Mac OS X ABI Mach-O File Format Reference

Tools > Darwin

2007-04-26

Contents

Figures and Tables

Mac OS X ABI Mach-O File Format Reference

Declared in:
loader.h
machine.h
fat.h
nlist.h
ranlib.h
reloc.h

Overview

This document describes the structure of the Mach-O (Mach object) file format, which is the standard used to store programs and libraries on disk in the Mac OS X application binary interface (ABI). To understand how the Xcode tools work with Mach-O files, and to perform low-level debugging tasks, you need to understand this information.

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 Mac OS X evolved, resulting in a single file format for both statically linked and dynamically linked code.

Basic Structure

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

■ At the beginning of every Mach-O file is a **header structure** that identifies the file as a Mach-O file. The header also contains other basic file type information, indicates the target 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 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 dynamic linking)

❏ 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.

■ In user-level fully linked Mach-O files, the last segment is the **link edit** segment. This segment contains the tables of link edit information, such as the symbol table, string table, and so forth, used by the dynamic loader to link an executable file or Mach-O bundle to its dependent libraries.

Figure 1 Mach-O file format basic structure
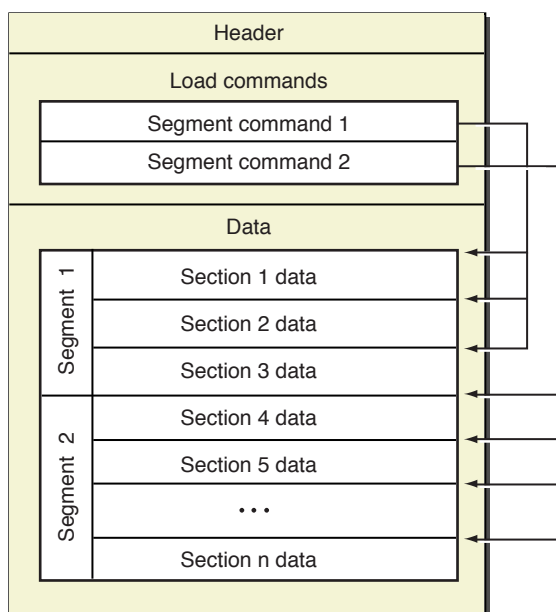
Various tables within a Mach-O file refer to sections by number. Section numbering begins at 1 (not 0) and continues across segment boundaries. Thus, the first segment in a file may contain sections 1 and 2 and the second segment may contain sections 3 and 4.

When using the Stabs debugging format, the symbol table also holds debugging information. When using DWARF, debugging information is stored in the image's corresponding dSYM file, specified by the `uuid_command` (page 20) structure

Header Structure and Load Commands

A Mach-O file contains code and data for one architecture. The header structure of a Mach-O file specifies the target architecture, which allows the kernel to ensure that, for example, code intended for PowerPC-based Macintosh computers is not executed on Intel-based Macintosh computers.

You can group multiple Mach-O files (one for each architecture you want to support) in one binary using the format described in "Universal Binaries and 32-bit/64-bit PowerPC Binaries" (page 55).

> **Note:** Binaries that contain object files for more than one architecture are not Mach-O files. They archive one or more Mach-O files.

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.

Segments

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. A segment contains zero or more sections.

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 0. Because `__PAGEZERO` contains no data, there is no need for it to occupy any space in the executable file.

> **Note:** Sections that are to be filled with zeros must always be placed at the end of the segment. Otherwise, the standard tools will not be able to successfully manipulate the Mach-O file.

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 object file. The data structure that defines a `section` (page 23) contains the name of the segment the section is intended for, and the static linker places each section in the final object file accordingly.

For best performance, segments should be aligned on virtual memory page boundaries—4096 bytes for PowerPC and 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.

These are the segments the standard Mac OS X development tools (contained in the Xcode Tools CD) may include in a 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 0 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 architecture (for Intel-based and PowerPC-based Macintosh computers, 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 0).

■ 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 __IMPORT segment contains symbol stubs and non-lazy pointers to symbols not defined in the executable. This segment is generated only for executables targeted for the IA-32 architecture.

■ The __LINKEDIT segment contains raw data used by the dynamic linker, such as symbol, string, and relocation table entries.

Sections

The __TEXT and __DATA segments may contain a number of standard sections, listed in Table 1, Table 2 (page 11), and Table 3 (page 11). 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 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 23) data type.

Table 1   The sections of a __TEXT segment

Segment and section name	Contents
__TEXT,__text	Executable machine code. The compiler generally places only executable code in this section, no tables or data of any sort.
__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 "Dynamic Code Generation" in *Mach-O Programming Topics* for more information.
__TEXT,__symbol_stub	Indirect symbol stubs. See "Dynamic Code Generation" in *Mach-O Programming Topics* for more information.

Segment and section name	Contents
__TEXT,__const	Initialized constant variables. The compiler places all nonrelocatable data declared const in this section. (The compiler typically places uninitialized constant variables in a zero-filled 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 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 architectures, it's more efficient for the compiler to use immediate load instructions rather than adding to this section.

