14

Mach Object Files

This chapter describes the format of Mach object files. This format is used by default, rather than the UNIX 4.3BSD **a.out** format, for object files on NEXTSTEP computers.

The current Mach object format is still evolving at Carnegie Mellon, and enhancements in NEXTSTEP are part of this evolving process. These enhancements refine the design and clean up some implementation details. The concepts of the original design are still present, but names have been changed for consistency.

The Mach object file format has two components:

- A static header containing information common to all files
- A variable number of load commands that provide information about the structure of the file

The load commands provide the following types of information:

- The layout of the run-time memory image
- The symbol table information
- The initial thread execution state
- The names of any referenced shared libraries

The layout of the file is determined by the file type:

- For types MH_EXECUTE and MH_FVMLIB the segments are padded out and aligned on a segment alignment boundary for efficient demand paging. Both these file types also have the headers included as part of their first segment.
- The type MH_OBJECT is a compact format (the ª.oº format). It's intended only as output of the assembler and input (or possibly output) of the link editor. All sections are in one unnamed segment with no padding.
- The type MH_PRELOAD is an executable format intended for files that aren't executed under the kernel (such as PROMs, standalone programs, and kernels).
- The type MH_CORE is for core files.

The structures of a Mach object file are defined in the header file **mach-o/loader.h**, and are described below. The structures and what they're used for are described first, followed by a list of what structures make up Mach object files.

The Mach Header

The Mach header appears at the beginning of the object file. Only information that's truly general to the file is contained in the Mach header. Other information is put in the load commands that

follow.

The format of the Mach header is:

```
struct mach_header {
    unsigned long  magic;      /* Mach magic number identifier */
    cpu_type_t     cputype;    /* cpu specifier */
    cpu_subtype_t  cpusubtype; /* machine specifier */
    unsigned long  filetype;   /* type of file */
    unsigned long  ncmds;      /* number of load commands */
    unsigned long  sizeofcmds; /* size of all load commands */
    unsigned long  flags;      /* flags */
};
```

The value for the **magic** field of the **mach_header** structure is:

```
#define MH_MAGIC    0xfeedface  /* the Mach magic number */
```

The values for the **cputype** and **cpusubtype** fields are defined as follows in the header file **sys/machine.h**:

```
#define CPU_TYPE_MC680x0     ((cpu_type_t) 6)
#define CPU_SUBTYPE_MC68030  ((cpu_subtype_t) 1)
#define CPU_SUBTYPE_MC68040  ((cpu_subtype_t) 2)
```

The values for the **filetype** field are defined as follows in the header file **sys/loader.h**:

```
#define MH_OBJECT   0x1    /* relocatable object file */
#define MH_EXECUTE  0x2    /* executable object file */
#define MH_FVMLIB   0x3    /* fixed vm shared library file */
#define MH_CORE     0x4    /* core file */
#define MH_PRELOAD  0x5    /* preloaded executable file */
```

The **ncmds** field contains the number of **load_command** structures that follow the Mach header. The **load_command** structures directly follow the Mach header in the object file.

The **sizeofcmds** field contains the total size in bytes of all of the load commands that follow it.

The following constants are used for the **flags** field:

```
#define MH_NOUNDEFS  0x1  /* object file has no undefined references;
                             can be executed */
#define MH_INCRLINK  0x2  /* object file is the output of an
                             incremental link against a base file;
                             can't be link-edited again */
```

The Load Commands

The load commands appear directly after the Mach header. They are variable in size. The number of load commands and the total size of the commands are given in the **ncmds** and **sizeofcmds** fields of the **mach_header** structure.

All load commands must have as their first two fields **cmd** and **cmdsize**:

- The **cmd** field contains a constant for that command type. Each command type has a specific structure corresponding to it.
- The **cmdsize** field is the size in bytes of the particular **load_command** structure plus anything that follows it that's a part of the load command (for example, **section** structures or strings). To advance to the next load command, the value of the **cmdsize** field can be added to the offset or pointer of the current load command.

The value of the **cmdsize** field must be a multiple of **sizeof(long)**. This is the maximum alignment

of any load command. The padded bytes must be zero-filled. Because the file will be memory mapped, all tables in the object file must also follow these rules; otherwise the pointers to these tables are not guaranteed to work. With all padding zero-filled, like objects will compare byte for byte.

