NAME

coff - Common Object File Format Disassembler/Dumper

SYNOPSIS

coff [-acdefhilmnopstvxDT] [+asm] [+nodefault] [+header] [+help] [+hex] [+infix] [+label] [+merlin] [+noheader] [+nooffset] [+orca] [+postfix] [+shorta] [+shorti] [+tool] [+version] [+compress] [+exact] [+thanks] *file* [*segments...*] [*loadsegments...*]

DESCRIPTION

This manual page documents **coff**. **coff** outputs Apple IIgs OMF (Object Module Format) files in either OMF format (dump) or as 65816 object code (disassembly). **coff** takes file *file* as an argument and outputs the disassembly to the standard output. If *segment* or *loadsegment* arguments are given, those segments/loadsegments are output in place of the entire file (*segment* and *loadsegment* do not have to match the segment/loadsegment names in the OMF file exactly).

OPTIONS

-a, +shorta
: **coff** assumes it will be disassembling an OMF file that was created for the Apple IIgs computer. Therefore, it assumes 16-bit addressing mode for the accumulator. This option disables 16-bit addressing mode for the accumulator and assumes 8-bit addressing mode.

-d, +asm
: Disassemble output of OMF file. 65816 code is output. This option is the same as +merlin as **coff** uses the Merlin 16+ assembler format as its default 65816 disassembly format.

-f, +nooffset
: Output for the body of OMF files is given by a displacement value into the file and an object displacement into the file. The displacement value is the actual position in the file where the OMF record occurs. The object displacement is the number of bytes of code the segment will link to. Enable this option to disable this output.

-h, +help
: Display list of options for **coff**. The list contains the short and long options in addition to a brief description of each option.

-i, +shorti
: **coff** assumes it will be disassembling an OMF file that was created for the Apple IIgs computer. Therefore, it assumes 16-bit addressing mode for the index registers. This option disables 16-bit addressing mode for the index registers and assumes 8-bit addressing mode.

-l, +label
: In assembly language programs certain routines have labels associated with them (i.e. `jsr print`). When this is assembled into an OMF file, the label '`print`' is replaced by an offset from the beginning of the segment it is contained in. Enabling this option will cause **coff** to output the name of the label, '`print`', rather than its offset, '<segname>+<offset>'.

-m, +merlin
: 65816 code is output in Merlin 16+ format. This is the default format for the +asm option.

-n, +noheader
: Default is to output the header of each segment. Enable this option to disable this default.

-o, +orca
: 65816 code is output in Orca/M format.

-p, +postfix
: Expressions are output in postfix notation (i.e. '`lda x+y`' will be output as '`lda x y +`'.

-s, +header
: Only the headers for each segment are output. If the +hex option is added to this option, output of the headers id displayed in hex format, as opposed to the usual text representation of the headers given in OMF disassembly.

-t, +infix
: Expressions are output in infix notation.

-v, +version
: Display version number of **coff**.

-x, +hex
: Dump OMF segment body in hexadecimal. File displacement is given but no code displacement. ASCII values of each hex digit is output to the right of each line. If used with the +asm option, outputs the hex and ASCII code of each disassembled line (useful for determining where ASCII code is in a program).

-D, +nodefault
: Disable default options attached with the **coff** program. The options are stored in the resource fork. Default options are read in last, if +nodefault is not enabled. Thus, it is not possible to disable an option in the list of default options. Instead, you must enable this option and all options in the default option list minus the option(s) not required.

-T, +tool
: Interprets certain hex code sequences and displays them as Toolbox calls. Also inter-

prets certain ROM and RAM addresses into their english equivalents (i.e. BREAK, HOME, WAIT, KEYIN).

-c, +compress
Two OMF record with the largest output are **CONST** and **LCONST** records. These records contain byte sequences that are not interpreted by the linker. They are included directly into the executable file. When disassembling an executable file such as an EXE or S16 type file, these records often contain large amounts of data. Enabling this option will only display the length of the bytes generated by the records.

-e, +exact
When a *segment* or *loadsegment* option is given to **coff** on the command line, it is often convenient to display all segment/loadsegments that begin with the *segment* or *loadsegment* given on the command line. This is **coff**'s default option. However, in case an exact match is needed for the command-line options, enable this option.

+thanks
List of individuals who helped on the **coff** project.

