in filetype).

Load Command Data Structures

The load command structures are located directly after the header of the Mach-O file, and they specify both the logical structure of the file and the layout of the file in virtual memory. Each load command begins with fields that specify the command type and the size of the command data.

load_command

Contains fields that are common to all load commands.

```
struct load_command
{
    unsigned long cmd;
    unsigned long cmdsize;
};
```

Field Descriptions

cmd

An integer indicating the type of load command. Table 3-2 lists the valid load command types.

`cmdsize`

An integer specifying the total size in bytes of the load command data structure. Each load command structure contains a different set of data, depending on the load command type, so each might have a different size. The size must always be a multiple of 4. This means the `cmdsize` field must always divide evenly by 4. If the load command data does not divide evenly by 4, add bytes containing zeros to the end until it does.

Discussion

Table 3-2 lists the valid load command types, with links to the full data structures for each type.

Table 3-2 Mach-O load commands

Commands	Data structures	Purpose
`LC_SEGMENT`	`segment_command` (page 55)	Defines a segment of this file to be mapped into the address space of the process that loads this file. It also includes all the sections contained by the segment. See "Mach-O File Format Reference" (page 47).
`LC_SYMTAB`	`symtab_command` (page 68)	Specifies the symbol table for this file. This information is used by both static and dynamic linkers when linking the file, and also by debuggers to map symbols to the original source code files from which the symbols were generated.
`LC_DYSYMTAB`	`dysymtab_command` (page 72)	Specifies additional symbol table information used by the dynamic linker.
`LC_THREAD` `LC_UNIXTHREAD`	`thread_command` (page 64)	For an executable file, the `LC_UNIXTHREAD` command defines the initial thread state of the main thread of the process. `LC_THREAD` is similar to `LC_UNIXTHREAD` but does not cause the kernel to allocate a stack.
`LC_LOAD_DYLIB`	`dylib_command` (page 62)	Defines the name of a dynamic shared library that this file links against.
`LC_ID_DYLIB`	`dylib_command` (page 62)	Specifies the install name of a dynamic shared library.
`LC_PREBOUND_DYLIB`	`prebound_dylib_command` (page 63)	For a shared library that this executable is linked prebound against, specifies the modules in the shared library that are used.
`LC_LOAD_DYLINKER`	`dylinker_command` (page 63)	Specifies the dynamic linker that the kernel executes to load this file.
`LC_ID_DYLINKER`	`dylinker_command` (page 63)	Identifies this file as a dynamic linker.
`LC_ROUTINES`	`routines_command` (page 65)	Contains the offset of the shared library initialization routine (specified by the linker's `-init` option).
`LC_TWOLEVEL_HINTS`	`twolevel_hints_command` (page 59)	Contains the two-level namespace lookup hint table.
`LC_SUB_FRAMEWORK`	`sub_framework_command` (page 66)	Identifies this file as the implementation of a subframework of an umbrella framework. The name of the umbrella framework is stored in the string parameter.

Commands	Data structures	Purpose
LC_SUB_UMBRELLA	sub_umbrella_command (page 66)	Specifies a file that is a subumbrella of this umbrella framework.
LC_SUB_LIBRARY	sub_library_command (page 67)	Identifies this file as the implementation of a sublibrary of an umbrella framework. The name of the umbrella framework is stored in the string parameter. Note that Apple has not defined a supported location for sublibraries.
LC_SUB_CLIENT	sub_client_command (page 67)	A subframework can explicitly allow another framework or bundle to link against it by including an LC_SUB_CLIENT load command containing the name of the framework or a client name for a bundle.

segment_command

Segments are defined by the LC_SEGMENT load command, which specifies a range of bytes in the file that are to be mapped by the loader into the address space of a program.

```
struct segment_command
{
    unsigned long cmd;
    unsigned long cmdsize;
    char segname[16];
    unsigned long vmaddr;
    unsigned long vmsize;
    unsigned long fileoff;
    unsigned long filesize;
    vm_prot_t maxprot;
    vm_prot_t initprot;
    unsigned long nsects;
    unsigned long flags;
};
```

Field Descriptions

cmd

Common to all load command structures. Set to LC_SEGMENT for this structure.

cmdsize

Common to all load command structures. For this structure, set this field to sizeof(segment_command) plus the size of all the section data structures that follow (sizeof(segment_command + (sizeof(section) * segment->nsect)).

segname

A C string specifying the name of the segment. The value of this field can be any sequence of ASCII characters, although segment names defined by Apple begin with two underscores and consist of capital letters (as in __TEXT and __DATA). This field is fixed at 16 bytes in length.

vmaddr

Indicates the starting virtual memory address of this segment.

`vmsize`

Indicates the number of bytes of virtual memory occupied by this segment. See also the description of `filesize`, below.

`fileoff`

Indicates the offset in this file of the data to be mapped at `vmaddr`.

`filesize`

Indicates the number of bytes occupied by this segment on disk. For segments that require more memory at runtime than they do at build time, `vmsize` can be larger than `filesize`. For example, the `__PAGEZERO` segment generated by the linker for `MH_EXECUTABLE` files has a `vmsize` of 0x1000 but a `filesize` of zero. Because `__PAGEZERO` contains no data, there is no need for it to occupy any space until runtime. Also, the static linker often allocates uninitialized data at the end of the `__DATA` segment; in this case, the `vmsize` is larger than the `filesize`. The loader guarantees that any memory of this sort is initialized with zeros.

`maxprot`

Specifies the maximum permitted virtual memory protections of this segment.

`initprot`

Specifies the initial virtual memory protections of this segment.

`nsects`

Indicates the number of section data structures following this load command.

`flags`

Defines a set of flags that affect the loading of this segment:

- `SG_HIGHVM`—The file contents for this segment are for the high part of the virtual memory space; the low part is zero filled (for stacks in core files).
- `SG_NORELOC`—This segment has nothing that was relocated in it and nothing relocated to it. It may be safely replaced without relocation.

section

Directly following a `segment_command` data structure is an array of `section` data structures, with the exact count determined by the `nsects` field of the `segment_command` (page 55) structure.

```
struct section
{
    char sectname[16];
    char segname[16];
    unsigned long addr;
    unsigned long size;
    unsigned long offset;
    unsigned long align;
    unsigned long reloff;
    unsigned long nreloc;
    unsigned long flags;
    unsigned long reserved1;
    unsigned long reserved2;
};
```

Field Descriptions

`sectname`

A string specifying the name of this section. The value of this field can be any sequence of ASCII characters, although section names defined by Apple begin with two underscores and consist of lowercase letters (as in `__text` and `__data`). This field is fixed at 16 bytes in length.

`segname`

A string specifying the name of the segment that should eventually contain this section. For compactness, intermediate object files—files of type `MH_OBJECT`—contain only one segment, in which all sections are placed. The static linker places each section in the named segment when building the final product (any file that is not of type `MH_OBJECT`).

`addr`

An integer specifying the virtual memory address of this section.

`size`

An integer specifying the size in bytes of the virtual memory occupied by this section.

`offset`

An integer specifying the offset to this section in the file.

`align`

An integer specifying the section's byte alignment. Specify this as a power of two; for example, a section with 8-byte alignment would have an align value of 3 (2 to the 3rd power equals 8).

`reloff`

An integer specifying the file offset of the first relocation entry for this section.

`nreloc`

An integer specifying the number of relocation entries located at `reloff` for this section.