The following structure is the minimum form of a load command:

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

Constants for the **cmd** field of the **load_command** structure are:

```
#define LC_SEGMENT     0x1   /* file segment to be mapped */
#define LC_SYMTAB      0x2   /* link-edit stab symbol table info
                                (obsolete) */
#define LC_SYMSEG      0x3   /* link-edit gdb symbol table info */
#define LC_THREAD      0x4   /* thread */
#define LC_UNIXTHREAD  0x5   /* UNIX thread (includes a stack) */
#define LC_LOADFVMLIB  0x6   /* load a fixed VM shared library */
#define LC_IDFVMLIB    0x7   /* fixed VM shared library id */
#define LC_IDENT       0x8   /* object identification information
                                (obsolete) */
#define LC_FVMFILE          0x9   /* fixed VM file inclusion */
```

A variable-length string in a load command is represented by an **lc_str** union. The string is stored just after the **load_command** structure, and the offset is from the start of the **load_command** structure. The size of the string is reflected in the **cmdsize** field of the load command. Any padded bytes to bring the **cmdsize** field to a multiple of **sizeof(long)** must be zero-filled.

```
union lc_str {
    unsigned long  offset;   /* offset to the string */
    char           *ptr;     /* pointer to the string */
};
```

The LC_SEGMENT Load Command

The LC_SEGMENT load command indicates that a part of this file is to be mapped into the task's address space. The size of this segment in memory, **vmsize**, can be equal to or larger than the amount to map from this file, **filesize**. The file, starting at **fileoff**, is mapped to the beginning of the segment in memory at **vmaddr**. The rest of the memory of the segment, if any, is allocated zero-fill on demand.

```
struct segment_command {
    unsigned long  cmd;           /* LC_SEGMENT */
    unsigned long  cmdsize;       /* includes size of section
                                     structures */
    char           segname[16];   /* segment's name */
    unsigned long  vmaddr;        /* segment's memory address */
    unsigned long  vmsize;        /* segment's memory size */
    unsigned long  fileoff;       /* segment's file offset */
    unsigned long  filesize;      /* amount to map from file */
    vm_prot_t      maxprot;       /* maximum VM protection */
    vm_prot_t      initprot;      /* initial VM protection */
    unsigned long  nsects;        /* number of sections */
    unsigned long  flags;         /* flags */
};
```

The segment's maximum virtual memory protection and initial virtual memory protection are specified by the **maxprot** and **initprot** fields. The values for these fields are set to some combination of the constants defined in the header file **vm/vm_prot.h**:

```
#define VM_PROT_NONE    ((vm_prot_t) 0x00)
#define VM_PROT_READ    ((vm_prot_t) 0x01)  /* read permission */
#define VM_PROT_WRITE   ((vm_prot_t) 0x02)  /* write permission */
#define VM_PROT_EXECUTE ((vm_prot_t) 0x04)  /* execute permission */

/* The default protection for newly created virtual memory */
#define VM_PROT_DEFAULT  \
        (VM_PROT_READ | VM_PROT_WRITE | VM_PROT_EXECUTE)

/* Maximum privileges possible, for parameter checking. */
#define VM_PROT_ALL  \
        (VM_PROT_READ | VM_PROT_WRITE | VM_PROT_EXECUTE)
```

A segment's address and virtual memory protection are set at link edit time.

The following constants can be used for the **flags** field of the **segment_command** structure:

```
#define SG_HIGHVM   0x1
#define SG_FVMLIB   0x2
#define SG_NORELOC  0x3
```

SG_HIGHVM indicates that the file contents for this segment occupy the high part of the virtual memory space; the low part is zero-filled (for stacks in core files). SG_FVMLIB indicates that the segment is the virtual memory that's allocated by a fixed virtual memory library for overlap checking in the link editor. SG_NORELOC indicates that the segment has nothing that was relocated in it and nothing relocated to it (that is, it may be safely replaced without relocation).

A segment is made up of zero or more sections. If the segment contains sections, the section structures directly follow the segment command and their size is reflected in the **cmdsize** field.

If sections have the same section name and are going into the same segment, they're combined by the link editor. The resulting section is aligned to the maximum alignment of the combined sections and is the new section's alignment. The combined sections are aligned to their original alignment in the combined section. Any padded bytes used to get the specified alignment are zero-filled.

