Mach-O Internals

William Woodruff

February 10, 2016

General Agenda

1. Who are you?
2. What is Mach-O?
3. An Extremely Brief History of Mach and Mach-O
4. Structure of a Mach-O File
5. Quirks Encountered
6. Concluding Notes

Who are you?

My name is William Woodruff.

I'm a Computer Science major and Philosophy minor at the University of Maryland, College Park.

Outside of school, I'm a member of the Homebrew project and a regular contributor to several open source groups. My work for Homebrew is largely concerned with the underlying system interface and reconciling OS X's intricacies/irregularities with the package manager.

What is Mach-O?

Mach-O is the Mach Object binary format.

Mach-O is used primarily by Apple in OS X and iOS.

“Apps” on both platforms are really just directory trees containing Mach-O binaries and resources (fonts, icons, configurations). Metadata is stored in a number of places, but mainly within bundled `plists` (XML) and the binaries themselves.

Like its cousins on Linux (ELF) and Windows (PE), Mach-O supports multiple object types:

- Executable
- Core dump
- Shared library/object
- Prelinked object file
- etc. . .

…and multiple architectures:

- m68k/m88k (yes, it's still supported!*)
- x86
- AMD64
- POWER
- ARMv6/7/8

Unlike ELF or PE, Mach-O has been extended to allow multi-architecture "fat" binaries. This has resulted in some interesting properties not shared by the other two. More on that later.

* `libmacho` will parse Mach-O files from several old architectures, including m68k, m88k, and PA-RISC. Don't expect to be able to execute them, however…

An Extremely Brief History of Mach and Mach-O

To understand why Mach-O was chosen as the binary format for OS X, it's first necessary to know and understand the parties involved:

- The Mach Project at Carnegie Mellon (1985 - 1994)
- NeXT Computer and the NeXTSTEP system (1987 - 1996)
- Finally, Apple and the Rhapsody project (1997 - Present)

The Mach Project (1985 - 1994)

The Mach project began in 1985 at Carnegie Mellon as an experiment in microkernel design.

Mach-O was created to ease representation of Mach's new microkernel primitives in compiled binaries.

CMU Mach development continued until 1994 (Mach 3), but was ultimately considered a failure due to severe performance penalties during IPC. GNU Mach picked up the goals of the CMU Mach project with the intention of becoming the kernel for the GNU Hurd project, but is *still* in development after over 20 years.

So how did Mach-O end up in OS X?

NeXT Computer and NeXTSTEP (1987 - 1996)

In 1987, NeXTSTEP was developed by NeXT for their workstations (which were designed to compete with both Apple and traditional UNIX workstations).

Besides being a UNIX itself, NeXTSTEP's kernel (XNU) was an amalgam of Mach 2.5 and 4.3BSD. This achieved a compromise between the performance of the monolithic BSD kernel and the IPC/message-passing abilities of Mach.

As a result of XNU's lineage, Mach-O became the binary format for NeXTSTEP. As NeXT expanded NeXTSTEP's hardware support beyond m68k, Mach-O was augmented to allow multiple binaries to exist within the same file.

NeXT was acquired by Apple in 1997.

Apple and the Rhapsody project (1997 - Present)

For several years, internal teams at Apple had been working on completely replacing the aging “System” OS family (then on System 9). After NeXT was acquired, NeXTSTEP became the basis for the Rhapsody project (which became the primary replacement team).

The NeXTStep userland became known as Darwin, while the XNU kernel was updated with components from Mach 3, FreeBSD, and NetBSD. The Mach-O format was tweaked accordingly, and support for PowerPC was brought in.

Rhapsody eventually became OS X, and the rest is history.

Structure of a thin Mach-O file

A single-architecture Mach-O can be broken into 3 main components:

- Header
  - Magic
  - CPU type and subtype
  - Filetype (executable, dump, etc)
  - Execution flags (e.g.: disable heap execution)
  - Number of load commands and size of load commands
- Load commands
  - Metadata (UUID, API level)
  - Shared object linkage locations (similar to `LD_PRELOAD`)
  - Offsets to segments
- Segments
  - Sections (up to 255 per segment)
    - Symbol lookup tables
    - TEXT, BSS, DATA