`flags`

An integer divided into two parts. The least significant 8 bits contain the section type, while the most significant 24 bits contain a set of flags that specify other attributes of the section. These types and flags are primarily used by the static linker and file analysis tools, such as `otool`, to determine how to modify or display the section. These are the possible types:

- `S_REGULAR`—This section has no particular type. The standard tools create a `__TEXT,__text` section of this type.
- `S_ZEROFILL`—Zero-fill-on-demand section—when this section is first read from or written to, each page within is automatically filled with bytes containing zero.
- `S_CSTRING_LITERALS`—This section contains only constant C strings. The standard tools create a `__TEXT,__cstring` section of this type.
- `S_4BYTE_LITERALS`—This section contains only constant values that are 4 bytes long. The standard tools create a `__TEXT,__literal4` section of this type.
- `S_8BYTE_LITERALS`—This section contains only constant values that are 8 bytes long. The standard tools create a `__TEXT,__literal8` section of this type.
- `S_LITERAL_POINTERS`—This section contains only pointers to constant values.
- `S_NON_LAZY_SYMBOL_POINTERS`—This section contains only non-lazy pointers to symbols. The standard tools create a section of the `__DATA,__nl_symbol_ptrs` section of this type.
- `S_LAZY_SYMBOL_POINTERS`—This section contains only lazy pointers to symbols. The standard tools create a `__DATA,__la_symbol_ptrs` section of this type.
- `S_SYMBOL_STUBS`——This section contains symbol stubs. The standard tools create `__TEXT,__symbol_stub` and `__TEXT,__picsymbol_stub` sections of this type. See "Indirect Addressing" (page 43) for more information.
- `S_MOD_INIT_FUNC_POINTERS`—This section contains pointers to module initialization functions. The standard tools create `__DATA,__mod_init_func` sections of this type.
- `S_MOD_TERM_FUNC_POINTERS`—This section contains pointers to module termination functions. The standard tools create `__DATA,__mod_term_func` sections of this type.
- `S_COALESCED`—This section contains symbols that are coalesced by the static linker and possibly the dynamic linker. More than one file may contain coalesced definitions of the same symbol without causing multiple-defined-symbol errors.

The following are the possible attributes of a section:

- `S_ATTR_PURE_INSTRUCTIONS`—This section contains only executable machine instructions. The standard tools set this flag for the sections `__TEXT,__text`, `__TEXT,__symbol_stub`, and `__TEXT,__picsymbol_stub`.
- `S_ATTR_NO_TOC`—This section contains coalesced symbols that must not be placed in the table of contents (`SYMDEF` member) of a static archive library.
- `S_ATTR_SOME_INSTRUCTIONS`—This section contains executable machine instructions and other data.
- `S_ATTR_EXT_RELOC`—This section contains references that must be relocated. These references refer to data that exists in other files (undefined symbols). To support external relocation, the maximum virtual memory protections of the segment that contains this section must allow both reading and writing.

- S_ATTR_LOC_RELOC—This section contains references that must be relocated. These references refer to data within this file.
- S_ATTR_STRIP_STATIC_SYMS—The static symbols in this section can be stripped if the MH_DYLDLINK flag of the image's mach_header (page 51) header structure is set.
- S_ATTR_NO_DEAD_STRIP—This section must not be dead-stripped. See "Dead-Code Stripping" in *Xcode Build System* for details.
- S_ATTR_LIVE_SUPPORT—This section must not be dead-stripped if they reference code that is live, but the reference is undetectable.

`reserved1`

An integer reserved for use with certain section types. For symbol pointer sections and symbol stubs sections that refer to indirect symbol table entries, this is the index into the indirect table for this section's entries. The number of entries is based on the section size divided by the size of the symbol pointer or stub. Otherwise this field is set to zero.

`reserved2`

For sections of type S_SYMBOL_STUBS, an integer specifying the size (in bytes) of the symbol stub entries contained in the section. Otherwise, this field is reserved for future use and should be set to zero.

Discussion

Each section in a Mach-O file has both a type and a set of attribute flags. In intermediate object files, the type and attributes determine how the static linker copies the sections into the final product. Object file analysis tools (such as otool) use the type and attributes to determine how to read and display the sections. The section type and attributes are not used by the dynamic linker. These are important variants of the symbol type and attributes as they apply to static linking:

- **Regular sections**. In a regular section, only one definition of an external symbol may exist in intermediate object files. The static linker returns an error if it finds any duplicate external symbol definitions.
- **Coalesced sections**. In the final product, the static linker retains only one instance of each symbol defined in coalesced sections. Some complex language features (such as C++ vtables and RTTI) require a definition of a particular symbol to be duplicated in every intermediate object file. This wastes a lot of space in the final product. To reduce the memory occupied by a program, the compiler can place symbol definitions in coalesced sections.
- **Coalesced sections with weak definitions** Weak symbol definitions may appear only in coalesced sections. When the static linker finds duplicate definitions for a symbol, it discards any coalesced symbol definition that has the weak definition attribute set (see nlist (page 69)). If there are no non-weak definitions, the first weak definition is used instead. This feature is designed to support C++ templates; it allows explicit template instantiations to override implicit ones. The C++ compiler places explicit definitions in a regular section, and it places implicit definitions in a coalesced section, marked as weak definitions. Intermediate object files (and thus static archive libraries) built with weak definitions can be used only with the static linker in Mac OS X v10.2 and later. Final products (applications and shared libraries) should not contain weak definitions, so they usually can be used on earlier versions of Mac OS X.

twolevel_hints_command

The data structure of a LC_TWOLEVEL_HINTS load command.

```
struct twolevel_hints_command
{
    unsigned long cmd;
    unsigned long cmdsize;
    unsigned long offset;
    unsigned long nhints;
};
```

Field Descriptions

`cmd`

Common to all load command structures. Set to `LC_TWOLEVEL_HINTS` for this structure.

`cmdsize`

Common to all load command structures. For this structure, set to `sizeof(twolevel_hints_command)`.

`offset`

An integer specifying the byte offset from the start of this file to an array of `twolevel_hint` (page 60) data structures, known as the two-level namespace hint table.

`nhints`

The number of `twolevel_hint` data structures located at `offset`.

Discussion

The static linker adds the `LC_TWOLEVEL_HINTS` load command and the two-level namespace hint table to the output file when building a two-level namespace image.

Special Considerations

By default, `ld` does not include the `LC_TWOLEVEL_HINTS` command or the two-level namespace hint table in an `MH_BUNDLE` file because the presence of this load command causes the version of the dynamic linker shipped with Mac OS X v10.0 to crash. If you know the code will run only on Mac OS X v10.1 and later, you should explicitly enable the two-level namespace hint table. See `-twolevel_namespace_hints` in the `ld` man page for more information.

twolevel_hint

Specifies an entry in the two-level namespace hint table.

```
struct twolevel_hint
{
    unsigned long isub_image:8,
        itoc:24;
};
```

Field Descriptions

`isub_image`

The subimage in which the symbol is defined. It is an index into the subimage list of the symbol's image (which is specified by the high eight bits of the `n_desc` field of the symbol data structure—see `nlist` (page 69)). If this field is zero, the symbol is assumed to be in the umbrella image itself. If the symbol is not from a umbrella framework or library, `isub_image` must be zero.

itoc

The symbol index into the table of contents of the image specified by the isub_image field.

Discussion

The two-level namespace hint table provides the dynamic linker with suggested positions to start searching for symbols in the libraries the current image is linked against.

Every undefined symbol (that is, every symbol of type N_UNDF or N_PBUD) in a two-level namespace image must have a corresponding entry in the two-level hint table, at the same index.

The static linker adds the LC_TWOLEVEL_HINTS load command and the two-level namespace hint table to the output file when building a two-level namespace image.

By default, the linker does not include the LC_TWOLEVEL_HINTS command or the two-level namespace hint table in an MH_BUNDLE file, because the presence of this load command causes the version of the dynamic linker shipped with Mac OS X v10.0 to crash. If you know the code will run only on Mac OS X v10.1 and later, you should explicitly enable the two-level namespace hints. See the linker documentation for more information.

lc_str

Defines a variable-length string.

```
union lc_str
{
    unsigned long offset;
    char *ptr;
};
```

Field Descriptions

offset

A long integer. A byte offset from the start of the load command that contains this string to the start of the string data.

ptr

A pointer to an array of bytes. At runtime, this pointer contains the virtual memory address of the string data.