EXAMPLES

The following is sample output with no options enabled:

```
block count   :  $00000001                                   1
reserved space:  $00000000                                   0
length        :  $00008045                               32837
label length  :  $00                                         0
number length :  $04                                         4
version       :  $01                                         1
bank size     :  $00010000                               65536
kind          :  $00                               static code
org           :  $00000000                                   0
alignment     :  $00000000                                   0
number sex    :  $00                                         0
segment number:  $0000                                       0
entry         :  $00000000                                   0
disp to names :  $002c                                      44
disp to data  :  $003a                                      58
load name     :
segment name  :  one
00003a 000000 | USING     (e4) | common
000042 000000 | CONST     (04) | Length:  4 ($4) bytes
000043 000000 |                | f4 34 12 22                   - .4."
000047 000004 | LEXPR     (f3) | truncate result to 3 bytes
00004f 000004 |                | (one+$8041)
00004f 000007 | CONST     (06) | Length:  6 ($6) bytes
000050 000007 |                | 22 a8 00 e1 10 20             - "....
000056 00000d | EXPR      (eb) | truncate result to 4 bytes
00005e 00000d |                | (one+$8041)
00005e 000011 | DS        (f1) | insert 32768 zeros
000064 008011 | CONST     (03) | Length:  3 ($3) bytes
000065 008011 |                | 74 65 ad                      - te.
000068 008014 | BEXPR     (ed) | truncate result to 2 bytes
00006e 008014 |                | t
00006e 008016 | CONST     (1f) | Length:  31 ($1f) bytes
00006f 008016 |                | a9 00 00 10 fb 10 00 a9 00 00 - ..........
000079 008020 |                | 10 17 6a a7 04 a9 34 12 20 34 - ..j...4.  4
000083 00802a |                | 12 af 56 34 12 22 a8 00 e1 10 - ..V4."....
00008d 008034 |                | 20                            -
00008e 008035 | EXPR      (eb) | truncate result to 4 bytes
000096 008035 |                | (one+$8041)
000096 008039 | CONST     (0c) | Length:  12 ($c) bytes
000097 008039 |                | 4a c9 00 00 d0 fa f0 f8 c0 00 - J.........
0000a1 008043 |                | 00 b2                         - ..
0000a3 008045 | END       (00)
```

The header of the output gives a detailed description of the contents of the body of the OMF segment. The body of the OMF segment consists of the byte offset (offset into file), code offset, OMF types and their hex values, and a detailed description of each OMF type in the segment. All **CONST, LCONST** records are displayed with their hex values and ASCII equivalent of those values, as indicated above. If a hex value does not have an ASCII equivalent (7-bit ASCII), it is replaced with a period.

The following is sample output with the +hex option enabled:

```
block count   :  $00000001                                 1
reserved space:  $00000000                                 0
length        :  $00000079                               121
label length  :  $00                                       0
number length :  $04                                       4
version       :  $01                                       1
bank size     :  $00010000                             65536
kind          :  $00                             static code
org           :  $00000000                                 0
alignment     :  $00000000                                 0
number sex    :  $00                                       0
segment number:  $0000                                     0
entry         :  $00000000                                 0
disp to names :  $002c                                    44
disp to data  :  $003a                                    58
load name     :
segment name  :  two
00023a | 4d f4 04 1a 22 00 00 e1 af 20 00 e1 a2 13 20 - M...".... ....
000249 | 22 b0 00 e1 f4 ff ff 22 00 00 e1 a2 ff ff 22 - "......"......"
000258 | b0 00 e1 a9 00 00 ad 00 c0 af 00 c0 e0 8d 01 - ...............
000267 | c0 af 56 34 12 af 80 00 e1 ad ff cf 22 a8 00 - ..V4......."..
000276 | e1 13 20 78 56 34 12 ea 20 00 bf c0 34 12 20 - .. xV4.. ...4.
000285 | 00 bf c0 eb 02 83 04 70 61 72 6d 00 03 20 00 - .......parm.. .
000294 | bf eb 01 83 04 63 61 6c 6c 00 eb 02 83 04 70 - .....call.....p
0002a3 | 61 72 6d 00 06 22 a8 00 e1 13 20 eb 04 83 04 - arm..".... ....
0002b2 | 70 61 72 6d 00 04 22 a8 00 e1 eb 02 83 04 63 - parm..".......c
0002c1 | 61 6c 6c 00 eb 04 83 04 70 61 72 6d 00 01 af - all.....parm...
0002d0 | eb 03 83 06 69 6e 6c 69 6e 65 81 01 00 00 00 - ....inline.....
0002df | 01 00 01 bf eb 03 83 06 69 6e 6c 69 6e 65 81 - ........inline.
0002ee | 01 00 00 00 01 00 eb 02 83 04 63 61 6c 6c 81 - ..........call.
0002fd | 01 00 00 00 01 00 eb 04 83 04 70 61 72 6d 81 - ..........parm.
00030c | 01 00 00 00 01 00 02 59 53 00 6c 69 6e 65 81 - .......YS.line.
00031b | 01 00 00 00 01 00 01 bf eb 03 83 06 69 6e 6c - ............inl
00032a | 69 6e 65 81 01 00 00 00 01 00 eb 02 83 04 63 - ine...........c
000339 | 61 6c 6c 81 01 00 00 00 01 00 eb 04 83 04 70 - all...........p
000348 | 61 72 6d 81 01 00 00 00 01 00 02 59 53 00 6c - arm........YS.l
000357 | 69 6e 65 81 01 00 00 00 01 00 01 bf eb 03 83 - ine............
000366 | 06 69 6e 6c 69 6e 65 81 01 00 00 00 01 00 eb - .inline........
000375 | 02 83 04 63 61 6c 6c 81 01 00 00 00 01 00 eb - ...call........
000384 | 04 83 04 70 61 72 6d 81 01 00 00 00 01 00 02 - ...parm........
000393 | 59 53 00 6c 69 6e 65 81 01 00 00 00 01 00 01 - YS.line........
0003a2 | bf eb 03 83 06 69 6e 6c 69 6e 65 81 01 00 00 - .....inline....
0003b1 | 00 01 00 eb 02 83 04 63 61 6c 6c 81 01 00 00 - .......call....
0003c0 | 00 01 00 eb 04 83 04 70 61 72 6d 81 01 00 00 - .......parm....
0003cf | 00 01 00 02 59 53 00 6c 69 6e 65 81 01 00 00 - ....YS.line....
0003de | 00 01 00 01 bf eb 03 83 06 69 6e 6c 69 6e 65 - .........inline
0003ed | 81 01 00 00 00 01 00 eb 02 83 04 63 61 6c 6c - ...........call
0003fc | 81 01 00 00 01 00 00 00 00 00 00 00 28 00 00 - ............(..
```