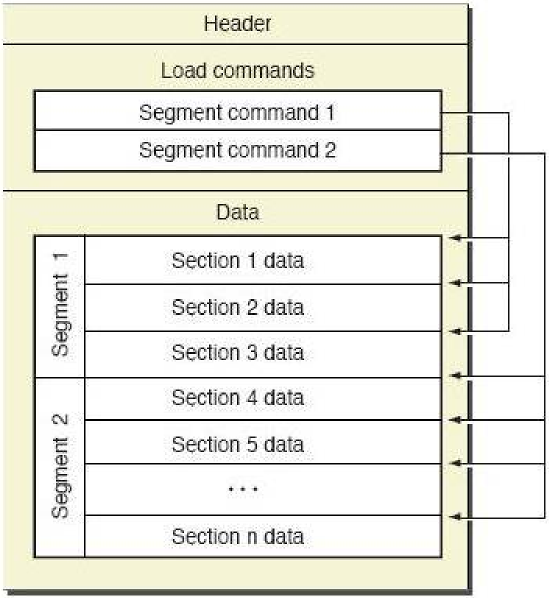


Figure : Mach-O layout (Source: OS X ABI Mach-O Reference)

Mach-O Header Structure

There are four possible (single-architecture) magic numbers:

- `MH_MAGIC = 0xfeedface` - 32-bit, big-endian
- `MH_CIGAM = 0xcefaedfe` - 32-bit, little-endian
- `MH_MAGIC_64 = 0xfeedfacf` - 64-bit, big-endian
- `MH_CIGAM_64 = 0xcffaedfe` - 64-bit, little-endian

The header's CPU type field specifies general architecture compatibility:

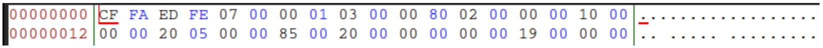
- `CPU_TYPE_POWERPC` - 32-bit PPC
- `CPU_TYPE_POWERPC64` - 64-bit PPC
- `CPU_TYPE_I386` - 80386 and above
- `CPU_TYPE_X86_64` - AMD64

The CPU subtype field provides more specific capability information.

mach_header_64

```
struct mach_header_64 {
  uint32_t magic;  /* mach magic number identifier */
  cpu_type_t cputype;  /* cpu specifier */
  cpu_subtype_t cpusubtype;  /* machine specifier */
  uint32_t filetype;  /* type of file */
  uint32_t ncmds;  /* number of load commands */
  uint32_t sizeofcmds;  /* size of all the load commands */
  uint32_t flags;  /* flags */
  uint32_t reserved;  /* reserved */
};
```

(Source: `mach-o/loader.h`)



```
00000000 CF FA ED FE 07 00 00 01 03 00 00 80 02 00 00 00 10 00 ..................
00000012 00 00 20 05 00 00 85 00 20 00 00 00 00 00 19 00 00 00 .. ..... .........
```

Figure : 32-bit Mach header

Relative offsets:

- `0 - 3` - file magic (`0xCFFAEDFE = MH_CIGAM`)
- `4 - 7` - CPU type (`0x10000070 = CPU_TYPE_X86_64`)
- `8 - 11` - CPU subtype (`0x80000003 = CPU_SUBTYPE_X86_ALL | CPU_SUBTYPE_LIB64`)
- `12 - 15` - filetype (`0x02 = MH_EXECUTE`)
- `16 - 19` - number of load commands (`0x10`)
- `20 - 23` - size of load commands (`0x520` bytes)
- `24 - 27` - execution flags (`0x85002000 = MH_PIE | MH_TWOLEVEL | MH_DYLDLINK | MH_NOUNDEFS`)

Load Commands

Load commands are variable-width binary blobs. They provide executable metadata, linker instructions, and references to the usual instructions/pages/memory regions that need to be loaded before execution can begin

They can either be self-contained or reference external structures, like segments and strings.

Each load command has an identifying number, like `LC_SEGMENT` (`0x01`, segment information) or `LC_SYMTAB` (`0x02`, symbol table information). There may be more than one of each load command type, and load commands of the same type are not (usually) contiguous within the binary.

Top-level Load Command Structure

```
struct load_command {
  uint32_t cmd;  /* type of load command */
  uint32_t cmdsize;  /* total size of command in bytes */
};
```

(Source: `mach-o/loader.h`)

Fields:

- `cmd` - The load command ID (e.g., `LC_ENCRYPTION_INFO`).
- `cmdsize` - The size, in bytes, of this load command.

No load commands use this literal structure - they all cast from it as a form of polymorphism.