Discussion

Load commands store variable-length data such as library names using the lc_str data structure. Unless otherwise specified, the data consists of a C string.

The data pointed to is stored just after the load command, and the size is added to the size of the load command. You can determine the size of the string by subtracting the size of the load command data structure from the cmdsize field of the load command data structure.

dylib

Defines the data used by the dynamic linker to match a shared library against the files that have linked to it. Used exclusively in the dylib_command (page 62) data structure.

```
struct dylib
{
    union lc_str  name;
```

```
    unsigned long timestamp;
    unsigned long current_version;
    unsigned long compatibility_version;
};
```

Field Descriptions

`name`

A data structure of type `lc_str` (page 61). Specifies the name of the shared library.

`timestamp`

The date and time when the shared library was built.

`current_version`

The current version of the shared library.

`compatibility_version`

The compatibility version of the shared library.

Discussion

In order for the dynamic linker to successfully link a file to a dynamic library at runtime, two criteria must be met:

- The name for the shared library must match exactly the install name previously recorded by the static linker in the file, or as modified by the various dynamic linker environment variables (see the `dyld` man page for more information.)
- The compatibility version for the shared library must be less than or equal to the compatibility version recorded by the static linker in the image.

The dynamic linker uses the timestamp to determine whether it can use the prebinding information. The current version is returned by the function `NSVersionOfRunTimeLibrary` to allow you to determine the version of the library your program is using.

dylib_command

The data structure for the `LC_LOAD_DYLIB` and `LC_ID_DYLIB` load commands.

```
struct dylib_command
{
    unsigned long cmd;
    unsigned long cmdsize;
    struct dylib dylib;
};
```

Field Descriptions

`cmd`

Common to all load command structures. For this structure, set to either `LC_LOAD_DYLIB` or `LC_ID_DYLIB`.

`cmdsize`

Common to all load command structures. For this structure, set to `sizeof(dylib_command)` plus the size of the data pointed to by the `name` field of the `dylib` field.

dylib

A data structure of type dylib (page 61). Specifies the attributes of the shared library.

Discussion

The static linker adds a LC_ID_DYLIB load command to shared libraries to identify the linking attributes of the library.

The static linker adds one LC_LOAD_DYLIB load command for each shared library that a file links against. All the LC_LOAD_DYLIB commands together form a list that is ordered according to location in the file, earliest LC_LOAD_DYLIB command first. For two-level namespace files, undefined symbol entries in the symbol table refer to their parent shared libraries by index into this list. The index is called a *library ordinal*, and it is stored in the n_desc field of the nlist (page 69) data structure.

dylinker_command

The data structure for the LC_LOAD_DYLINKER and LC_ID_DYLINKER load commands.

```
struct dylinker_command
{
    unsigned long cmd;
    unsigned long cmdsize;
    union lc_str name;
};
```

Field Descriptions

cmd

Common to all load command structures. For this structure, set to either LC_ID_DYLINKER or LC_LOAD_DYLINKER.

cmdsize

Common to all load command structures. For this structure, set to sizeof(dylinker_command), plus the size of the data pointed to by the name field.

name

A data structure of type lc_str (page 61). Specifies the name of the dynamic linker.

Discussion

Every executable file that is dynamically linked contains a LC_LOAD_DYLINKER command that specifies the name of the dynamic linker that the kernel must load in order to execute the file. The dynamic linker itself specifies its name using the LC_ID_DYLINKER load command.

prebound_dylib_command

The data structure for the LC_PREBOUND_DYLIB load command. For every library that a prebound executable file links to, the static linker adds one LC_PREBOUND_DYLIB command.

```
struct prebound_dylib_command
{
    unsigned long cmd;
    unsigned long cmdsize;
    union lc_str name;
    unsigned long nmodules;
    union lc_str linked_modules;
};
```

Field Descriptions

`cmd`

Common to all load command structures. For this structure, set to `LC_PREBOUND_DYLIB`.

`cmdsize`

Common to all load command structures. For this structure, set to `sizeof(prebound_dylib_command)` plus the size of the data pointed to by the `name` and `linked_modules` fields.

`name`

A data structure of type `lc_str` (page 61). Specifies the name of the prebound shared library.

`nmodules`

An integer. Specifies the number of modules the prebound shared library contains. The size of the `linked_modules` string is `(nmodules / 8) + (nmodules % 8)`.

`linked_modules`

A data structure of type `lc_str` (page 61). Usually, this data structure defines the offset of a C string; in this usage, it is a variable-length bitset, containing one bit for each module. Each bit represents whether the corresponding module is linked to a module in the current file, 1 for yes, zero for no. The bit for the first module is the low bit of the first byte.

thread_command

The data structure for the `LC_THREAD` and `LC_UNIXTHREAD` load commands. The data of this command is specific to each CPU architecture and appears in `thread_status.h`, located in the CPU's directory in `/usr/include/mach`.

```
struct thread_command
{
    unsigned long cmd;
    unsigned long cmdsize;
    /* unsigned long flavor;*/
    /* unsigned long count; */
    /* struct cpu_thread_state state;*/
};
```

Field Descriptions

`cmd`

Common to all load command structures. For this structure, set to `LC_THREAD` or `LC_UNIXTHREAD`.

`cmdsize`

Common to all load command structures. For this structure, set to `sizeof(thread_command)` plus the size of the `flavor` and `count` fields plus the size of the CPU-specific thread state data structure.

`flavor`

Integer specifying the particular flavor of the thread state data structure. See the `thread_status.h` file for your CPU architecture.

count

Size of the thread state data, in number of 32-bit integers. The thread state data structure must be fully padded to 32-bit alignment.

routines_command

The data structure for the LC_ROUTINES load command. Describes the location of the shared library initialization function, which is a function that the dynamic linker calls before allowing any of the routines in the library to be called.

```
struct routines_command
{
    unsigned long cmd;
    unsigned long cmdsize;
    unsigned long init_address;
    unsigned long init_module;
    unsigned long reserved1;
    unsigned long reserved2;
    unsigned long reserved3;
    unsigned long reserved4;
    unsigned long reserved5;
    unsigned long reserved6;
};
```

Field Descriptions

cmd

Common to all load command structures. For this structure, set to LC_ROUTINES.

cmdsize

Common to all load command structures. For this structure, set to sizeof(routines_command).

init_address

An integer specifying the virtual memory address of the initialization function.

init_module

An integer specifying the index into the module table of the module containing the initialization function.

reserved1

Reserved for future use. Set this field to zero.

reserved2

Reserved for future use. Set this field to zero.

reserved3

Reserved for future use. Set this field to zero.

reserved4

Reserved for future use. Set this field to zero.

`reserved5`

Reserved for future use. Set this field to zero.

`reserved6`

Reserved for future use. Set this field to zero.

Discussion

The static linker adds an `LC_ROUTINES` command when you specify a shared library initialization function using the `-init` option (see the `ld` man page for more information).

sub_framework_command

The data structure for the `LC_SUB_FRAMEWORK` load command. Identifies the umbrella framework of which this file is a subframework.

```
struct sub_framework_command
{
    unsigned long cmd;
    unsigned long cmdsize;
    union lc_str umbrella;
};
```

Field Descriptions

`cmd`

Common to all load command structures. For this structure, set to `LC_SUB_FRAMEWORK`.

`cmdsize`

Common to all load command structures. For this structure, set to `sizeof(sub_framework_command)` plus the size of the data pointed to by the `umbrella` field.