Only non-MH_OBJECT files have all their segments with the proper sections in each padded to the specified segment alignment. The default segment alignment for the link editor is the page size. The first segment of an executable or shared library always contains the Mach header and load commands of the object file before its first section. The zero-filled sections are always last in their segment, allowing the zeroed segment padding to be mapped into memory where zero-filled sections might be.

```
struct section {
    char  sectname[16];         /* section's name */
    char  segname[16];          /* segment the section is in */
    unsigned long  addr;        /* section's memory address */
    unsigned long  size;        /* section's size in bytes */
    unsigned long  offset;      /* section's file offset */
    unsigned long  align;       /* section's alignment */
    unsigned long  reloff;      /* file offset of relocation entries */
    unsigned long  nreloc;      /* number of relocation entries */
    unsigned long  flags;       /* flags */
    unsigned long  reserved1;   /* reserved */
    unsigned long  reserved2;   /* reserved */
};
```

Flags currently defined for the **flags** field of a **section** structure are the following:

```
#define S_ZEROFILL          0x1  /* zero-filled on demand */
#define S_CSTRING_LITERALS 0x2  /* section has only literal C
                                   strings */
#define S_4BYTE_LITERALS    0x2  /* section has only 4-byte literals */
#define S_8BYTE_LITERALS    0x2  /* section has only 8-byte literals */
#define S_LITERAL_POINTERS 0x2  /* section has only pointers to
                                   literals */
```

S_ZEROFILL is used for the uninitialized data sections; sections with literal flags cause the link editor to coalesce redundant literals into sections and perform the proper relocation, resulting in a smaller file.

The format of the relocation entries referenced by the **reloff** and **nreloc** fields is described in the header file **reloc.h**.

Although the names of segments and sections in them are mostly meaningless to the link editor, there are a few things to support traditional UNIX executables that will require the link editor and assembler to use some agreed-upon names.

The link editor will allocate common symbols at the end of the **__common** section in the **__DATA** segment, creating the section and segment if needed. The **__common** section must be a zero-fill section (marked with S_ZEROFILL).

The default **maxprot** and **initprot** (maximum and initial virtual memory protection) will always be read, write, and execute. If there's a **__TEXT** or **__LINKEDIT** segment its **initprot** won't be writable by default.

The following are constants for the conventional segment and section names:

```
#define SEG_PAGEZERO      "__PAGEZERO"  /* pagezero segment; has no
                                           protections; catches NULL
                                           references for MH_EXECUTE
                                           files */
#define SEG_TEXT          "__TEXT" /* traditional UNIX text segment */
#define SECT_TEXT         "__text" /* real text part of the text
                                      section; no headers and
                                      padding */
#define SECT_FVMLIB_INIT0 "__fvmlib_init0"  /* fvmlib initialization
                                               section */
#define SECT_FVMLIB_INIT1 "__fvmlib_init1"  /* the section following
                                               the fvmlib
                                               initialization
                                               section */
#define SEG_DATA          "__DATA" /* traditional UNIX data segment */
#define SECT_DATA         "__data" /* real initialized data section;
                                      no padding, no bss overlap */
#define SECT_BSS          "__bss"  /* real uninitialized data
                                      section; no padding */
#define SECT_COMMON       "__common" /* the section common symbols
                                        are allocated in by the link
                                        editor */
#define SEG_OBJC          "__OBJC"          /* run-time segment */
#define SECT_OBJC_SYMBOLS "__symbol_table"  /* symbol table */
#define SECT_OBJC_MODULES "__module_info"   /* (obsolete!) */
#define SECT_OBJC_STRINGS "__selector_strs" /* string table */
#define SECT_OBJC_REFS    "__selector_refs" /* string table */
#define SEG_ICON          "__ICON"          /* NeXT icon segment */
#define SECT_ICON_HEADER  "__header"        /* icon headers */
#define SECT_ICON_TIFF    "__tiff"    /* icons in TIFF format */
```

The LC_SYMTAB Load Command

The LC_SYMTAB command specifies the location and size of the symbol table information created by the compiler used for link editing and debugging. This UNIX 4.3BSD **stab**-style symbol table information is defined in the header files **nlist.h** and **stabs.h**:

```
struct symtab_command {
    unsigned long  cmd;      /* LC_SYMTAB */
    unsigned long  cmdsize;  /* sizeof(struct symtab_command) */
    unsigned long  symoff;   /* symbol table offset */
```

```
    unsigned long  nsyms;    /* number of symbol table entries */
    unsigned long  stroff;   /* string table offset */
    unsigned long  strsize;  /* string table size in bytes */
};
```

The LC_SYMTAB command contains the offsets for both the symbol table entries and the string table used by those entries. This format is different from the UNIX 4.3BSD **a.out** format: The string table offset and size are explicitly defined, and the symbol table and string tables are located at the end of the file (not after the LC_SYMTAB command).