Table 2   The sections of a __DATA segment

Segment and section name	Contents
__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 "Dynamic Code Generation" in *Mach-O Programming Topics* for more information.
__DATA,__nl_symbol_ptr	Non-lazy symbol pointers, which are indirect references to data items imported from a different file. See "Dynamic Code Generation" in *Mach-O Programming Topics* for more information.
__DATA,__dyld	Placeholder section used by the dynamic linker.
__DATA,__const	Initialized relocatable 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).

Table 3   The sections of a __IMPORT segment

Segment and section name	Contents
__IMPORT,__jump_table	Stubs for calls to functions in a dynamic library.

Segment and section name	Contents
`__IMPORT,__pointers`	Non-lazy symbol pointers, which are direct references to functions imported from a different file.

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

Data Types

This reference 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 56) and `fat_arch` (page 56), which are written in big-endian byte order.

Header Data Structure

mach_header

Specifies the general attributes of a file. Appears at the beginning of object files targeted to 32-bit architectures. Declared in `/usr/include/mach-o/loader.h`. See also `mach_header_64` (page 14).

```
struct mach_header
{
    uint32_t magic;
    cpu_type_t cputype;
    cpu_subtype_t cpusubtype;
    uint32_t filetype;
    uint32_t ncmds;
    uint32_t sizeofcmds;
    uint32_t flags;
};
```

Fields

magic

> An integer containing a value identifying this file as a 32-bit Mach-O 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 architecture you intend to use the file on. Appropriate values include:

- `CPU_TYPE_POWERPC` to target PowerPC-based Macintosh computers

- `CPU_TYPE_I386` to target the Intel-based Macintosh computers

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 of the `dyld` file.

- The `MH_DSYM` file type designates files that store symbol information for a corresponding binary file.

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_TWOLEVEL`—The image is using two-level namespace bindings.

- `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_PREBINDABLE`—This file is not prebound but can have its prebinding redone. Used only when `MH_PREBEOUND` is not set.

- `MH_NOFIXPREBINDING`—The dynamic linker doesn't notify the prebinding agent about this executable.

- `MH_ALLMODSBOUND`—Indicates that this binary binds to all two-level namespace modules of its dependent libraries. Used only when `MH_PREBINDABLE` and `MH_TWOLEVEL` are set.

- `MH_CANONICAL`—This file has been canonicalized by unprebinding—clearing prebinding information from the file. See the `redo_prebinding` man page for details.

- `MH_SPLIT_SEGS`—The file has its read-only and read-write segments split.

- `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 "Linking" in Xcode User Guide for details.

- `MH_NOMULTIDEFS`—This umbrella guarantees there are no multiple definitions of symbols in its subimages. As a result, the two-level namespace hints can always be used.

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 and 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`).

mach_header_64

Defines the general attributes of a file targeted for a 64-bit architecture. Declared in `/usr/include/mach-o/loader.h`.

```
struct mach_header_64
{
    uint32_t magic;
    cpu_type_t cputype;
    cpu_subtype_t cpusubtype;
    uint32_t filetype;
    uint32_t ncmds;
    uint32_t sizeofcmds;
    uint32_t flags;
    uint32_t reserved;
};
```

Fields

magic

> An integer containing a value identifying this file as a 64-bit Mach-O file. Use the constant
> `MH_MAGIC_64` 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_64` can be used when the byte
> ordering scheme of the target machine is the reverse of the host CPU.

cputype

> An integer indicating the architecture you intend to use the file on. The only appropriate value
> for this structure is:

> ■ `CPU_TYPE_x86-64` to target 64-bit Intel-based Macintosh computers.

> ■ `CPU_TYPE_POWERPC64` to target 64-bit PowerPC–based Macintosh computers.

cpusubtype

> An integer specifying the exact model of the CPU. To run on all PowerPC processors supported
> by the Mac OS X kernel, this should be set to `CPU_SUBTYPE_POWERPC_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 of the `dyld` file.

- The `MH_DSYM` file type designates files that store symbol information for a corresponding binary file.

`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_TWOLEVEL`—The image is using two-level namespace bindings.

- `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_PREBINDABLE`—This file is not prebound but can have its prebinding redone. Used only when `MH_PREBEOUND` is not set.

- `MH_NOFIXPREBINDING`—The dynamic linker doesn't notify the prebinding agent about this executable.

- `MH_ALLMODSBOUND`—Indicates that this binary binds to all two-level namespace modules of its dependent libraries. Used only when `MH_PREBINDABLE` and `MH_TWOLEVEL` are set.

- `MH_CANONICAL`—This file has been canonicalized by unprebinding—clearing prebinding information from the file. See the `redo_prebinding` man page for details.

- `MH_SPLIT_SEGS`—The file has its read-only and read-write segments split.

- `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 "Linking" in *Xcode User Guide* for details.

- `MH_NOMULTIDEFS`—This umbrella guarantees there are no multiple definitions of symbols in its subimages. As a result, the two-level namespace hints can always be used.

`reserved`

Reserved for future use.