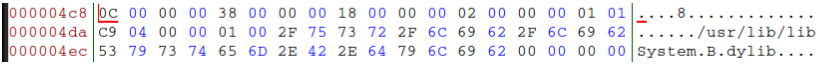
Specific Interesting Load Commands

- `LC_LOAD_DYLIB` and `LC_ID_DYLIB`
  Both of these commands use the **`dylib_command`** structure. The former specifies dynamic libraries to be linked from the Mach-O, while the latter specifies the "install name" of a dynamic library. These fields are commonly rewritten by programs like Homebrew to load dynamic libraries from their new installation location.
- `LC_SEGMENT` and `LC_SYMTAB`
  These commands use the **`segment_command[64]`** and **`symtab_command`** structures respectively. The former specifies a file region to be memory mapped into the process's address space, while the latter specifies the offset and dimensions of a BSD-style symbol table (recall OS X's heritage).

```c
struct dylib_command {
  uint32_t cmd;  /* LC_ID_DYLIB, LC_LOAD_DYLIB, etc */
  uint32_t cmdsize;  /* includes pathname string */
  struct dylib dylib;  /* the library identification */
};

struct dylib {
  union lc_str name;  /* library's path name */
  uint32_t timestamp;
  uint32_t current_version;
  uint32_t compatibility_version;
};

union lc_str {
  uint32_t offset;  /* offset to the string */
#ifndef __LP64__
  char *ptr;  /* pointer to the string */
#endif
};
```



```
000004c8 0C 00 00 00 38 00 00 00 18 00 00 00 02 00 00 00 01 01 ....8.............
000004da C9 04 00 00 01 00 2F 75 73 72 2F 6C 69 62 2F 6C 69 62 ....../usr/lib/lib
000004ec 53 79 73 74 65 6D 2E 42 2E 64 79 6C 69 62 00 00 00 00 System.B.dylib....
```

Figure : `LC_LOAD_DYLIB` command

Relative offsets:

- `0 - 3` - command name (`0x0c = LC_LOAD_DYLIB`)
- `4 - 7` - command size (`0x38` bytes)
- `8 - 11` - `lc_str` offset
  - `/usr/lib/libSystem.B.dylib`
- `12 - 15` - library build timestamp (usually not correct)
- `16 - 19` - library version number
- `20 - 23` - library compatibility number

Segments and Sections

Each segment command specifies its name (e.g.: `__TEXT`) and its section count.

Each section specifies its name (e.g: `__text`), its type, and its offset among other fields and flags.

Sections are the "meat" of the Mach-O file:

- `__text` - Executable instructions (read-only)
- `__data` - Initialized static data (read + write)
- `__bss` - Uninitialized static data (read + write)

Text sections can be made read + write with the `S_ATTR_SELF_MODIFYING_CODE` flag.

Executability for data and BSS sections is determined by the presence of a NX (No eXecute) bit in the Mach-O header. This can be disabled for the stack with `MH_ALLOW_STACK_EXECUTION` and enabled for the heap with `MH_NO_HEAP_EXECUTION`.

```
struct segment_command_64 { /* for 64-bit architectures */
  uint32_t cmd;  /* LC_SEGMENT_64 */
  uint32_t cmdsize;  /* includes sizeof section_64s */
  char segname[16];  /* segment name */
  uint64_t vmaddr;  /* memory address of this segment */
  uint64_t vmsize;  /* memory size of this segment */
  uint64_t fileoff;  /* file offset of this segment */
  uint64_t filesize;  /* amount to map from the file */
  vm_prot_t maxprot;  /* maximum VM protection */
  vm_prot_t initprot;  /* initial VM protection */
  uint32_t nsects;  /* number of sections in segment */
  uint32_t flags;  /* flags */
};
```

The `maxprot`, `initprot`, and `flags` fields (as well as header flags) are all important in determining the permissions afforded to a given segment and its sections.

Where do fat binaries come in?

Fat binaries are an *encapsulation* of the Mach-O format. They don't change the internal layout of each individual architecture's binary.

A fat Mach-O contains N entire single-architecture files (including their headers) with a little bit of extra metadata on the top.

Fat Mach-Os are identified by two additional magic numbers:

- `FAT_MAGIC = 0xcafebabe` - big endian
- `FAT_CIGAM = 0xbebafeca` - little endian

This is why some file managers mistake OS X binaries for Java classfiles. . .

Structure of a fat Mach-O file

- Header
  - Magic
  - Number of `fat_arch` structures
- `fat_arch` structures
  - CPU type, CPU subtype…
  - Offset to Mach-O corresponding to this architecture
  - Size of internal Mach-O blob and alignment
- Single-architecture Mach-Os
  - Header
  - Load commands
  - Segments
    - Sections

FIG. 2

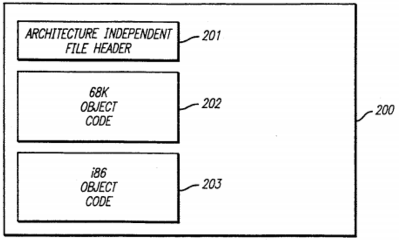


FIG. 3

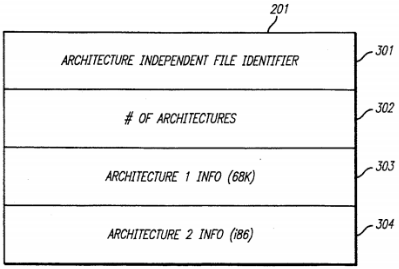

201
ARCHITECTURE INDEPENDENT FILE IDENTIFIER
301
# OF ARCHITECTURES
302
ARCHITECTURE 1 INFO (68K)
303
ARCHITECTURE 2 INFO (i86)
304

Figure : Source: US Pat. 5,432,937 (NeXT Computer)

So, how does OS X load the average Mach-O file?