This is the output of the same segment as on the previous page. As discussed before, when the +hex option is enabled, only the byte offset into the file is given. The hex codes of the OMF body are output along with their ASCII equivalent to the right of each line.

The following is sample output with the +orca and +hex options enabled (default options are read in when **coff** starts):

```
block count   :  $00000001                                  1
reserved space:  $00000000                                  0
length        :  $00000079                                121
label length  :  $00                                        0
number length :  $04                                        4
version       :  $01                                        1
bank size     :  $00010000                              65536
kind          :  $00                              static code
org           :  $00000000                                  0
alignment     :  $00000000                                  0
number sex    :  $00                                        0
segment number:  $0000                                      0
entry         :  $00000000                                  0
disp to names :  $002c                                     44
disp to data  :  $003a                                     58
load name     :
segment name  :  two
00023a 000000 |             longa  on
00023a 000000 |             longi  on
00023a 000000 | two         start
00023a 000000 |             _ClosePort
000242 000007 |             lda    IRQ.APTALK                 af 20 00 e1 - .  ..
000246 00000b |             _WriteGS
00024d 000012 |             pea    $ffff                      f4 ff ff    - ...
000250 000015 |             jsl    DISPATCH1                  22 00 00 e1 - "...
000254 000019 |             ldx    #$ffff                     a2 ff ff    - ...
000257 00001c |             jsl    GS/OS                      22 b0 00 e1 - "...
00025b 000020 |             lda    #$0000                     a9 00 00    - ...
00025e 000023 |             lda    IOADR                      ad 00 c0    - ...
000261 000026 |             lda    e0_IOADR                   af 00 c0 e0 - ....
000265 00002a |             sta    SET80COL                   8d 01 c0    - ...
000268 00002d |             lda    |$123456                   af 56 34 12 - .V4.
00026c 000031 |             lda    TOWRITEBR                  af 80 00 e1 - ....
000270 000035 |             lda    $cfff                      ad ff cf    - ...
000273 000038 |             _WriteGS $345678
00027d 000042 |             nop                               ea          - .
00027e 000043 |             _CreateP8 $1234
000284 000049 |             _CreateP8 parm
000291 00004f |             jsr    PRO8MLI                    20 00 bf    -  ..
000295 000052 |             dc     i1'call'
00029e 000053 |             dc     i2'parm'
0002a7 000055 |             _WriteGS parm
0002b7 00005f |             jsl    PRO16MLI                   22 a8 00 e1 - "...
0002bc 000063 |             dc     i2'call'
0002c5 000065 |             dc     i4'parm'
0002ce 000069 |             lda    |inline + $1
0002e0 00006d |             lda    |inline + $1,x
0002f2 000071 |             dc     i2'call + $1'
000301 000073 |             dc     i4'parm + $1'
000310 000077 |             dc     h'5953'                    59 53       - YS
000313 000079 |             end
```

The +default option enables the '+tool, +infix, +label' options. The +tool options enable **coff** to interpret certain hex sequences as toolbox calls. Only the toolbox calls are interpreted, not the parameters passed to any toolbox call. The +infix option enables expressions to be displayed in infix format. Selecting +orca without any other options enables the default postfix option. It is easier to read the disassembly when it is in infix form. Also, if you enable infix notation, parentheses in expressions during the conversion from postfix to infix are minimized. This will be apparent in the next example. The +tool option also recognizes the ROM addresses such as SET80COL, TOWRITEBR, and PRO16MLI. If the address is accessed in bank $e0, the call is preceded with 'e0_' as IOADR is above. GS/OS and ProDOS calls are also recognized.