Special Considerations

See comment in `mach_header` (page 12)

Load Command Data Structures

The load command structures are located directly after the header of the object 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
{
    uint32_t cmd;
    uint32_t cmdsize;
};
```

Fields

cmd

An integer indicating the type of load command. Table 4 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. In 32-bit architectures, the size must always be a multiple of 4; in 64-bit architectures, the size must always be a multiple of 8. If the load command data does not divide evenly by 4 or 8 (depending on whether the target architecture is 32-bit or 64-bit, respectively), add bytes containing zeros to the end until it does.

Discussion

Table 4 lists the valid load command types, with links to the full data structures for each type.

Table 4   Mach-O load commands

Commands	Data structures	Purpose
LC_UUID	uuid_command (page 20)	Specifies the 128-bit UUID for an image or its corresponding dSYM file.
LC_SEGMENT	segment_command (page 20)	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.
LC_SEGMENT_64	segment_command_64 (page 21)	Defines a 64-bit 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.
LC_SYMTAB	symtab_command (page 39)	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 45)	Specifies additional symbol table information used by the dynamic linker.
LC_THREAD LC_UNIXTHREAD	thread_command (page 34)	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.

Commands	Data structures	Purpose
`LC_LOAD_DYLIB`	`dylib_command` (page 32)	Defines the name of a dynamic shared library that this file links against.
`LC_ID_DYLIB`	`dylib_command` (page 32)	Specifies the install name of a dynamic shared library.
`LC_PREBOUND_DYLIB`	`prebound_dylib_command` (page 33)	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 33)	Specifies the dynamic linker that the kernel executes to load this file.
`LC_ID_DYLINKER`	`dylinker_command` (page 33)	Identifies this file as a dynamic linker.
`LC_ROUTINES`	`routines_command` (page 35)	Contains the address of the shared library initialization routine (specified by the linker's `-init` option).
`LC_ROUTINES_64`	`routines_command_64` (page 36)	Contains the address of the shared library 64-bit initialization routine (specified by the linker's `-init` option).
`LC_TWOLEVEL_HINTS`	`twolevel_hints_command` (page 30)	Contains the two-level namespace lookup hint table.
`LC_SUB_FRAMEWORK`	`sub_framework_command` (page 37)	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.
`LC_SUB_UMBRELLA`	`sub_umbrella_command` (page 37)	Specifies a file that is a subumbrella of this umbrella framework.
`LC_SUB_LIBRARY`	`sub_library_command` (page 38)	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 38)	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.

Declared In

`/usr/include/mach-o/loader.h`

uuid_command

Specifies the 128-bit universally unique identifier (UUID) for an image or for its corresponding dSYM file.

```
struct uuid_command
{
    uint32_t cmd;
    uint32_t cmdsize;
    uint8_t uuid[16];
};
```

Fields

cmd

Set to `LC_UUID` for this structure.

cmdsize

Set to `sizeof(uuid_command)`.

uuid

128-bit unique identifier.

Declared In

`/usr/include/mach-o/loader.h`

segment_command

Specifies the range of bytes in a 32-bit Mach-O file that make up a segment. Those bytes are mapped by the loader into the address space of a program. Declared in `/usr/include/mach-o/loader.h`. See also

```
struct segment_command
{
    uint32_t cmd;
    uint32_t cmdsize;
    char segname[16];
    uint32_t vmaddr;
    uint32_t vmsize;
    uint32_t fileoff;
    uint32_t filesize;
    vm_prot_t maxprot;
    vm_prot_t initprot;
    uint32_t nsects;
    uint32_t flags;
};
```

Fields

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 0. 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.

Special Considerations

Segments with sections of type `S_GB_ZEROFILL` are placed after all other segments. See `section` (page 23) for additional information.

segment_command_64

Specifies the range of bytes in a 64-bit Mach-O file that make up a segment. Those bytes are mapped by the loader into the address space of a program. If the 64-bit segment has sections, they are defined by `section_64` (page 27) structures. Declared in `/usr/include/mach-o/loader.h`.

```
struct segment_command_64
{
    uint32_t cmd;
    uint32_t cmdsize;
    char segname[16];
    uint64_t vmaddr;
    uint64_t vmsize;
    uint64_t fileoff;
    uint64_t filesize;
    vm_prot_t maxprot;
    vm_prot_t initprot;
    uint32_t nsects;
    uint32_t flags;
};
```

Fields

cmd

> See description in segment_command (page 20). Set to LC_SEGMENT_64 for this structure.

cmdsize

> Common to all load command structures. For this structure, set this field to
> sizeof(segment_command_64) plus the size of all the section data structures that follow
> (sizeof(segment_command_64 + (sizeof(section_64) * 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 0. 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

Defines the elements used by a 32-bit 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 20) structure. Declared in `/usr/include/mach-o/loader.h`. See also `section_64` (page 27).

