OS X ABI Mach-O File Format Reference

Inherits From	Conforms To	Import Statement
Not Applicable	Not Applicable	Not applicable

Availability

Not Applicable

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 app 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.

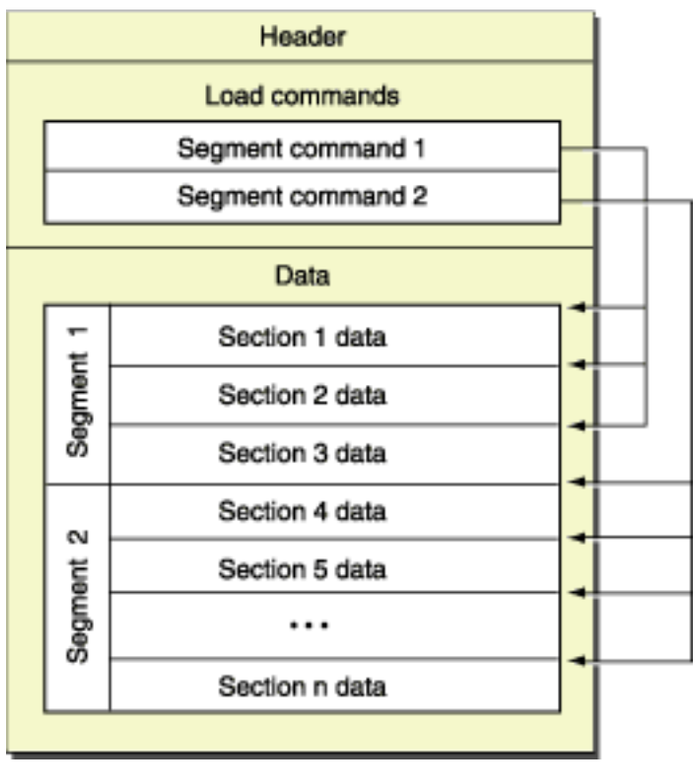
The Mach-O file format provides both intermediate (during the build process) and final (after linking the final product) storage of machine code and data. It was designed as a flexible replacement for the BSD `a.out` format, to be used by the compiler and the static linker and to contain statically linked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

> 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` 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 OS X development tools (contained in the Xcode Tools CD) may include in an 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 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, and Table 3. 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` 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 "Position-Independent Code" in *Mach-O Programming Topics* for more information.
`__TEXT,__symbol_stub`	Indirect symbol stubs. See "Position-Independent Code" in *Mach-O Programming Topics* for more information.
`__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 “Position-Independent Code” 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 “Position-Independent Code” 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.
`__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

Header Data Structure

`mach_header`

`mach_header_64`

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`
- `uuid_command`
- `segment_command`
- `segment_command_64`
- `section`
- `section_64`
- `twolevel_hints_command`
- `twolevel_hint`
- `lc_str`
- `dylib`
- `dylib_command`
- `dylinker_command`
- `prebound_dylib_command`
- `thread_command`
- `routines_command`
- `routines_command_64`
- `sub_framework_command`
- `sub_umbrella_command`
- `sub_library_command`
- `sub_client_command`

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`
- `nlist`
- `nlist_64`
- `dysymtab_command`
- `dylib_table_of_contents`
- `dylib_module`
- `dylib_module_64`
- `dylib_reference`

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 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`

`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`

Static Archive Libraries

This section describes the file format used for static archive libraries. 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`

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` data structure, followed by two `fat_arch` 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`

`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`.

Declaration

```
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` 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.