The +hex option that, if enabled with any of the disassembly options, dom, will display the hex and ASCII value of the instruction being displayed. This is useful for displaying ASCII strings contained in the code. While the disassembly will not make much sense while displaying the ASCII data, the +hex option does provide a more meaningful way to interpret why the disassembly does not make sense.

The following is sample output with the +orca option enabled:

```
block count   :  $00000002                              2
reserved space:  $00000000                              0
length        :  $000001a5                            421
label length  :  $00                                    0
number length :  $04                                    4
version       :  $01                                    1
bank size     :  $00000000                              0
kind          :  $01                          static data
org           :  $00000000                              0
alignment     :  $00000000                              0
number sex    :  $00                                    0
segment number:  $0000                                  0
entry         :  $00000000                              0
disp to names :  $002c                                 44
disp to data  :  $003d                                 61
load name     :  ~globals
segment name  :  common

00003d 000000 |             longa  on
00003d 000000 |             longi  on
00003d 000000 | common      data
00003d 000000 | aa          equ    $30
000049 000000 | bb          equ    z + $1
000059 000000 | cc          equ    q1
000064 000000 | a           dc     h'0102030405060708090a0b0c0d0e0f1011121314'
00007a 000014 |             dc     h'1516171819202122'
000086 00001c | b           dc     i1'254,10,12,13,14,15,16,17,18,19,20'
000093 000027 |             dc     i1'21,22,23,24,25,26,27,28,29,30'
0000a1 000031 | c           dc     f'1.1234,2.3456,3.45679'
0000af 00003d |             dc     f'4.4,5.5,6.12346'
0000bb 000049 |             dc     f'7.1234'
0000c3 00004d | d           dc     d'1.1,2.2,3.3'
0000dd 000065 |             dc     d'4.4,5.5,6.6'
0000f5 00007d |             dc     d'7.7,8.8,9.9'
00010d 000095 |             dc     d'10.1'
000119 00009d | e           dc     c'now is the time for all of us to leave i'
000143 0000c5 |             dc     c't and all the'
000154 0000d2 | f           ds     200
00015f 00019a | g           dc     i1'a + b + c + f'
00017e 00019b | h           dc     i3'q - (r - s)'
000191 00019e | i           dc     i1'a'
00019e 00019f | j           dc     i2'a * ((b + c) * d * e + f)'
0001c9 0001a1 | k           dc     r'reference'
0001d8 0001a1 | l           dc     s'soft_reference'
0001ef 0001a3 | m           equ    $0
0001fb 0001a3 | n           equ    a + b + c
000213 0001a3 | o           dc     i2'albert'
000223 0001a5 |             end
```

As indicated before, the +infix option minimizes the parentheses in the output during the postfix to infix conversion. This takes out much confusion in reading the expression. It is also necessary when disassembling addressing modes such as Direct Page Indirect, Direct Page Indexed Indirect,X, and Stack Relative Indirect Indexed,Y. In the above output, the +label option has been enabled by the +default switch. Thus, when **coff** encounters a label (either `GLOBAL` or `LOCAL`), it saves it and, if the label is referenced again, the name of the label is used in place of the default output.

When disassembling labels, output is formatted in either Orca/M or Merlin as indicated by the user. However, certain type attributes of labels, such as double-precision floating-point-type or floating-point-type are not supported by the Merlin assembler and are output in Orca/M format. When disassembling integer-type labels, all integers are assumed to be signed.

The following is sample with the +header option enabled:

```
byte count    :  $0000065c                           1628
reserved space:  $00000000                              0
length        :  $000003e2                            994
label length  :  $00                                    0
number length :  $04                                    4
version       :  $02                                    2
bank size     :  $00010000                          65536
kind          :  $2000          static position-independent
code
org           :  $00000000                              0
alignment     :  $00000000                              0
number sex    :  $00                                    0
segment number:  $0002                                  2
entry         :  $00000000                              0
disp to names :  $002c                                 44
disp to data  :  $003b                                 59
load name     :
segment name  :  INIT

byte count    :  $00005ded                          24045
reserved space:  $00000000                              0
length        :  $00005627                          22055
label length  :  $00                                    0
number length :  $04                                    4
version       :  $02                                    2
bank size     :  $00010000                          65536
kind          :  $0000                        static code
org           :  $00000000                              0
alignment     :  $00000000                              0
number sex    :  $00                                    0
segment number:  $0003                                  3
entry         :  $00000000                              0
disp to names :  $002c                                 44
disp to data  :  $003b                                 59
load name     :
segment name  :  MAIN
```