`umbrella`

A data structure of type `lc_str` (page 61). Specifies the name of the umbrella framework of which this file is a member.

sub_umbrella_command

The data structure for the `LC_SUB_UMBRELLA` load command. Identifies the named framework as a subumbrella of this framework. Unlike a subframework, any client may link to a subumbrella.

```
struct sub_umbrella_command
{
    unsigned long cmd;
    unsigned long cmdsize;
    union lc_str sub_umbrella;
};
```

Field Descriptions

`cmd`

Common to all load command structures. For this structure, set to `LC_SUB_UMBRELLA`.

cmdsize

Common to all load command structures. For this structure, set to `sizeof(sub_umbrella_command)` plus the size of the data pointed to by the `sub_umbrella` field.

sub_umbrella

A data structure of type `lc_str` (page 61). Specifies the name of the umbrella framework of which this file is a member.

sub_library_command

The data structure for the `LC_SUB_LIBRARY` load command. Identifies a sublibrary of this framework and marks this framework as an umbrella framework. Unlike a subframework, any client may link to a sublibrary.

```
struct sub_library_command
{
    unsigned long cmd;
    unsigned long cmdsize;
    union lc_str sub_library;
};
```

Field Descriptions

cmd

Common to all load command structures. For this structure, set to `LC_SUB_LIBRARY`.

cmdsize

Common to all load command structures. For this structure, set to `sizeof(sub_library_command)` plus the size of the data pointed to by the `sub_library` field.

sub_library

A data structure of type `lc_str` (page 61). Specifies the name of the sublibrary of which this file is a member.

sub_client_command

The data structure for the `LC_SUB_CLIENT` load command. Specifies the name of a file that is allowed to link to this subframework. This file would otherwise be required to link to the umbrella framework of which this file is a component.

```
struct sub_client_command
{
    unsigned long cmd;
    unsigned long cmdsize;
    union lc_str client;
};
```

Field Descriptions

cmd

Common to all load command structures. For this structure, set to `LC_SUB_CLIENT`.

`cmdsize`

Common to all load command structures. For this structure, set to `sizeof(sub_client_command)` plus the size of the data pointed to by the `client` field.

`client`

A data structure of type `lc_str` (page 61). Specifies the name of a client authorized to link to this library.

Special Considerations

`ld` generates a `sub_client_command` load command in the built product if you pass the option `-allowable_client_name name`, where *name* is the install name of a framework or the client name of a bundle. See the `ld` man page, specifically about the options `-allowable_client_name` and `-client_name`, for more information.

Symbol Table and Related Data Structures

Two load commands, `LC_SYMTAB` and `LC_DYSYMTAB`, describe the size and location of the symbol tables, along with additional metadata. The other data structures listed in this section represent the symbol tables themselves.

symtab_command

The data structure for the `LC_SYMTAB` load command. Describes the size and location of the symbol table data structures.

```
struct symtab_command
{
    unsigned long cmd;
    unsigned long cmdsize;
    unsigned long symoff;
    unsigned long nsyms;
    unsigned long stroff;
    unsigned long strsize;
};
```

Field Descriptions

`cmd`

Common to all load command structures. For this structure, set to `LC_SYMTAB`.

`cmdsize`

Common to all load command structures. For this structure, set to `sizeof(symtab_command)`.

`symoff`

An integer containing the byte offset from the start of the file to the location of the symbol table entries. The symbol table is an array of `nlist` (page 69) data structures.

`nsyms`

An integer indicating the number of entries in the symbol table.

`stroff`

An integer containing the byte offset from the start of the image to the location of the string table.

`strsize`

An integer indicating the size (in bytes) of the string table.

Discussion

`LC_SYMTAB` should exist in both statically linked and dynamically linked file types.

nlist

Describes an entry in the symbol table. It's declared in `/usr/include/mach-o/nlist.h`.

```
struct nlist
{
    union {
        char *n_name;
        long  n_strx;
    } n_un;
    unsigned char n_type;
    unsigned char n_sect;
    short n_desc;
    unsigned long n_value;
};
```

Field Descriptions

`n_un`

A union that holds an index into the string table, `n_strx`. To specify an empty string (`""`), set this value to zero. The `n_name` field is not used in Mach-O files.

`n_type`

A byte value consisting of data accessed using four bit masks:

- `N_STAB` (0xe0)—If any of these 3 bits are set, the symbol is a symbolic debugging table (`stab`) entry. In that case, the entire `n_type` field is interpreted as a `stab` value. See `/usr/include/mach-o/stab.h` for valid stab values.
- `N_PEXT` (0x10)—If this bit is on, this symbol is marked as having limited global scope. When the file is fed to the static linker, it clears the `N_EXT` bit for each symbol with the `N_PEXT` bit set. (The `ld` option `-keep_private_externs` turns off this behavior.) With Mac OS X GCC, you can use the `__private_extern__` function attribute to set this bit.
- `N_TYPE` (0x0e)—These bits define the type of the symbol.
- `N_EXT` (0x01)—If this bit is on, this symbol is an external symbol, a symbol that is either defined outside this file or that is defined in this file but can be referenced by other files.

Values for the `N_TYPE` field include:

- `N_UNDF` (0x0)—The symbol is undefined. Undefined symbols are symbols referenced in this module but defined in a different module. Set the `n_sect` field to `NO_SECT`.
- `N_ABS` (0x2)—The symbol is absolute. The linker does not update the value of an absolute symbol. Set the `n_sect` field to `NO_SECT`.
- `N_SECT` (0xe)—The symbol is defined in the section number given in `n_sect`.
- `N_PBUD` (0xc)—The symbol is undefined and the image is using a prebound value for the symbol. Set the `n_sect` field to `NO_SECT`.
- `N_INDR` (0xa)—The symbol is defined to be the same as another symbol. The `n_value` field is an index into the string table specifying the name of the other symbol. When that symbol is linked, both this and the other symbol point to the same defined type and value.

`n_sect`

An integer specifying the number of the section that this symbol can be found in, or `NO_SECT` if the symbol is not to be found in any section of this image. The sections are contiguously numbered across segments, starting from 1, according to the order they appear in the `LC_SEGMENT` load commands.

`n_desc`

A 16-bit value providing additional information about the nature of this symbol. The reference flags can be accessed using the `REFERENCE_TYPE` mask (0xF) and are defined as follows:

- `REFERENCE_FLAG_UNDEFINED_NON_LAZY` (0x0)—This symbol is a reference to an external non-lazy (data) symbol.
- `REFERENCE_FLAG_UNDEFINED_LAZY` (0x1)—This symbol is a reference to an external lazy symbol—that is, to a function call.
- `REFERENCE_FLAG_DEFINED` (0x2)—This symbol is defined in this module.
- `REFERENCE_FLAG_PRIVATE_DEFINED` (0x3)—This symbol is defined in this module and is visible only to modules within this shared library.
- `REFERENCE_FLAG_PRIVATE_UNDEFINED_NON_LAZY` (0x4)—This symbol is defined in another module in this file, is a non-lazy (data) symbol, and is visible only to modules within this shared library.
- `REFERENCE_FLAG_PRIVATE_UNDEFINED_LAZY` (0x5)—This symbol is defined in another module in this file, is a lazy (function) symbol, and is visible only to modules within this shared library.

Additionally, the following bits might also be set:

- `REFERENCED_DYNAMICALLY` (0x10)—Must be set for any symbol that might be referenced by another image. The `strip` tool uses this bit to avoid removing symbols that must exist: If the symbol has this bit set, `strip` does not strip it.
- `N_DESC_DISCARDED` (0x20)—Used by the dynamic linker at runtime. Do not set this bit.
- `N_WEAK_REF` (0x40)—Indicates that this symbol is a weak reference. If the dynamic linker cannot find a definition for this symbol, it sets the address of this symbol to zero. The static linker sets this symbol given the appropriate weak-linking flags.
- `N_WEAK_DEF` (0x80)—Indicates that this symbol is a weak definition. If the static linker or the dynamic linker finds another (non-weak) definition for this symbol, the weak definition is ignored. Only symbols in a coalesced `section` (page 56) can be marked as a weak definition.

If this file is a two-level namespace image (that is, if the `MH_TWOLEVEL` flag of the `mach_header` structure is set), the high 8 bits of `n_desc` specify the number of the library in which this symbol is defined. Use the macro `GET_LIBRARY_ORDINAL` to obtain this value and the macro `SET_LIBRARY_ORDINAL` to set it. Zero specifies the current image. 1 through 254 specify the library number according to the order of `LC_LOAD_DYLIB` commands in the file. For plug–ins that load symbols from the executable program they are linked against, 255 specifies the executable image. For flat namespace images, the high 8 bits must be zero.