```
struct section
{
    char sectname[16];
    char segname[16];
    uint32_t addr;
    uint32_t size;
    uint32_t offset;
    uint32_t align;
    uint32_t reloff;
    uint32_t nreloc;
    uint32_t flags;
    uint32_t reserved1;
    uint32_t reserved2;
};
```

Fields

`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 "Dynamic Code Generation" in *Mach-O Programming Topics* 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.

■ `S_GB_ZEROFILL`—This is a zero-filled on-demand section. It can be larger than 4 GB. This section must be placed in a segment containing only zero-filled sections. If you place a zero-filled section in a segment with non–zero-filled sections, you may cause those sections to be unreachable with a 31-bit offset. That outcome stems from the fact that the size of a zero-filled section can be larger than 4 GB (in a 32-bit address space). As a result of this, the static linker would be unable to build the output file. See `segment_command` (page 20) for more information.

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_SOME_INSTRUCTIONS`—This section contains executable machine instructions.

■ `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_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 12) header structure is set.

- `S_ATTR_NO_DEAD_STRIP`—This section must not be dead-stripped. See "Linking" in *Xcode User Guide* 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 0.

`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 0.

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. Some section types and attributes are used by the dynamic linker.

These are important static-linking variants of the symbol type and attributes:

- **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. To support complex language features (such as C++ vtables and RTTI) the compiler may create a definition of a particular symbol in every intermediate object file. The static linker and the dynamic linker would then reduce the duplicate definitions to the single definition used by the program.

- **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 39)). 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 if they are expected to be used on earlier versions of Mac OS X.

section_64

Defines the elements used by a 64-bit section. Directly following a `segment_command_64` (page 21) data structure is an array of `section_64` data structures, with the exact count determined by the `nsects` field of the segment_command_64 structure. Declared in `/usr/include/mach-o/loader.h`.

```
struct section_64
{
    char sectname[16];
    char segname[16];
    uint64_t addr;
    uint64_t size;
    uint32_t offset;
    uint32_t align;
    uint32_t reloff;
    uint32_t nreloc;
    uint32_t flags;
    uint32_t reserved1;
    uint32_t reserved2;
};
```

Fields

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 "Dynamic Code Generation" in *Mach-O Programming Topics* 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.

■ `S_GB_ZEROFILL`—This is a zero-filled on-demand section. It can be larger than 4 GB. This section must be placed in a segment containing only zero-filled sections. If you place a zero-filled section in a segment with non–zero-filled sections, you may cause those sections to be unreachable with a 31-bit offset. That outcome stems from the fact that the size of a zero-filled section can be larger than 4 GB (in a 32-bit address space). As a result of this, the static linker would be unable to build the output file. See `segment_command_64` (page 21) for more information.

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_SOME_INSTRUCTIONS`—This section contains executable machine instructions.

■ `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_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 12) header structure is set.

- `S_ATTR_NO_DEAD_STRIP`—This section must not be dead-stripped. See "Linking" in *Xcode User Guide* 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 0.

`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 0.

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. Some section types and attributes are used by the dynamic linker.

These are important static-linking variants of the symbol type and attributes:

- **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. To support complex language features (such as C++ vtables and RTTI) the compiler may create a definition of a particular symbol in every intermediate object file. The static linker and the dynamic linker would then reduce the duplicate definitions to the single definition used by the program.

- **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 39)). 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 if they are expected to be used on earlier versions of Mac OS X.

twolevel_hints_command

Defines the attributes of a `LC_TWOLEVEL_HINTS` load command. Declared in `/usr/include/mach-o/loader.h`.

```
struct twolevel_hints_command
{
    uint32_t cmd;
    uint32_t cmdsize;
    uint32_t offset;
    uint32_t nhints;
};
```

Fields

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 30) 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. Declared in `/usr/include/mach-o/loader.h`.

```
struct twolevel_hint
{
    uint32_t isub_image:8,
        itoc:24;
};
```

Fields

isub_image

The subimage in which the symbol is defined. It is an index into the list of images that make up the umbrella image. If this field is 0, the symbol is in the umbrella image itself. If the image is not an umbrella framework or library, this field is 0.

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 has 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. Declared in `/usr/include/mach-o/loader.h`.

```
union lc_str
{
    uint32_t offset;
    #ifndef __LP64__
    char *ptr;
    #endif
};
```

Fields

`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. The `ptr` field is not used in Mach-O files.

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. The string should be null terminated; any extra bytes to round up the size should be null. You can also 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 32) data structure. Declared in `/usr/include/mach-o/loader.h`.

```
struct dylib
{
    union lc_str  name;
    uint_32 timestamp;
    uint_32 current_version;
    uint_32 compatibility_version;
};
```

Fields

name

A data structure of type lc_str (page 31). 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.

dylib_command

Defines the attributes of the LC_LOAD_DYLIB and LC_ID_DYLIB load commands. Declared in /usr/include/mach-o/loader.h.