Headers can be displayed in both disassembled and hex format. The above display is the disassembly output. The hex output is similar to the +hex option but applies to the header. Below is a sample of this:

```
00012d | 5c 06 00 00 00 00 00 00 e2 03 00 00 00 00 04 - \..............
00013c | 02 00 00 01 00 00 20 00 00 00 00 00 00 00 00 - ...... ........
00014b | 00 00 00 00 02 00 00 00 00 00 2c 00 3b 00 00 - ..........,.;..
00015a | 00 00 00 00 00 00 00 00 00 04 49 4e 49 54    - ..........INIT

000789 | ed 5d 00 00 00 00 00 00 27 56 00 00 00 00 04 - .]......'V.....
000798 | 02 00 00 01 00 00 00 00 00 00 00 00 00 00 00 - ...............
0007a7 | 00 00 00 00 03 00 00 00 00 00 2c 00 3b 00 00 - ..........,.;..
0007b6 | 00 00 00 00 00 00 00 00 00 04 4d 41 49 4e    - ..........MAIN
```

OMF FORMAT

The OMF (Object Module Format) file format is a specification for code object files output by assemblers and compilers. OMF files consist of object files, library files, load files, and run-time library files. Each OMF file consists of segments which are composed of a header and OMF records. The general format of an OMF file is given in Figure 1.

Segment types and attributes

Each OMF segment has a segment type and can have several attributes. The following segment types are defined by OMF:

- **Code** and **Data** segments are object segments provided to support languages (such as assembly language) that distinguish program code from data.
- **Jump-table** segments and pathname segments are load segments that facilitate the dynamic loading of segments.
- **Pathname segment**.
- **Library dictionary** segments allow the linker to scan library files quickly for needed segments.
- **Initialization segments** are optional parts of load files that are used to perform any initialization required by the application during an initial load. If used, they are loaded and executed immediately as the System Loader encounters them and are re-executed any time the program is restarted from memory.
- **Direct-page/stack** segments are load segments used to preset the location and contents of the direct page and stack for an application.

A Segment can have only one segment *type* but can have any combination of *attributes*. The following segment attributes are defind by the object module format:

- **Reload** segments are load segments that the loader must reload even if the program is restartable and is restarted from memory. They usually contain data that must be restored to its initial values before a program can be restarted.
- **Absolute-bank** segments are load segments that are restricted to a specified bank but that can be relocated within that bank. The **org** field in the segment header specifies the bank to which the segment is restricted.
- **Loadable in special memory** means that a segment can be loaded in banks $00, $01, $E0, and $E1. Because these are the banks used by programs running under ProDOS 8 in standard-Apple II emulation mode, you may prevent your program from being loaded in these banks so that it can remain in memory while programs are run under ProDOS 8.
- **Position-independent** segments can be moved by the Memory Manager during program execution if they have been unlocked by the program.

- **private code** segment is a code segment whose name is available only to other code segments within the same object file (The labels within a code segment are local to that segment).
- A **private data** segment is a data segment whose labels are available only to code segments in the same object file.
- **Static** segments are load segments that are loaded at program execution time and are not unloaded during execution; **dynamic** segments are loaded and unloaded during program execution as needed.
- **Bank-relative** segments must be loaded at a specified address within any bank. The **org** field in the segment header specifies the bank-relative address (the address must be less than $10000).
- **Skip** segments will not be linked by the linker or loaded by the System Loader. However, all references to global definitions in a Skip object segment will be processed by a linker.

Segment Header

Each segment in an OMF file has a header that contains general information about the segment, such as its name and length. The format of the segment header is illustrated in Figure 2. The following is a detailed description of the fields that comprise the segment header:

bytecnt A 4-byte field indicating the number of bytes in the file that the segment requires. This number includes the segment header, so you can calculate the starting mark of the next segment from the starting mark of this segment plus **bytecnt** (OMF 1.0 segments are a multiple of 512 bytes). Segments need not be aliged to block boundaries.

resspc A 4-byte field specifying the number of bytes of 0's to add to the end of the segment. This field can be used in an object segment instead of a large blocks of zeros at the end of the segment. This field duplicates the effect of a **ds** record at the end of the segment.

length A 4-byte field field specifying the memory size that the segment will require when loaded. It includes the extra memory specified by **resspc**.

length is followed by one undefined byte. and aldsjf lkjdsaf ljadsf ldfkj als fdjlasd jflasjd flasjdf lakjsdf lkasjdf lkajsdf lkdsafasdkfj lkasjdf lkjdsa flkjasd flsaj fljas dflajsd flksaj flksaj flaksjd flksajdf

lablen A 1-byte field indicating the length, in bytes, of each name or label record in the segment body. If **lablen** is 0, the length of each name of label is specified in the first byte of the record (that is, the first byte of the record specifies how many bytes follow). **lablen** also specifies the length of the **segname** field of the segment header, or, if **lablen** is 0, the first byte of **segname** specifies how many bytes follow. (The **loadname** field always has a length of 10 bytes).