`n_value`

An integer that contains the value of the symbol. The format of this value is different for each type of symbol table entry (as specified by the `n_type` field). For the `N_SECT` symbol type, `n_value` is the address of the symbol. See the description of the `n_type` field for information on other possible values.

Discussion

Common symbols must be of type `N_UNDF` and must have the `N_EXT` bit set. The `n_value` for a common symbol is the size (in bytes) of the data of the symbol. In C, a common symbol is a variable that is declared but not initialized in this file. Common symbols can appear only in `MH_OBJECT` Mach-O files.

dysymtab_command

The data structure for the `LC_DYSYMTAB` load command. It describes the sizes and locations of the parts of the symbol table used for dynamic linking.

```
struct dysymtab_command
{
    unsigned long cmd;
    unsigned long cmdsize;
    unsigned long ilocalsym;
    unsigned long nlocalsym;
    unsigned long iextdefsym;
    unsigned long nextdefsym;
    unsigned long iundefsym;
    unsigned long nundefsym;
    unsigned long tocoff;
    unsigned long ntoc;
    unsigned long modtaboff;
    unsigned long nmodtab;
    unsigned long extrefsymoff;
    unsigned long nextrefsyms;
    unsigned long indirectsymoff;
    unsigned long nindirectsyms;
    unsigned long extreloff;
    unsigned long nextrel;
    unsigned long locreloff;
    unsigned long nlocrel;
};
```

Field Descriptions

`cmd`

Common to all load command structures. For this structure, set to `LC_DYSYMTAB`.

`cmdsize`

Common to all load command structures. For this structure, set to `sizeof(dysymtab_command)`.

`ilocalsym`

An integer indicating the index of the first symbol in the group of local symbols.

`nlocalsym`

An integer indicating the total number of symbols in the group of local symbols.

`iextdefsym`

An integer indicating the index of the first symbol in the group of defined external symbols.

`nextdefsym`

An integer indicating the total number of symbols in the group of defined external symbols.

`iundefsym`
An integer indicating the index of the first symbol in the group of undefined external symbols.

`nundefsym`
An integer indicating the total number of symbols in the group of undefined external symbols.

`tocoff`
An integer indicating the byte offset from the start of the file to the table of contents data.

`ntoc`
An integer indicating the number of entries in the table of contents.

`modtaboff`
An integer indicating the byte offset from the start of the file to the module table data.

`nmodtab`
An integer indicating the number of entries in the module table.

`extrefsymoff`
An integer indicating the byte offset from the start of the file to the external reference table data.

`nextrefsyms`
An integer indicating the number of entries in the external reference table.

`indirectsymoff`
An integer indicating the byte offset from the start of the file to the indirect symbol table data.

`nindirectsyms`
An integer indicating the number of entries in the indirect symbol table.

`extreloff`
An integer indicating the byte offset from the start of the file to the external relocation table data.

`nextrel`
An integer indicating the number of entries in the external relocation table.

`locreloff`
An integer indicating the byte offset from the start of the file to the local relocation table data.

`nlocrel`
An integer indicating the number of entries in the local relocation table.

Discussion
The `LC_DYSYMTAB` load command contains a set of indexes into the symbol table and a set of file offsets that define the location of several other tables. Fields for tables not used in the file should be set to zero. These tables are described in "Indirect Addressing" (page 43).

dylib_table_of_contents

Describes an entry in the table of contents of a dynamic shared library.

```
struct dylib_table_of_contents
{
    unsigned long symbol_index;
    unsigned long module_index;
};
```

Field Descriptions

`symbol_index`

An index into the symbol table indicating the defined external symbol to which this entry refers.

`module_index`

An index into the module table indicating the module in which this defined external symbol is defined.

dylib_module

Describes a module table entry for a dynamic shared library.

```
struct dylib_module
{
    unsigned long module_name;
    unsigned long iextdefsym;
    unsigned long nextdefsym;
    unsigned long irefsym;
    unsigned long nrefsym;
    unsigned long ilocalsym;
    unsigned long nlocalsym;
    unsigned long iextrel;
    unsigned long nextrel;
    unsigned long iinit_iterm;
    unsigned long ninit_nterm;
    unsigned long objc_module_info_addr;
    unsigned long objc_module_info_size;
};
```

Field Descriptions

`module_name`

An index to an entry in the string table indicating the name of the module.

`iextdefsym`

The index into the symbol table of the first defined external symbol provided by this module.

`nextdefsym`

The number of defined external symbols provided by this module.

`irefsym`

The index into the external reference table of the first entry provided by this module.

`nrefsym`
The number of external reference entries provided by this module.

`ilocalsym`
The index into the symbol table of the first local symbol provided by this module.

`nlocalsym`
The number of local symbols provided by this module.

`iextrel`
The index into the external relocation table of the first entry provided by this module.

`nextrel`
The number of entries in the external relocation table that are provided by this module.

`iinit_iterm`
Contains both the index into the module initialization section (the low 16 bits) and the index into the module termination section (the high 16 bits) to the pointers for this module.

`ninit_nterm`
Contains both the number of pointers in the module initialization (the low 16 bits) and the number of pointers in the module termination section (the high 16 bits) for this module.

`objc_module_info_addr`
The statically linked address of the start of the data for this module in the `__module_info` section in the `__OBJC` segment.

`objc_module_info_size`
The number of bytes of data for this module that are used in the `__module_info` section in the `__OBJC` segment.

dylib_reference

The structure of an external reference table entry for the external reference entries provided by a module in a shared library.

```
struct dylib_reference
{
    unsigned  long isym:24,
                   flags:8;
};
```

Field Descriptions

`isym`
An index into the symbol table for the symbol being referenced.

`flags`
A constant for the type of reference being made. Use the same `REFERENCE_FLAG` constants as described in the `nlist` (page 69) structure description.

Relocation Data Structures

Relocation is the process of moving symbols to a different address. When the static linker moves a symbol (a function or an item of data) to a different address, it needs to change all the references to that symbol to use the new address. The **relocation entries** in a Mach-O file contain offsets in the file to addresses that need to be relocated when the contents of the file are relocated. The addresses are usually relative offsets stored in CPU instructions; the exact format of the address is specified in each relocation entry. When creating the intermediate object file, the compiler generates one relocation entry for every instruction that contains a relative address. The static linker typically removes the relocation entries when building the final product, as relocation local to a single Mach-O file does not usually occur at runtime.

relocation_info

Describes an item in the file that uses an address that needs to be updated when the address is changed. This data structure is declared in `/usr/include/mach-o/reloc.h`.

```
struct relocation_info
{
    long r_address;
    unsigned int r_symbolnum:24,
                 r_pcrel:1,
                 r_length:2,
                 r_extern:1,
                 r_type:4;
};
```

Field Descriptions

`r_address`

In `MH_OBJECT` files, this is an offset from the start of the section to the item containing the address requiring relocation. If the high bit of this field is set (which you can check using the `R_SCATTERED` bit mask), the `relocation_info` structure is actually a `scattered_relocation_info` (page 77) structure.

In images used by the dynamic linker, this is an offset from the virtual memory address of the data of the first `segment_command` (page 55) that appears in the file (not necessarily the one with the lowest address). For images with the `MH_SPLIT_SEGS` flag set, this is an offset from the virtual memory address of data of the first read/write `segment_command` (page 55).

`r_symbolnum`

Indicates either an index into the symbol table (when the `r_extern` field is set to 1) or a section number (when the `r_extern` field is set to zero). As previously mentioned, sections are ordered from 1 to 255 in the order in which they appear in the `LC_SEGMENT` load commands. This field is set to `R_ABS` for relocation entries for absolute symbols, which need no relocation.

`r_pcrel`

Indicates whether the item containing the address to be relocated is part of a CPU instruction that uses PC-relative addressing.