The format of a symbol table entry is defined in the header file **nlist.h**:

```
struct nlist {
    union {
        char      *n_name;   /* for use when in-core */
        long       n_strx;   /* index into file string table */
    } n_un;
    unsigned char  n_type;   /* type flag; see below */
    unsigned char  n_sect;   /* section number or NO_SECT */
    short          n_desc;   /* see the header file stab.h */
    unsigned       n_value;  /* value of this symbol table entry
                                (or stab offset) */
};
```

Symbols with an index into the string table of zero (**n_un.n_strx** == 0) are defined to have a null ("") name. Therefore, all string indexes to non-null names must not have a zero string length.

In the file, a symbol's **n_un.n_strx** field gives an index into the string table. An **n_strx** value of 0 indicates that no name is associated with a particular symbol table entry. The field **n_un.n_name** can be used to refer to the symbol name only if the program sets this up using **n_strx** and appropriate data from the string table.

The flag values that distinguish symbol types are defined in the header file **nlist.h**. The **n_type** field actually contains three fields, and if declared as such would be:

```
unsigned char  N_STAB:3,
               N_TYPE:4,
               N_EXT:1;
```

These fields are used by specifying the following masks:

```
#define N_STAB  0xe0  /* if any bits are set, this is a symbolic
                         debugging entry */
#define N_TYPE  0x1e  /* mask for the type bits */
#define N_EXT   0x01  /* external symbol bit; set for external
                         symbols */
```

Some of the N_STAB bits will be set if and only if the entry is a symbolic debugging entry (an **stab**)Ðin this case, the values for the N_TYPE bits of the **n_type** field (the entire field) are as shown in the header file **stab.h**. Normal values for the N_TYPE bits of the **n_type** field are:

```
#define N_UNDF  0x0   /* undefined; n_sect == NO_SECT */
#define N_ABS   0x2   /* absolute; n_sect == NO_SECT */
#define N_SECT  0xe   /* defined in section number n_sect */
#define N_INDR  0xa   /* indirect */
```

If the type is N_SECT, the **n_sect** field contains an ordinal of the section the symbol is defined in. The sections are numbered from 1 and refer to sections in the order in which they appear in the load commands for the file they're in. Therefore the same ordinal may refer to different sections in different files. This is the most common type of symbol.

If the type is N_INDR, the symbol is defined to be the same as another symbol. In this case the **n_value** field is an index into the string table of the other symbol's name. When the other symbol is defined, they both take on the defined type and value.

The **n_value** field for all symbol table entries (including N_STABs) gets updated by the link editor

based on the value of the **n_sect** field and where the section's **n_sect** references get relocated. If the value of the **n_sect** field is NO_SECT, its **n_value** field isn't relocated by the link editor.

```
#define NO_SECT     0   /* the symbol isn't in any section */
#define MAX_SECT  255   /* 1 through 255 inclusive */
```

Common symbols are represented by undefined (N_UNDF) external (N_EXT) types whose values (**n_value**) are nonzero. In this case the value of the **n_value** field is the size in bytes of the common symbol, and the value of the **n_sect** field is NO_SECT.

The LC_THREAD and LC_UNIXTHREAD Load Commands

Thread commands contain machine-specific data structures suitable for use in the thread state primitives. The machine-specific data structures follow the **struct thread_command** or **struct unixthread_command** as follows: Each flavor of machine-specific data structure is preceded by an unsigned long constant for the flavor of that data structure and an unsigned long that's the count of longs of the size of the state data structure, and then the state data structure follows that. This triple may be repeated for many flavors.