Fixed-length labels are always left justified and padded with spaces.

numlen A 1-byte field indicating the length, in bytes, of each number field in the segment body. This field is 4 bytes for the Apple IIgs.

version A 1-byte field indicating the version number of the object module format with which the segment is compatible. OMF is currently at version 2.0.

revision A 1-byte field indicating the revision number of the object module format with which the segment is compatible. Together with the **version** field, **revision** specifies the OMF compatibility level of this segment. OMF is current at revision 0.

banksize A 4-byte binary number indicating the maximum memory-bank size for the segment. If the segment is in an object file, the linker ensures that the segment is not larger that this value. (The linker returns an error if the segment is too large). If the segment is in a load file, the loader ensures that the segment is loaded into a memory block that does not cross this boundary. For Apple IIgs code segments, this field must be $00010000, indicating a 64K bank size. A value of 0 in this field indicates that the segment can cross bank boundaries. Apple IIgs data segments can use any number from $00 to $00010000 for **banksize.**

kind A 2-byte field specifying the type and attributes of the segment. The bits are defined as shown in **Table 1**.

Important: If segment **kind**s are specified in the source file, and the **kind**s of the object segments placed in a given load segment are not all the same, the segment **kind** of the first object segment determines the segment **kind** of the entire load segment.

kind is followed by two undefined bytes, reserved for future changes to the segment header specification.

Table1: **kind** field definition

Bit(s)	**Values**	**Meaning**
0-4		*Segment Type subfield*
	$00	Code
	$01	Date
	$02	Jump-table segment
	$04	Pathname segment
	$08	Library dictionary segment
	$10	Initialization Segment
	$12	Direct-page/stack segment
8-15		*Segment Attributes bits*
8	if = 1	Bank-relative segment
9	if = 1	Skip segment
10	if = 1	Reload segment
11	if = 1	Absolute-bank segment
12	if = 0	Can be loaded in special memory
13	if = 1	Position independent
14	if = 1	Private
15	if = 0	Static; otherwise dynamic

org A 4-byte field indicating the absolute address at which this segment is to be loaded in memory, or, for an absolute-bank segment, the bank number. A value of 0 indicates that this segment is relocatable and can be loaded anywhere in memory. A value of 0 is normal for the Apple IIgs.

align A 4-byte binary number indicating the boundary on which this segment must be aligned. For example, if the segment is to be aligned on a page boundary, this field is $00000100; if the segment is to be aligned on a bank boundary, this field is $00010000. A value of 0 indicates that no alignment is needed. For the Apple IIgs, this field must be a power of 2, less than or equal to $00010000.

numsex A 1-byte field indicating the order of the bytes in a number field. If this field is 0, the least significant byte is first. If this field is 1, the most significant byte is first. This field is set to 0 for the Apple IIgs.

numsex is followed by one undefined byte, reserved for future changes to the segment header specification.

segnum A 2-byte field specifying the segment number. The segment number corresponds to the relative position of the segment in the file (starting with 1).

entry A 4-byte field indicating the offset into the segment that corresponds to the entry point of the segment.

dispname A 2-byte field indicating the displacement of the **loadname** field within the segment header. Currently, **dispname** = 44. **dispname** is provided to allow for future additions to the segment header; any new fields will be added between **dispdata** and **loadname**. **dispname** allows you to reference **loadname** and **segname** no matter what the actual size of the header.

dispdata	A 2-byte field indicating the displacement from the start of the segment header to the start of the segment body. **dispdata** is provided to allow for future additions to the segment header; any new fields will be added between **dispdata** and **loadname**. **dispdata** allows you to reference the start of the segment body no matter what the actual size of the header.
tempORG	A 4-byte field indicating the temporary origin of the Object segment. A nonzero value indicates that all references to globals within this segment will be interpreted as if the Object segment started at that location. However, the actual load address of the Object segment is still determined by the **org** field.
loadname	A 10-byte field specifying the name of the load segment that will contain the code generated by the linker for this segment. More than one segment in an object file can be merged by the linker into a single segment in the load file. This field is unused in a load segment. The position of **loadname** may change in future versions of the OMF; therefore, you should always use **dispname** to reference **loadname.**
segname	A field that is **lablen** bytes long, and that specifies the name of the segment. The position of **segname** may change in future revisions of the OMF; therefore, you should always use **dispname** to reference **segname.**

Segment Body

The body of each segment is composed of sequential records, each of which starts with a 1-byte operation code. Each record contains either program code or information for the linker or loader. All names and labels included in these record are **lablen** bytes long, and all numbers and addresses are **numlen** bytes long (unless otherwise specified in the following definitions).

Several of the OMF records contain expressions that have to be evaluated by the linker. The operation and syntax of expressions are described in **Expressions**. If the description of the record type does not explicitly state that the opcode is followed by an expression, then an expression cannot be used. Expressions are never used in load segments.

The operation codes and segment records are described below, listed in order of the opcodes. **Table 2** provides an alphabetical cross-reference between segment record types and opcodes. Library files consist of object segments and so can use any record type that can be used in an object segment. The table also lists the segment types in which each record type can be used.