```
struct dylib_command
{
    uint_32 cmd;
    uint_32 cmdsize;
    struct dylib dylib;
};
```

Fields

cmd

Common to all load command structures. For this structure, set to either LC_LOAD_DYLIB, LC_LOAD_WEAK_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 31). Specifies the attributes of the shared library.

Discussion

For each shared library that a file links against, the static linker creates an LC_LOAD_DYLIB command and sets its dylib field to the value of the dylib field of the LC_ID_DYLD load command of the target library. 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 39) data structure.

At runtime, the dynamic linker uses the name in the `dyld` field of the `LC_LOAD_DYLIB` command to locate the shared library. If it finds the library, the dynamic linker compares the version information of the `LC_LOAD_DYLIB` load command against the library's version. For the dynamic linker to successfully link the shared library, the compatibility version of the shared library must be less than or equal to the compatibility version in the `LC_LOAD_DYLIB` command.

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.

dylinker_command

Defines the attributes of the `LC_LOAD_DYLINKER` and `LC_ID_DYLINKER` load commands. Declared in `/usr/include/mach-o/loader.h`.

```
struct dylinker_command
{
    uint32_t cmd;
    uint32_t cmdsize;
    union lc_str name;
};
```

Fields

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 31). 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

Defines the attributes of 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. Declared in `/usr/include/mach-o/loader.h`.

```
struct prebound_dylib_command
{
    uint32_t cmd;
    uint32_t cmdsize;
    union lc_str name;
    uint32_t nmodules;
    union lc_str linked_modules;
};
```

Fields

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 31). 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 31). 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, 0 for no. The bit for the first module is the low bit of the first byte.

thread_command

Defines the attributes of the `LC_THREAD` and `LC_UNIXTHREAD` load commands. The data of this command is specific to each architecture and appears in `thread_status.h`, located in the architecture's directory in `/usr/include/mach`. Declared in `/usr/include/mach-o/loader.h`.

```
struct thread_command
{
    uint32_t cmd;
    uint32_t cmdsize;
    /* uint32_t flavor;*/
    /* uint32_t count; */
    /* struct cpu_thread_state state;*/
};
```

Fields

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 target 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

Defines the attributes of the `LC_ROUTINES` load command, used in 32-bit architectures. 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. Declared in
`/usr/include/mach-o/loader.h`. See also `routines_command_64` (page 36).

```
struct routines_command
{
    uint32_t cmd;
    uint32_t cmdsize;
    uint32_t init_address;
    uint32_t init_module;
    uint32_t reserved1;
    uint32_t reserved2;
    uint32_t reserved3;
    uint32_t reserved4;
    uint32_t reserved5;
    uint32_t reserved6;
};
```

Fields

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 `0`.

reserved2

>   Reserved for future use. Set this field to `0`.

reserved3

>   Reserved for future use. Set this field to `0`.

reserved4

>   Reserved for future use. Set this field to `0`.

reserved5

>   Reserved for future use. Set this field to `0`.

35

reserved6

>Reserved for future use. Set this field to `0`.

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).

routines_command_64

Defines the attributes of the `LC_ROUTINES_64` load command, used in 64-bit architectures. 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. Declared in `/usr/include/mach-o/loader.h`.

```
struct routines_command_64
{
    uint32_t cmd;
    uint32_t cmdsize;
    uint64_t init_address;
    uint64_t init_module;
    uint64_t reserved1;
    uint64_t reserved2;
    uint64_t reserved3;
    uint64_t reserved4;
    uint64_t reserved5;
    uint64_t reserved6;
};
```

Fields

cmd

>Common to all load command structures. For this structure, set to `LC_ROUTINES_64`.

cmdsize

>Common to all load command structures. For this structure, set to `sizeof(routines_command_64)`.

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 `0`.

reserved2

>Reserved for future use. Set this field to `0`.

reserved3

>Reserved for future use. Set this field to `0`.

reserved4

>Reserved for future use. Set this field to `0`.

reserved5

>Reserved for future use. Set this field to `0`.

`reserved6`

> Reserved for future use. Set this field to `0`.

Discussion

The static linker adds an `LC_ROUTINES_64` command when you specify a shared library initialization function using the `-init` option (see the `ld` man page for more information).

sub_framework_command

Defines the attributes of the `LC_SUB_FRAMEWORK` load command. Identifies the umbrella framework of which this file is a subframework. Declared in `/usr/include/mach-o/loader.h`.

```
struct sub_framework_command
{
    uint32_t cmd;
    uint32_t cmdsize;
    union lc_str umbrella;
};
```

Fields

`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 31). Specifies the name of the umbrella framework of which this file is a member.

sub_umbrella_command

Defines the attributes of 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. Declared in `/usr/include/mach-o/loader.h`.

```
struct sub_umbrella_command
{
    uint32_t cmd;
    uint32_t cmdsize;
    union lc_str sub_umbrella;
};
```

Fields

`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 31). Specifies the name of the umbrella framework of which this file is a member.

sub_library_command

Defines the attributes of 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. Declared in `/usr/include/mach-o/loader.h`.

```
struct sub_library_command
{
    uint32_t cmd;
    uint32_t cmdsize;
    union lc_str sub_library;
};
```

Fields

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 31). Specifies the name of the sublibrary of which this file is a member.

sub_client_command

Defines the attributes of 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. Declared in `/usr/include/mach-o/loader.h`.