- Open the file
- Check the magic
  - If single-arch and compatible with the machine, load normally
  - If single-arch but incompatible, fail
  - If multi-arch, parse the `fat_arch` list for a compatible Mach-O
    - If found, load normally beginning at that Mach-O's offset
    - If not found, fail

This is a lot more complex than single-architecture-only loading would be, but that's the cost of convenience.

FIG. 5

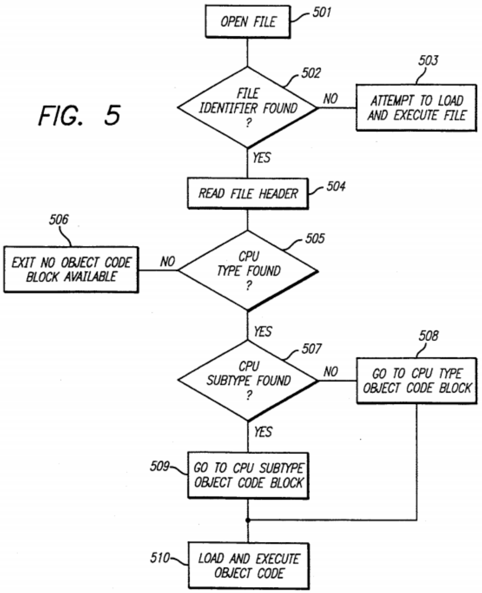

OPEN FILE
501
FILE IDENTIFIER FOUND ?
502
NO
ATTEMPT TO LOAD AND EXECUTE FILE
503
YES
READ FILE HEADER
504
CPU TYPE FOUND ?
505
NO
EXIT NO OBJECT CODE BLOCK AVAILABLE
506
YES
CPU SUBTYPE FOUND ?
507
NO
GO TO CPU TYPE OBJECT CODE BLOCK
508
YES
GO TO CPU SUBTYPE OBJECT CODE BLOCK
509
LOAD AND EXECUTE OBJECT CODE
510

Figure : Source: US Pat. 5,432,937 (NeXT Computer)

Quirks!

Hidden Masks!

A large number of system binaries on OS X have a mask (`0x80`) in their CPU subtype.

This isn't documented anywhere in `libmacho` or official Apple sources...

...but it *is* found as `CPU_SUBTYPE_LIB64` in the `clang` sources. No documentation besides that constant name is given.

My best guess: It was added to signify changes to a 64-bit ABI, and was stirred in when OS X made the switch to `clang` from `gcc` in userland.

Endian nightmares!

Mach-O does not have a uniform endianness like PE (Windows). This makes sense from a historical perspective (both NeXTSTEP and OS X originally ran on big-endian platforms), but both eventually settled on little-endian.

The result: Parsing fat Mach-Os with both big and little-endian data is messy (and not very well documented). Even when the file itself is little-endian, headers and flags may intentionally be big-endian for historical reasons.

This follows in the long Apple tradition of not updating the behavior of their core systems across architectures - just look at HFS(+).

Concluding Notes

Mach-O is fairly unique among object formats.

Despite being used solely on UNIX, Mach-O has no direct relationship to historical `a.out` or COFF formats. In fact, Windows' PE is more closely related to COFF than Mach-O is.

PE and ELF both have metadata in their headers, but neither has the concept of variable-length load commands. ELF comes close with its header table.

Neither has support for multi-architecture binaries:

- ELF has been extended by the FatELF project (now inactive/abandoned), which uses a layout directly inspired by the Mach-O fat format.
- Microsoft doesn't seem to have any interest in multi-architecture PE binaries. Providing separate binaries for each architecture remains standard practice.

References

- OS X ABI Mach-O File Format Reference
- US Pat. 5,432,937: Method and Apparatus for Architecture Independent Executable Files
- Homebrew/ruby-macho - GitHub
- libmacho - Apple Open Source
- FatELF: Universal Binaries for Linux

Miscellanea

Want to read the slides? They're here as a PDF and as Pandoc-style Markdown:

- `http://woodruffw.us/publications#macho-internals`