Table 2: Segment-body record types

Record Type	Opcode	Found in what segment types	Record Type	Opcode	Found in what segment types
ALIGN	$E0	Object	**BEXPR**	$ED	Object
cINTERSEG	$F6	Load	**CONST**	$01-$DF	Object
cRELOC	$F5	Load	**DS**	$F1	All
END	$00	All	**ENTRY**	$F4	Run-time Library Dictionary
EQU	$F0	Object	**EXPR**	$EB	Object
GEQU	$E7	Object	**GLOBAL**	$E6	Object
INTERSEG	$E3	All	**LCONST**	$F2	All
LEXPR	$F3	Object	**LOCAL**	$EF	Object
MEM	$E8	Object	**ORG**	$E1	Object
RELEXPR	$EE	Object	**RELOC**	$E2	Load
STRONG	$E5	Object	**SUPER**	$F7	Load
USING	$E4	Object	**ZEXPR**	$EC	Object

END	$00	This record indicates the end of the segment.
CONST	$01-$DF	This record contains absolute data that needs no relocation. The operation code specifies how many bytes of data follow.
ALIGN	$E0	This record contains a number that indicates an alignment factor. The linker inserts as many 0 bytes as necessary to move to the memory boundary indicated by this factor. The value of this factor is in the same format as the **ALIGN** field in the segment header and cannot have a value greater than that in the **ALIGN** field. **ALIGN** must equal a power of 2.
ORG	$E1	This record contains a number that is used to increment or decrement the location counter. If the location counter is incremented (**ORG** is positive), 0's are inserted to get to the new address. If the location counter is decremented (**ORG** is a complement negative number of 2), subsequent code overwrites the old code.
RELOC	$E2	This is a relocation record, which is used in three relocation dictionary of a load segment. It is used to patch an address in a load segment with a reference to another address in the same load segment. It contains two 1-byte counts followed by two offsets. The first count is the number of bytes to be relocated. The second count is a bit-shift operator, telling how many times to shift the relocated address before inserting the result into memory. If the bit-shift operator is positive, the number is shifted to the left, filling vacated bit positions with 0's (arithmetic shift left). If the bit-shift operator is (two's

complement) negative, the number is shifted right (logical shift right) and 0-filled.

The first offset gives the location (relative to the start of the segment) of the first byte of the number that is to be patched (relocated). The second offset is the location of the reference relative to the start of the segment; that is, it is the value that the number would have if the segment containing it started at address $000000. For example, suppose the segment includes the following lines:

```
35    label        •••
      •
      •
      •
400   lda          label+4
```

The **RELOC** record contains a patch to the operand of the `lda` instruction. The value of the patch is `label+4`, so the value of the last field in the **RELOC** record is $39-the value the patch would have if the segment started at address $000000. `label+4` is two bytes long; that is, the number of bytes to be relocated is 2. No bit-shift operation is needed. The location of the patch is 1025 ($401) bytes after the start of the segment (immediately after the `lda`, which is one byte).

The **RELOC** record for the number to be loaded into the A register by this statement would therefore look like this (note that the values are stored low byte first, as specified by **numsex**):

```
E2020001 04000039 000000
```

This sequence corresponds to the following values:

$E2	operation code
$02	number of bytes to be relocated
$00	bit-shift operator
$00000401	offset of value from start of segment
$00000039	value if segment started at $000000

Note: Certain types of arithmetic expressions are illegal in a relocatable segment; specifically, any expression that the assembler cannot evaluate (relative to the start of the segment) cannot be used. The expression `LAB|4` can be evaluated, for example, since the **RELOC** record includes a bit-shift operator. The expression `LAB|4+4` cannot be used, however, because the assembler would have to know the absolute value of `LAB` to perform the bit-shift operation before adding 4 to it. Similarly, the value of `LAB*4` depends on the absolute value of `LAB` and cannot be evaluated relative to the start of the segment, so multiplication is illegal in expressions in relocatable segments.

INTERSEG $E3

This record is used in the relocation dictionary of a load segment. It contains a patch to a long call to an external reference; that is, the **INTERSEG** record is used to patch an address in a load segment with a reference to another address in a different load segment. It contains two 1-byte counts followed by an offset, a 2-byte file number, a 2-byte segment number, and a second offset. The first count is the number of bytes to be relocated, and the second count is a bit-shift operator, telling how many times to shift the relocated address before inserting the result into memory. If the bit-shift operator is positive, the number is shifted to the left, filling vacated bit positions with 0's (arithmetic shift left). If the bit-shift operator is (two's complement) negative, the number is shifted right (logical shift right) and 0-filled.

The first offset is the location (relative to the start of the segment) of the (first byte of the) number that is to be relocated. If the reference is to a static segment, the file number, segment number, and second offset correspond to the subroutine referenced. (The linker assigns a file number to each load file in a program. This feature is provided primarily to support run-time libraries. In the normal case of a program having one load file, the file number is 1. The load segments in a load file are numbered by their relative locations in the laod file, where the frist load segment is number 1). If the reference is to a dynamic segment, the file and segment number correspond to the jump-table segment, and the second offset corresponds to the call to the Loader for that reference.