```
struct sub_client_command
{
    uint32_t cmd;
    uint32_t cmdsize;
    union lc_str client;
};
```

Fields

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 31). Specifies the name of a client authorized to link to this library.

Special Considerations

The `ld` tool generates a `sub_client_command` load command in the built product if you pass the option `-allowable_client <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` 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

Defines the attributes of the `LC_SYMTAB` load command. Describes the size and location of the symbol table data structures. Declared in `/usr/include/mach-o/loader.h`.

```
struct symtab_command
{
    uint_32 cmd;
    uint_32 cmdsize;
    uint_32 symoff;
    uint_32 nsyms;
    uint_32 stroff;
    uint_32 strsize;
};
```

Fields

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 39) 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 for 32-bit architectures. Declared in `/usr/include/mach-o/nlist.h`. See also `nlist_64` (page 42).

```
struct nlist
{
    union {
    #ifndef __LP64__
        char *n_name;
    #endif
        int32_t n_strx;
    } n_un;
    uint8_t n_type;
    uint8_t n_sect;
    int16_t n_desc;
    uint32_t n_value;
};
```

Fields

n_un

A union that holds an index into the string table, n_strx. To specify an empty string (""), set this value to 0. 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. The n_sect field is set to NO_SECT.

- N_ABS (0x2)—The symbol is absolute. The linker does not change the value of an absolute symbol. The n_sect field is set 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. The n_sect field is set 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 have 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 for non-stab symbols. 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 defined symbol that is referenced by dynamic-loader APIs (such as `dlsym` and `NSLookupSymbolInImage`) and not ordinary undefined symbol references. 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)—Sometimes used by the dynamic linker at runtime in a fully linked image. Do not set this bit in a fully linked image.

- `N_NO_DEAD_STRIP`(0x20)—When set in a relocatable object file (file type `MH_OBJECT`) on a defined symbol, indicates to the static linker to never dead-strip the symbol. (Note that the same bit (0x20) is used for two nonoverlapping purposes.)

- `N_WEAK_REF` (0x40)—Indicates that this undefined symbol is a weak reference. If the dynamic linker cannot find a definition for this symbol, it sets the address of this symbol to 0. 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 23) 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 undefined 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 253 specify the library number according to the order of `LC_LOAD_DYLIB` commands in the file. The value 254 is used for undefined symbols that are to be dynamically looked up (supported only in Mac OS X v10.3 and later). 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 0.

`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.

nlist_64

Describes an entry in the symbol table for 64-bit architectures. Declared in `/usr/include/mach-o/nlist.h`.

```
struct nlist_64
{
    union {
        uint32_t n_strx;
    } n_un;
    uint8_t n_type;
    uint8_t n_sect;
    uint16_t n_desc;
    uint64_t n_value;
};
```

Fields

`n_un`

>   A union that holds an index into the string table, `n_strx`. To specify an empty string (`""`), set this value to 0.

`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 defined symbol that is referenced by dynamic-loader APIs (such as `dlsym` and `NSLookupSymbolInImage`) and not ordinary undefined symbol references. 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 0. 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 23) 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 0.

`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

See discussion in `nlist` (page 39).

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. Declared in `/usr/include/mach-o/loader.h`.

```
struct dysymtab_command
{
    uint32_t cmd;
    uint32_t cmdsize;
    uint32_t ilocalsym;
    uint32_t nlocalsym;
    uint32_t iextdefsym;
    uint32_t nextdefsym;
    uint32_t iundefsym;
    uint32_t nundefsym;
    uint32_t tocoff;
    uint32_t ntoc;
    uint32_t modtaboff;
    uint32_t nmodtab;
    uint32_t extrefsymoff;
    uint32_t nextrefsyms;
    uint32_t indirectsymoff;
    uint32_t nindirectsyms;
    uint32_t extreloff;
    uint32_t nextrel;
    uint32_t locreloff;
    uint32_t nlocrel;
};
```

Fields

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 0. These tables are described in "Dynamic Code Generation" in *Mach-O Programming Topics*.

dylib_table_of_contents

Describes an entry in the table of contents of a dynamic shared library. Declared in `/usr/include/mach-o/loader.h`.

```
struct dylib_table_of_contents
{
    uint32_t symbol_index;
    uint32_t module_index;
};
```

Fields

`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 for 32-bit architectures. Declared in `/usr/include/mach-o/loader.h`. See also `dylib_module_64` (page 48).

```
struct dylib_module
{
    uint32_t module_name;
    uint32_t iextdefsym;
    uint32_t nextdefsym;
    uint32_t irefsym;
    uint32_t nrefsym;
    uint32_t ilocalsym;
    uint32_t nlocalsym;
    uint32_t iextrel;
    uint32_t nextrel;
    uint32_t iinit_iterm;
    uint32_t ninit_nterm;
    uint32_t objc_module_info_addr;
    uint32_t objc_module_info_size;
};
```

Fields

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_module_64

Describes a module table entry for a dynamic shared library for 64-bit architectures. Declared in `/usr/include/mach-o/loader.h`.