For addresses contained in PC-relative instructions, the CPU adds the address of the instruction to the address contained in the instruction.

`r_length`

Indicates the length of the item containing the address to be relocated. A value of zero indicates a single byte; a value of 1 indicates a 2-byte address, and a value of 2 indicates a 4-byte address.

`r_extern`

Indicates whether the `r_symbolnum` field is an index into the symbol table (1) or a section number (zero).

`r_type`

Indicates the type of relocation to be performed. Possible values for this field are shared between this structure and the `scattered_relocation_info` (page 77) data structure; see the description of the `r_type` field in the `scattered_relocation_info` (page 77) data structure for more details.

scattered_relocation_info

Describes an item in the file—using a non-zero constant in its relocatable expression or two addresses in its relocatable expression—that needs to be updated if the addresses that it uses are changed. This information is needed to reconstruct the addresses that make up the relocatable expression's value in order to change the addresses independently of each other. This data structure is declared in `/usr/include/mach-o/reloc.h`.

```
struct scattered_relocation_info
{
#ifdef __BIG_ENDIAN__
    unsigned int r_scattered:1,
                 r_pcrel:1,
                 r_length:2,
                 r_type:4,
                 r_address:24;
    long r_value;
#endif /* __BIG_ENDIAN__ */
#ifdef __LITTLE_ENDIAN__
    unsigned int r_address:24,
                 r_type:4,
                 r_length:2,
                 r_pcrel:1,
                 r_scattered:1;
    long r_value;
#endif /* __LITTLE_ENDIAN__ */
};
```

Field Descriptions

`r_scattered`

If this bit is zero, this structure is actually a `relocation_info` (page 76) structure.

`r_address`

In `MH_OBJECT` files, this is an offset from the start of the section to the item containing the address requiring relocation. If the high bit of this field is clear (which you can check using the `R_SCATTERED` bit mask), this structure is actually a `relocation_info` (page 76) structure.

In images used by the dynamic linker, this is an offset from the virtual memory address of the data of the first `segment_command` (page 55) that appears in the file (not necessarily the one with the lowest address). For images with the `MH_SPLIT_SEGS` flag set, this is an offset from the virtual memory address of data of the first read/write `segment_command` (page 55).

Since this field is only 24 bits long, the offset in this field can never be larger than 0x00FFFFFF, thus limiting the size of the relocatable contents of this image to 16 megabytes.

`r_pcrel`

See `relocation_info` (page 76).

`r_length`

See `relocation_info` (page 76).

r_type

For x86 processors, the r_type field may contain any of these values:

- GENERIC_RELOC_VANILLA—A generic relocation entry for both addresses contained in data and addresses contained in CPU instructions.
- GENERIC_RELOC_PAIR—The second relocation entry of a pair.
- GENERIC_RELOC_SECTDIFF—A relocation entry for an item that contains the difference of two section addresses. This is generally used for position-independent code generation. GENERIC_RELOC_SECTDIFF contains the address from which to subtract; it must be followed by a GENERIC_RELOC_PAIR containing the address to subtract.
- GENERIC_RELOC_PB_LA_PTR—A relocation entry for a prebound lazy pointer. This is always a scattered relocation entry. The r_value field contains the non-prebound value of the lazy pointer.

For PowerPC processors, the r_type field is usually PPC_RELOC_VANILLA for addresses contained in data. Relocation entries for addresses contained in CPU instructions are described by other r_type values:

- PPC_RELOC_PAIR—The second relocation entry of a pair. A PPC_RELOC_PAIR entry must follow each of the other relocation entry types, except for PPC_RELOC_VANILLA, PPC_RELOC_BR14, PPC_RELOC_BR24, and PPC_RELOC_PB_LA_PTR.
- PPC_RELOC_BR14—The instruction contains a 14-bit branch displacement.
- PPC_RELOC_BR24—The instruction contains a 24-bit branch displacement.
- PPC_RELOC_HI16—The instruction contains the high 16 bits of a relocatable expression. The next relocation entry must be a PPC_RELOC_PAIR specifying the low 16 bits of the expression in the low 16 bits of the r_value field.
- PPC_RELOC_LO16—The instruction contains the low 16 bits of an address. The next relocation entry must be a PPC_RELOC_PAIR specifying the high 16 bits of the expression in the low (*not* the high) 16 bits of the r_value field.
- PPC_RELOC_HA16—Same as the PPC_RELOC_HI16 except the low 16 bits and the high 16 bits are added together with the low 16 bits sign-extended first. This means if bit 15 of the low 16 bits is set, the high 16 bits stored in the instruction are adjusted.
- PPC_RELOC_LO14—Same as PPC_RELOC_LO16 except that the low 2 bits are not stored in the CPU instruction and are always zero. PPC_RELOC_LO14 is used in 64-bit load/store instructions.
- PPC_RELOC_SECTDIFF—A relocation entry for an item that contains the difference of two section addresses. This is generally used for position-independent code generation. PPC_RELOC_SECTDIFF contains the address from which to subtract; it must be followed by a PPC_RELOC_PAIR containing the section address to subtract.
- PPC_RELOC_PB_LA_PTR—A relocation entry for a prebound lazy pointer. This is always a scattered relocation entry. The r_value field contains the non-prebound value of the lazy pointer.
- PPC_RELOC_HI16_SECTDIFF—Section difference form of PPC_RELOC_HI16.
- PPC_RELOC_LO16_SECTDIFF—Section difference form of PPC_RELOC_LO16.
- PPC_RELOC_HA16_SECTDIFF—Section difference form of PPC_RELOC_HA16.

- PPC_RELOC_JBSR—A relocation entry for the assembler synthetic opcode jbsr, which is a 24-bit branch-and-link instruction using a branch island. The branch displacement is assembled to the branch island address and the relocation entry indicates the actual target symbol. If the linker is able to make the branch reach the actual target symbol, it does. Otherwise, the branch is relocated to the branch island.
- PPC_RELOC_LO14_SECTDIFF—Section difference form of PPC_RELOC_LO14.

r_value

The address of the relocatable expression for the item in the file that needs to be updated if the address is changed. For relocatable expressions with the difference of two section addresses, the address from which to subtract (in mathematical terms, the minuend) is contained in the first relocation entry and the address to subtract (the subtrahend) is contained in the second relocation entry.

Discussion

Mach-O relocation data structures support two types of relocatable expressions in machine code and data:

- **Symbol address + constant**. The most typical form of relocation, adding a simple constant value to the existing address.
- **Address of section y - address of section x + constant**. The section difference form of relocation. This form of relocation supports position-independent code.

Static Archive Libraries

This section describes the file format used for static archive libraries, which are described in "Static Archive Libraries" (page 14). Mac OS X uses a format derived from the original BSD static archive library format, with a few minor additions. See the discussion for the ranlib data structure for more information.

ranlib

The data structure of a static archive library symbol table entry. It is declared in /usr/include/mach-o/ranlib.h.

```
struct ranlib
{
    union
    {
        unsigned long ran_strx;
        char * ran_name;
    } ran_un;
    unsigned long ran_off;
};
```

Field Descriptions

ran_strx

The index number (zero-based) of the string in the string table that follows the array of ranlib data structures.

`ran_name`

The byte offset, from the start of the file, at which the symbol name can be found.

`ran_off`

The byte offset, from the start of the file, at which the header line for the member containing this symbol can be found.