The constants for the **flavor**, **count**, and **state** data structure definitions are expected to be in the header file **machine/thread_status.h**; these machine-specific data structure sizes must be multiples of **sizeof(long)**. The **cmdsize** reflects the total size of the **thread_command** structure and all of the sizes of the constants for the **flavor**, **count**, and **state** data structures.

```
struct thread_command {
    unsigned long  cmd;           /* LC_THREAD or LC_UNIXTHREAD */
    unsigned long  cmdsize;       /* sizeof(struct thread_command) */
    /* unsigned long  flavor         flavor of thread state */
    /* unsigned long  count          count of longs in thread state */
    /* struct XXX_thread_state state   flavor's thread state */
    /* . . . */
};
```

The LC_UNIXTHREAD command specifies an initial thread execution state for a UNIX process. For an executable object that's a UNIX process, there's one **unixthread_command** created by the link editor. A stack is created based on the UNIX rlimit for the stack. This stack will contain the command arguments and environment variables when the program is executed. The entry point is placed in the program counter in the thread state. The stack address is placed in the stack pointer by the kernel when this program is executed. The stack is created as a zero-fill on demand region when the object is launched. Then the command line and environment arguments are placed on the stack and the stack pointer in the thread state is modified.

The LC_LOADFVMLIB and LC_IDFVMLIB Commands

A fixed virtual shared library has the file type MH_FVMLIB in the Mach header, and contains the **fvmlib_command** LC_IDFVMLIB to identify the library. An object that uses a fixed virtual shared library contains the **fvmlib_command** LC_LOADFVMLIB for each library it uses:

```
struct fvmlib_command {
    unsigned long  cmd;      /* LC_IDFVMLIB or LC_LOADFVMLIB */
    unsigned long  cmdsize;  /* includes pathname string */
    struct fvmlib  fvmlib;   /* the library identification */
};
```

Fixed virtual memory shared libraries are identified by the target pathname (the name of the library as found for execution) and the minor version number:

```
struct fvmlib {
    union lc_str   name;           /* library's target pathname */
    unsigned long  minor_version;  /* library's minor version
                                      number */
};
```

The LC_LOADFVMFILE Command

The LC_LOADFVMFILE command contains a reference to a file to be loaded at the specified virtual address:

```
struct fvmfile_command {
    unsigned long  cmd;            /* LC_FVMFILE */
    unsigned long  cmdsize         /* includes pathname string */
    union lc_str   name;           /* files pathname */
    unsigned long  header_addr;    /* files virtual address */
};
```

Relocation Information

The value of a byte in a section that isn't a portion of a reference to an undefined external symbol is exactly the value that will appear in memory when the file is executed. If a byte in a section involves a reference to an undefined external symbol, as indicated by the relocation information, the value stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added to the bytes in the file.

If relocation information is present, it amounts to eight bytes for each relocatable entry. The structure of a relocation entry as given in the header file **reloc.h** is as follows:

```
struct relocation_info {
    int       r_address;       /* offset in the section to what is
                                  being relocated */
    unsigned  r_symbolnum:24,  /* symbol index if r_extern == 1 or
                                  section ordinal if r_extern == 0 */
              r_pcrel:1,       /* was relocated pc-relative already */
              r_length:2,      /* 0=byte, 1=word, 2=long */
              r_extern:1,      /* doesn't include value of symbol
                                  referenced */
              r_reserved:4;    /* reserved */
};
#define R_ABS  0   /* absolute relocation type for Mach-O files */
```

The **r_address** is an offset rather than an address. For Mach-O object files this offset is from the start of the section the relocation entry is for.

If **r_extern** is 0, **r_symbolnum** is an ordinal representing the section that contains the symbol being relocated. These ordinals refer to the sections in the object file in the order in which their section structures appear in the headers of the object file they're in. The first section has the ordinal 1, the second has the ordinal 2, and so on. Therefore the same ordinal in two different object files could refer to two different sections. Furthermore, the ordinals could change when combined by the link editor. The value R_ABS is used for relocation entries of absolute symbols that need no further relocation.

To make scattered loading by the link editor work correctly, [a]local[o] relocation entries can't be used

when the item to be relocated is the value of a symbol plus an offset (where the resulting expression is outside the block the link editor is moving, blocks are divided at symbol addresses). If the item is a symbol value plus offset, the link editor needs to know more than just the section in which the symbol was defined. What is needed is the actual value of the symbol without the offset, so the link editor can do the relocation correctly based on where the value of the symbol got relocated to, not the value of the expression (with the offset added to the symbol value). For Release 2.0, no ªlocalº relocation entries are ever used when there is a nonzero offset added to a symbol. The ªexternalº and ªlocalº relocation entries remain unchanged.