```
struct dylib_module_64
{
    uint32_t module_name;
    uint32_t iextdefsym;
    uint32_t nextdefsym;
    uint32_t irefsym;
    uint32_t nrefsym;
    uint32_t ilocalsym;
    uint32_t nlocalsym;
    uint32_t iextrel;
    uint32_t nextrel;
    uint32_t iinit_iterm;
    uint32_t ninit_nterm;
    uint32_t objc_module_info_size;
    uint64_t objc_module_info_addr;
};
```

Fields

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

Defines the attributes of an external reference table entry for the external reference entries provided by a module in a shared library. Declared in `/usr/include/mach-o/loader.h`.

```
struct dylib_reference
{
    uint32_t isym:24,
                 flags:8;
};
```

Fields

`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 39) 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 stored in CPU instructions can be absolute or relative. Each relocation entry specifies the exact format of the address. When creating the intermediate object file, the compiler generates one or more relocation entries for every instruction that contains an address. Because relocation to symbols at fixed addresses, and to relative addresses for position independent references, does not occur at runtime, the static linker typically removes some or all the relocation entries when building the final product.

Note: In the Mac OS X x86-64 environment scattered relocations are not used. Compiler-generated code uses mostly external relocations, in which the `r_extern` bit is set to 1 and the `r_symbolnum` field contains the symbol-table index of the target label.

relocation_info

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

```
struct relocation_info
{
    int32_t r_address;
    uint32_t r_symbolnum:24,
                r_pcrel:1,
                r_length:2,
                r_extern:1,
                r_type:4;
};
```

Fields

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 52) 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 20) 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 20).

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 0). 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. The following table lists r_length values and the corresponding address length.

Value	Address length
0	1 byte
1	2 bytes
2	4 bytes
3	4 bytes. See description for the `PPC_RELOC_BR14` r_type in `scattered_relocation_info` (page 52).

r_extern

Indicates whether the `r_symbolnum` field is an index into the symbol table (1) or a section number (0).

`r_type`

For the x86 environment, 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_LOCAL_SECTDIFF`—Similar to `GENERIC_RELOC_SECTDIFF` except that this entry refers specifically to the address in this item. If the address is that of a globally visible coalesced symbol, this relocation entry does not change if the symbol is overridden. This is used to associate stack unwinding information with the object code this relocation entry describes.

■ `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 the x86-64 environment, the `r_type` field may contain any of these values:

■ `X86_64_RELOC_BRANCH`—A CALL/JMP instruction with 32-bit displacement.

■ `X86_64_RELOC_GOT_LOAD`—A MOVQ load of a GOT entry.

■ `X86_64_RELOC_GOT`—Other GOT references.

■ `X86_64_RELOC_SIGNED`—Signed 32-bit displacement.

■ `X86_64_RELOC_UNSIGNED`—Absolute address.

■ `X86_64_RELOC_SUBTRACTOR`—Must be followed by a `X86_64_RELOC_UNSIGNED` relocation.

For PowerPC environments, 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. If the `r_length` is 3, the branch was statically predicted by setting or clearing the Y bit depending on the sign of the displacement or the opcode.

■ `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 0. `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_LOCAL_SECTDIFF`—Similar to PPC_RELOC_SECTDIFF except that this entry refers specifically to the address in this item. If the address is that of a globally visible coalesced symbol, this relocation entry does not change if the symbol is overridden. This is used to associate stack unwinding information with the object code this relocation entry describes

- `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`.

scattered_relocation_info

Describes an item in the file—using a nonzero 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. Declared in `/usr/include/mach-o/reloc.h`.

```
struct scattered_relocation_info
{
#ifdef __BIG_ENDIAN__
    uint32_t r_scattered:1,
                r_pcrel:1,
                r_length:2,
                r_type:4,
                r_address:24;
    int32_t r_value;
#endif /* __BIG_ENDIAN__ */
#ifdef __LITTLE_ENDIAN__
    uint32_t r_address:24,
                r_type:4,
                r_length:2,
                r_pcrel:1,
                r_scattered:1;
    int32_t r_value;
#endif /* __LITTLE_ENDIAN__ */
};
```

Fields

r_scattered

> If this bit is 0, this structure is actually a `relocation_info` (page 49) 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 49) 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 20) 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 20).

> 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

> 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 0 indicates a single byte; a value of 1 indicates a 2-byte address, and a value of 2 indicates a 4-byte address.

r_type

> Indicates the type of relocation to be performed. Possible values for this field are shared between this structure and the `relocation_info` data structure; see the description of the `r_type` field in the `relocation_info` (page 49) data structure for more details.

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 is referencing a symbol's address with no constant added. In this case, the value of the constant expression is 0.

■ **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. 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

Defines the attributes of a static archive library symbol table entry. Declared in `/usr/include/mach-o/ranlib.h`.