Discussion

A static archive library begins with the file identifier string `!<arch>`, followed by a newline character (ASCII value 0x0A). The file identifier string is followed by a series of member files. Each member consists of a fixed-length header line followed by the file data. The header line is 60 bytes long and is divided into five fixed-length fields, as shown in this example header line:

```
grapple.c       999514211   501   20    100644  167       `
```

The last 2 bytes of the header line are a grave accent (`) character (ASCII value 0x60) and a newline character. All header fields are defined in ASCII and padded with spaces to the full length of the field. All fields are defined in decimal notation, except for the file mode field, which is defined in octal. These are the descriptions for each field:

- The name field (16 bytes) contains the name of the file. If the name is either longer than 16 bytes or contains a space character, the actual name should be written directly after the header line and the name field should contain the string `#1/` followed by the length. To keep the archive entries aligned to 4 byte boundaries, the length of the name that follows the `#1/` is rounded to 4 bytes and the name that follows the header is padded with null bytes.
- The modified date field (12 bytes) is taken from the `st_time` field returned by the `stat` system call.
- The user ID field (6 bytes) is taken from the `st_uid` field returned by the `stat` system call.
- The group ID field (6 bytes) is taken from the `st_gid` field returned by the `stat` system call.
- The file mode field (8 bytes) is taken from the `st_mode` field returned by the `stat` system call. This field is written in octal notation.
- The file size field (8 bytes) is taken from the `st_size` field returned by the `stat` system call.

The first member in a static archive library is always the symbol table describing the contents of the rest of the member files. This member is always called either `__.SYMDEF` or `__.SYMDEF SORTED` (note the two leading underscores and the period). The name used depends on the sort order of the symbol table. The older variant—`__.SYMDEF`—contains entries in the same order that they appear in the object files. The newer variant—`__.SYMDEF SORTED`— contains entries in alphabetical order, which allows the static linker to load the symbols faster.

The `__.SYMDEF` and `.__SORTED SYMDEF` archive members contain an array of `ranlib` data structures preceded by the length in bytes (a long integer, 4 bytes) of the number of items in the array. The array is followed by a string table of null-terminated strings, which are preceded by the length in bytes of the entire string table (again, a 4-byte long integer).

The string table is an array of C strings, each terminated by a null byte.

The `ranlib` declarations can be found in `/usr/include/mach-o/ranlib.h`.

Special Considerations

Prior to the advent of `libtool`, a tool called `ranlib` was used to generate the symbol table. `ranlib` has since been integrated into `libtool`. See the man page for `libtool` for more information.

Multi-CPU Architecture Files

The standard development tools normally accept multiple-CPU (or "fat") files as parameters wherever a normal Mach-O file or static archive library is accepted. The dynamic linker loads the correct data for the currently running CPU from fat shared libraries, frameworks, and bundles.

A fat file is not really a Mach-O file at all. It is a simple archive format that contains the data of either multiple Mach-O files (one for each CPU architecture you wish to support) or multiple static archive libraries (again, one for each CPU architecture you wish to support). You can have a fat file containing data for only one CPU architecture, although it's not very useful to do so.

A multiple-architecture, or "fat" file is one that contains compiled code and data for more than one CPU architecture. A fat file contains a set of single-CPU files, one for each CPU architecture, with a special header at the beginning of the file to allow the various runtime tools to quickly find a particular CPU architecture. Each single-CPU file is stored as a continuous set of bytes at an offset in the fat file. The single-CPU files may currently be either Mach-O files or static archive libraries. For example, a fat static archive library might contain the data of one static archive library containing PowerPC modules and also the data for one static archive library containing x86 modules.

A fat file always begins with a `fat_header` (page 82) data structure, followed by a set of `fat_arch` (page 83) data structures and the actual data for the CPU architectures contained in the file. All data in these data structures is stored in big-endian byte order.

fat_header

Describes the layout of a fat file, which is a file that can contain code and data for more than one CPU architecture. This data structure is declared in the header `/usr/include/mach-o/fat.h`.

```
struct fat_header
{
    unsigned long magic;
    unsigned long nfat_arch;
};
```

Field Descriptions

`magic`

An integer containing the value `0xCAFEBABE` in big-endian byte order format. On a big-endian host CPU, this can be validated using the constant `FAT_MAGIC`; on a little-endian host CPU, it can be validated using the constant `FAT_CIGAM`.

`nfat_arch`

An integer specifying the number of `fat_arch` (page 83) data structures that follow. This is the number of CPU architectures contained in this file.

Discussion

The `fat_header` data structure is placed at the start of a file that contains Mach-O files for multiple CPU architectures. Directly following the `fat_header` data structure is a set of `fat_arch` (page 83) data structures, one for each CPU architecture included in the file.

Regardless of the content it describes, all the fields in this data structure are stored in big-endian byte order.

fat_arch

Describes the location within the file of data for a single CPU architecture. This data structure is declared in `/usr/include/mach-o/fat.h`.

```
struct fat_arch
{
    cpu_type_t cputype;
    cpu_subtype_t cpusubtype;
    unsigned long offset;
    unsigned long size;
    unsigned long align;
};
```

Field Descriptions

`cputype`

An enumeration value of type `cpu_type_t`. Specifies the CPU family.

`cpusubtype`

An enumeration value of type `cpu_subtype_t`. Specifies the specific member of the CPU family on which this entry may be used or a constant specifying all members.

`offset`

Offset to the beginning of the data for this CPU.

`size`

Size of the data for this CPU.

`align`

The power of 2 alignment for the offset of the contents of this CPU architecture. This is required to ensure that, if this fat file is changed, the contents it retains are correctly aligned for virtual memory paging and other uses.

Discussion

An array of `fat_arch` data structures appears directly after the `fat_header` (page 82) data structure of a file that contains Mach-O files for multiple CPU architectures.

Regardless of the content it describes, all the fields in this data structure are stored in big-endian byte order.

Revision History

The table below describes the revisions to *Mach-O Runtime Architecture*.

Date	Notes
2004-08-31	Added information on parameter passing, section names, dynamic linking of libraries, dead-code stripping flags, and GPR11. Removed dynamic linking functions reference. Minor technical and editorial corrections throughout.
	Added information on `MH_SUBSECTIONS_VIA_SYMBOLS` flag to "mach_header" (page 51) struct.
	Added information on the `S_ATTR_STRIP_STATIC_SYMS`, `S_ATTR_LIVE_SUPPORT`, and `S_ATTR_NO_DEAD_STRIP` flags to "section" (page 56) struct.
	Added explanation of `PPC_RELOC_LO14_SECTDIFF` to `scattered_relocation_info` (page 77).
	Added clarification on when callers put parameters in the stack, in addition to placing them in registers in "Parameter Passing" (page 34).
	Added details on parameter passing for single-member structures to "Parameter Passing" (page 34).
	Added note to introduction in "Mach-O File Format Reference" (page 47) indicating that compilers can define additional section names that are not shown in Table 3-1 (page 49).
	Refined description of GPR11 in "Register Preservation" (page 39).
	Specified correct sizes for composite parameters that are preceded by padding to make them 4 bytes in size in "Parameter Passing" (page 34).
	Corrected sample of a private external symbol in "Scope and Treatment of Symbol Definitions" (page 19).
	Corrected ranges for `unsigned int`, `unsigned long`, and `unsigned long long` in Table 2-1 (page 27), and `vector unsigned int` in Table 2-2 (page 28).

Date	Notes
	Corrected framework-building example in Listing 1-1 (page 23).
	Removed "Mach-O Dynamic Linking Functions Reference" chapter and placed its content in *Mach-O Runtime Reference*.
2003-08-07	Added description of new API for Mac OS X version 10.3.
2003-01-01	Incorporated developer feedback. Updated code-generation examples.
	Fixed bugs 2462895, 2749339, 2909989, 2910422, 2921574.
2002-07-01	More developer feedback. Document weak definitions and weak references (new for 10.2). Substantially update the glossary. Other tweaks and additional material. Clarify common vs. coalesced symbol definitions.
	ABI: Rewrote position-independent and indirect code section, incorporating correct examples and separating PIC and indirect code generation. Add C99 `_Bool` data type.
	Fixed bugs 2909989, 2910422, and 2921574.
2002-05-01	This was a preliminary draft distributed with the WWDC 2002 developer tools.
	Incorporated many corrections from developer review. More to come.
	By popular demand, added some common usage scenarios to map runtime features to the options in the standard Mac OS X tools that implement those features. To satisfy a related popular demand, this information is collected in a separate chapter, which allows users of third-party tool sets to ignore it. This chapter is currently unfinished, and the overview chapter is yet to be modified to cross-reference it.
	Updated umbrella framework description to better match reality.
	Added `long double` and `long long` return value information. Removed last vestiges of CFM. Rewrote data alignment section, incorporating the correct rules (inherited from IBM's `xlc` compiler) for power alignment mode, and adding new natural alignment mode.
2002-04-01	This was a preliminary draft distributed with the April 2002 Developer Tools CD.