It's assumed that a section will never be bigger than 2**24 - 1 (0x00ffffff or 16,777,215) bytes. This assumption allows the **r_address** (which is really an offset) to fit into 24 bits, and for the high bit of the **r_address** field in the **relocation_info** structure to indicate that it's really a **scattered_relocation_info** structure. Since these are only used in places where ªlocalº relocation entries are used and not where ªexternalº relocation entries are used, the **r_extern** field has been removed.

```
#define R_SCATTERED 0x80000000 /* mask to be applied to r_address
                                  field of a relocation_info struct
                                  to tell that it is really a
                                  scattered_relocation_info struct */
struct scattered_relocation_info {
  unsigned int r_scattered:1,  /* 1=scattered, 0=non-scattered */
               r_pcrel:1,      /* was relocated pc relative already */
               r_length:2,     /* 0=byte, 1=word, 2=long */
               r_reserved:4,   /* reserved */
               r_address:24;   /* offset in the section to what is
                                  being relocated */
  long         r_value;        /* the value the item to be relocated
                                  refers to (with no offset added) */
};
```

The Makeup of Executable Object Files

A typical executable (that is, with the filetype MH_EXECUTE) Mach-O object file produced by the link editor would contain the following components, in the order shown here:

- A Mach header
- An LC_SEGMENT load command for the **__PAGEZERO** segment
- An LC_SEGMENT load command for the **__TEXT** segment, followed by section headers for the sections in that segment. These section headers could include **__text**, **__fvmlib_init0**, **__fvmlib_init1**, **__const**, **__string**, **__literal8**, and **__literal4**.
- An LC_SEGMENT load command for the **__DATA** segment, followed by the section headers for the sections in that segment. These section headers could include **__data**, **__bss**, and **__common**.
- An LC_SEGMENT load command for the **__OBJC** segment, followed by the section headers for the sections in that segment. These section headers could include **__class**, **__meta_class**, **__cat_inst_meth**, **__els_meth**, **__inst_meth**, **__message_refs**, **__symbols**, **__category**, **__class_vars**, **__module_info**, and **__selector_strs**.
- An LC_SEGMENT load command for the **__LINKEDIT** segment
- An LC_SYMTAB load command
- An LC_UNIXTHREAD load command
- An LC_LOADFMVLIB load command for each shared library it uses

- The **__TEXT** segment rounded out to the segment alignment
- The **__DATA** segment rounded out to the segment alignment
- The **__OBJC** segment rounded out to the segment alignment
- All the relocation entries, if saved (normally not saved)
- All the **stab** symbol and string tables, if not stripped

You can use the **otool** command to print the contents of object files and libraries that are in Mach-O format or in UNIX 4.3BSD **a.out** format. Various options allow you to specify certain portions of the Mach-O file. For example:

-h	Print the Mach header
-l	Print the load commands
-t	Print the contents of the **__text** section (used with the **-v** flag, this disassembles the text; with the **-V** flag it also symbolically disassembles the operands)
-d	Print the contents of the **__data** section
-r	Print the relocation entries

Complete documentation for the **otool** command is contained in a UNIX manual page, which you can access through the Digital Librarian.

Additional information related to the Mach-O file format is contained in section 1 (commands), section 3 (subroutines), and section 5 (file formats and conventions) of the UNIX manual pages. You can use the following list and the Digital Librarian to find the documentation you need:

atom(1)	Converts an object file from **a.out** to Mach-O format
gdb(1)	Debugs using the GNU debugger
ld(1)	Links using the link editor
nm(1)	Prints a symbol table
otool(1)	Prints parts of an object file or library
size(1)	Prints the size of an object file
strip(1)	Removes symbols and relocation bits
getmachheaders(3)	Gets the Mach headers for an executable
getsectbyname(3)	Gets the section information for a section
getsegbyname(3)	Gets the segment command for a segment
nlist(3)	Gets entries from a name list
Mach-O(5)	Describes Mach-O assembler and link editor output
stab(5)	Describes symbol table types