```
struct ranlib
{
    union
    {
        uint32_t ran_strx;
    #ifndef __LP64__
        char *ran_name;
    #endif
    } ran_un;
    uint32_t ran_off;
};
```

Fields

`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. This field is not used in Mach-O files.

`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 8 byte boundaries, the length of the name that follows the #1/ is rounded to 8 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.

Universal Binaries and 32-bit/64-bit PowerPC Binaries

The standard development tools accept as parameters two kinds of binaries:

■ Object files targeted at one architecture. These include Mach-O files, static libraries, and dynamic libraries.

■ Binaries targeted at more than one architecture. These binaries contain compiled code and data for one of these system types:

❏ PowerPC-based (32-bit and 64-bit) Macintosh computers. Binaries that contain code for both 32-bit and 64-bit PowerPC-based Macintosh computers are are known as **PPC/PPC64 binaries**.

❏ Intel-based and PowerPC-based (32-bit, 64-bit, or both) Macintosh computers. Binaries that contain code for both Intel-based and PowerPC-based Macintosh computers are known as **universal binaries**.

Each object file is stored as a continuous set of bytes at an offset from the beginning of the binary. They use a simple archive format to store the two object files with a special header at the beginning of the file to allow the various runtime tools to quickly find the code appropriate for the current architecture.

A binary that contains code for more than one architecture always begins with a `fat_header` (page 56) data structure, followed by two `fat_arch` (page 56) data structures and the actual data for the architectures contained in the file. All data in these data structures is stored in big-endian byte order.

fat_header

Defines the layout of a binary that contains code for more than one architecture. Declared in the header `/usr/include/mach-o/fat.h`.

```
struct fat_header
{
    uint32_t magic;
    uint32_t nfat_arch;
};
```

Fields

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 56) data structures that follow. This is the number of architectures contained in this binary.

Discussion

The `fat_header` data structure is placed at the start of a binary that contains code for multiple architectures. Directly following the `fat_header` data structure is a set of `fat_arch` (page 56) data structures, one for each architecture included in the binary.

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

fat_arch

Describes the location within the binary of an object file targeted at a single architecture. Declared in `/usr/include/mach-o/fat.h`.

```
struct fat_arch
{
    cpu_type_t cputype;
    cpu_subtype_t cpusubtype;
    uint32_t offset;
    uint32_t size;
    uint32_t align;
};
```

Fields

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 object file for the architecture specified in `cputype` within the binary. This is required to ensure that, if this binary 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 56) data structure of a binary that contains object files for multiple architectures.

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

Document Revision History

This table describes the changes to *Mac OS X ABI Mach-O File Format Reference*.

Date	Notes
2007-04-26	Added details about Mach-O files targeted for the Mac OS X x86-64 environment.
	Added relocation details to "Relocation Data Structures" (page 49).
	Updated the `cputype` field of the `mach_header_64` (page 14) structure.
	Updated the `r_type` bit of the `relocation_info` (page 49) structure.
	Added DWARF debugging-format information to the introduction.
2006-10-03	Added information about the uuid_command load command.
	Added `uuid_command` (page 20) and updated `load_command` (page 17).
2006-09-05	Added information about IA-32-specific structures and the file type for dSYM files.
2006-03-08	Corrected the mach_header_64 description.
	Replaced cross-references to "Indirect Addressing" throughout to cross-references to "Dynamic Code Generation" in *Mach-O Programming Topics*.
	Removed `CPU_SUBTYPE_I386_ALL` from the description for the `cpusubtype` field of `mach_header_64` (page 14).
2005-11-09	Changed title from "Mach-O File Format Reference."
	Added the phrase "Mac OS X application binary interface (ABI) to the introduction to raise this document's visibility in searches.
2005-08-11	Clarified terminology for binaries that contain object files for more than one architecture.

Date	Notes
2005-04-29	Added information on 64-bit support in the Mach-O file format. Removed the "Overview of the Runtime Architecture" and "Runtime Conventions for PowerPC" chapters. That content was placed in "Mac OS X Runtime Overview" and "PowerPC Runtime Programming Guide," respectively.
	Changed title to *Mach-O File Format Reference*.
	Updated symbol declarations to match headers.
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 12) struct.
	Added information on the `S_ATTR_STRIP_STATIC_SYMS`, `S_ATTR_LIVE_SUPPORT`, and `S_ATTR_NO_DEAD_STRIP` flags to "section" (page 23) struct.
	Added explanation of `PPC_RELOC_LO14_SECTDIFF` to `scattered_relocation_info` (page 52).
	Added clarification on when callers put parameters in the stack, in addition to placing them in registers. See .
	Added details on parameter passing for single-member structures. See "PowerPC Runtime Architecture Guide".
	Refined description of GPR11. See "PowerPC Runtime Architecture Guide".
	Specified correct sizes for composite parameters that are preceded by padding to make them 4 bytes in size. See "PowerPC Runtime Architecture Guide".
	Added note to introduction in "Mach-O File Format Reference" (page 12) indicating that compilers can define additional section names that are not shown in Table 1 (page 10).
	Corrected example of a private external symbol. See "Mach-O Programming Topics".
	Corrected ranges for `unsigned int`, `unsigned long`, and `unsigned long long`, and `vector unsigned int`. See "PowerPC Runtime Architecture Guide".
	Corrected framework-building example. See "Mach-O Programming Topics".
	Removed "Mach-O Dynamic Linking Functions Reference" chapter and placed its content in *Mach-O Runtime Reference*.

Date	Notes
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. See "PowerPC Runtime Architecture Guide".
	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.

Index

U