Glossary

bundle (1) A Mach-O file that must be explicitly loaded by the client application before use. Bundles typically contain code and data that extend the capabilities of an application. Also called a *plug-in*, *application extension* or *drop-in addition*. Compare dynamic shared library.

bundle (2) A file package containing loadable code and resources, in the format understood by the NSBundle and CFBundle classes.

client application In the context of a shared library or a bundle, the application that imports the shared library or that loads the bundle. Usually refers specifically to the main program file of the application.

coalesced symbol A symbol that may be defined in multiple intermediate object files. The symbols have the same name and occupy the same amount of space. The static linker ignores all but one copy of the symbol. Compare common symbol.

Code Fragment Manager (CFM) The part of Mac OS 9 that loads code from Preferred Executable Format (PEF) files into memory and prepares it for execution. Supported in Carbon for compatibility with Mac OS 9.

common symbol A symbol that may be defined in multiple intermediate object files. The symbols have the same name but may occupy different amounts of space. The static linker uses only one definiton—the largest— in the output file. Compare coalesced symbol. See also tentative definition.

debugging symbol A symbol generated by the compiler to enable the debugger to map machine code to source code.

defined external symbol A data item or executable routine within a fragment that is made available for use by other Mach-O files. Also called *imported symbol*. Compare undefined external symbol.

dependent library Another name for import library.

dynamic shared library An image that exports functions and global variables to other images. A shared library is not included with the application code at link time but is linked in dynamically at runtime. Also known as a *shared library*, *dynamic library*, *dylib*, *dynamic link library*, and *DLL*. Compare framework.

ELF An executable file format commonly used in UNIX operating systems.

embedding alignment The alignment of a data item within a composite data item (such as a data structure). Compare natural alignment.

entry point A location (offset) within a module.

epilog A sequence of code that cleans up the stack after a procedure call (restoring registers, restoring the stack pointer, and so on).

executable file As a generic term, refers to any file containing binary machine code, including bundles, shared libraries and programs. Often used to specifically refer to the main program file of an application. See also program.

exported symbol See defined external symbol.

external reference A reference to a routine or variable defined in a separate compilation unit or assembly.

fat application An application that contains code for two or more CPU architectures. For example, a fat application may contain both x86 and PowerPC code. Compare fat file.

fat file A file that contains an archive of one or more Mach-O files, each containing code and data tailored to a specific CPU architecture.

final product Any file output from the static linker that the static linker cannot perform further binding on; dynamic shared libraries, main program files, and

bundles are all final products, while intermediate object files and static archive libraries are not.

fragment In the Code Fragment Manager runtime architecture, an executable unit of code and its associated data. No defined meaning for Mach-O.

frame pointer (FP)A pointer to the beginning of a stack frame.

framework A shared library packaged with associated resources, such as header files, localized strings, and documentation files.

image Any Mach-O file built for use with the dynamic linker. In Mac OS X, this currently includes shared libraries, bundles, and executables (that is, Mach-O files of types `MH_DYLIB`, `MH_BUNDLE`, and `MH_EXECUTABLE`).

imported symbol See undefined external symbol

import library A dynamic shared library referenced by a Mach-O file. Also called dependent library.

initialization function A function contained in a shared library or bundle that is executed immediately after the file is loaded. Compare termination function.

install name the pathname to a dynamic shared library, as recorded at build time. Mach-O files refer to shared libraries using the install name.

leaf procedure A routine that calls no other routines.

linkage area The area in the PowerPC stack that holds the calling routine's RTOC value and the saved values of the Condition Register and the Link Register. Compare parameter area.

main symbol For applications, the main routine or main entry point. Shared libraries and bundles do not require a main symbol.

main program file The Mach-O file (of type `MH_EXECUTABLE`) that contains the main entry point of a program. Often called an *executable*, but *executable* can also be used to describe any file containing machine code that can be executed.

module The smallest indivisible unit of machine code and data that can be linked independently in a Mach-O file.

natural alignment The alignment of a data type when allocated in memory or assigned a memory address. Compare embedding alignment.

parameter area The area in the PowerPC stack that holds the parameters for any routines called by a given routine. Compare linkage area.

PEF See Preferred Executable Format (PEF).

plug-in See bundle (1).

PowerPC microprocessor Any member of the family of PowerPC microprocessors.

CFM runtime architecture The runtime architecture for PowerPC computers running classic Mac OS or using the Preferred Executable Format (PEF) with Carbon on Mac OS X.

Preferred Executable Format (PEF) The format of executable files used for Carbon applications and shared libraries that use the Code Fragment Manager.

private framework A framework that is a part of one or more applications, but not part of the system. Private frameworks are often installed in a `Frameworks` directory inside an application package. Private frameworks are often used to provide functionality that is shared between the main program and any bundles loaded by the main program.

process A running program. See also program.

program An executable unit that contains executable code.

prolog A sequence of code that prepares the stack for a procedure call (by saving registers, adjusting the stack, and so on).

red zone In the PowerPC CPU architecture, the area of memory immediately above the address pointed to by the stack pointer. The red zone is reserved for temporary use by a routine's prolog and as an area to store a leaf procedure's nonvolatile registers.

reference The location within one module that contains the address of another module or entry point.

relocation The process of replacing references to symbols with actual addresses during fragment preparation.

runtime architecture A set of basic rules that define how software operates. It dictates how code and data are addressed, the form of generated code, how applications are handled, and how to enable system calls. The runtime architecture defines the core of the runtime environment. Compare runtime environment.

runtime environment The execution environment provided by the Mac OS X kernel and dynamic linker. The runtime environment dictates how executable code is loaded into memory, where data is stored, and how routines call other functions. Compare runtime architecture.

section A named storage unit in a segment of a Mach-O file that contains either code or data.

segment A named collection of sections in a Mach-O file.

shared library See dynamic shared library.

stack An area of memory in the application partition that is used for temporary storage of data during the operation of that application or other software.

stack frame The area of the stack used by a routine for its parameters, return address, local variables, and temporary storage.

stack pointer (SP) A pointer to the top of the stack.

static archive library An archive of modules whose code is included in the application at link time. Also called a *static library*.

symbol A reference to a function or data item.

termination function A function contained in a shared library or bundle that is executed just before the file is unloaded. Compare initialization function.

tentative definition A symbol that has no initializer and is not marked with the ANSI C `extern` keyword. The standard compiler transforms tentative definitions into common symbols. Compare coalesced symbol.

transition vector In the CFM runtime architecture, an 8-byte data structure that describes the entry point and base register address of a routine.

undefined external symbol A data item or executable routine referenced by a fragment but not contained in it. An import is identified by name to the linker, but its actual address is bound at load time by the dynamic linker. Also called *imported symbol*. Compare defined external symbol.

weak library A shared library that does not need to be present at runtime for the client application to run. Sometimes called a *soft library*.

weak definition A defined external symbol that the static linker may ignore in the presence of a non-weak defined external symbol. Used to support some features of C++. Compare weak reference.

weak reference An undefined external symbol that does not need to be present at runtime for the client application to run. If the dynamic linker cannot find a definition of a weak reference, it sets the address of the symbol to zero. The client application can then test the weak symbol against `NULL` to see whether or not the symbol was found. Also known as a *weak import* or *soft import*. Compare weak definition.

x86 microprocessor Any microprocessor capable of directly executing machine code for the IA-32 instruction set.

Index

M

N

O

P

R

S

T

U

W