For example, suppose the segment includes an instruction such as:

```
jsl         ext
```

The label `ext` is an external reference to a location in a static segment.

If this instruction is at relative address $720 within its segment and ext is at relative address $345 in segment $000a in file $0001, the linker creates an **INTERSEG** record in the relocation dictionary that looks like this (note that the values are stored low byte first, as specified by **NUMSEX**):

```
E3030021 07000001 000A0045 030000
```

This sequence corresponds to the following values:

`$E3`	operation code
`$03`	number of bytes to be relocated
`$00`	bit-shift operator
`$00000721`	offset of instruction's operand
`$0001`	file number
`$000A`	segment number
`$00000345`	offset of subroutine referenced

When the loader processes the relocation dictionary, it uses the first offset to find the `jsl` and patches in the address corresponding to the file number, segment number, and offset of the referenced subroutine.

If the `jsl` is to an external reference in a dynamic segment, the **INTERSEG** records refer to the file number, segment number, and offset of the call to the Loader in the jump-table segment.

If the jump-table segment is in segment 6 of file 1, and the call to the Loader is at relative location $2A45 in the jump-table segment, then the **INTERSEG** record looks like this (note that the values are stored low byte first, as specified by **NUMSEX**):

```
E3030021 07000001 00060045 2A0000
```

This sequence corresponds to the following values:

`$E3`	operation code
`$03`	number of bytes to be relocated
`$00`	bit-shift operator
`$00000721`	offset of instruction's operand
`$0001`	file number of jump-table segment
`$0006`	segment number of jump-table segment
`$00002A45`	offset of call to loader

The jump-table segment entry that corresponds to the external reference `ext` contains the following values:

User ID

`$0001`	file number
`$0005`	segment number
`$00000200`	offset of instruction call to Loader

INTERSEG records are used for any long-address reference to a static segment.

USING	$E4	This record contains the name of a data segment. After this record is encountered, local labels from that data segment can be used in the current segment.
STRONG	$E5	This record contains the name of a segment that must be included during linking, even if no external references have been made to it.
GLOBAL	$E6	This record contains the name of a global label followed by three attribute fields. The label is assigned the current value of the location counter. The first attribute field is two bytes long and gives the number of bytes generated by the line that defined the label. If this field is $FFFF, it indicates that the actual length is unknown but that it is greater than or equal to $FFFF. The second attribute field is one byte long and specifies the type of operation in the line that defined the label. The following type attributes are defined (uppercase ASCII characters with the high bit off):

A	address-type `dc` statement
B	boolean-type `dc` statement
C	character-type `dc` statement
D	double-precision floating-point-type `dc` statement
F	floating-point-type `dc` statement
G	`equ` or `gequ` statement
H	hexadecimal-type `dc` statement
I	integer-type `dc` statement
K	reference-address-type `dc` statement
L	soft-reference-type `dc` statement
M	instruction
N	assembler directive
O	`org` statement
P	`align` statement
S	`ds` statement
X	arithmetic symbolic parameter
Y	boolean symbolic parameter
Z	character symbolic parameter

The third attribute field is one byte long and is the private flag (1 = private). This flag is used to designate a code or data segment as private.

GEQU	$E7	This record contains the name of a global label followed by three attribute fields and an expression. The label is given the value of the expression. The first attribute field is 2 bytes long and gives the number of bytes generated by the line that defined the label. The second attribute field is 1 byte long and specifies the

		type of operation in the line that defined the label, as listed in the discussion of the **GLOBAL** record. The third attribute field is 1 byte long and is the private flag (1 = private). This flag is used to designate a code or data segment as private.
MEM	$E8	This record contains two numbers that represent the starting and ending addresses of a range of memory that must be reserved. If the size of the numbers is not specified, the length of the numbers is defined by the **NUMLEN** field in the segment header.
EXPR	$EB	This record contains a 1-byte count followed by an expression. The expression is evaluated, and its value is truncated to the number of bytes specified in the count. The order of the truncation is from most significant to least significant.
ZEXPR	$EC	This record contains a 1-byte count followed by an expression. **ZEXPR** is identical to **EXPR**, except that any bytes truncated must be al 0's. If the bytes are not 0's, the record is flagged as an error.
BEXPR	$ED	This record contains a 1-byte count followed by an expression. **BEXPR** is identical to **EXPR**, except that any bytes truncated must match the corresponding bytes of the location counter. If the bytes don't match, the record is flagged as an error. This record allows the linker to make sure that an expression evaluates to an address in the current memory bank.
RELEXPR	$EE	This record contains a 1-byte length followed by an offset and an expression. The offset is **NUMLEN** bytes long. **RELEXPR** is used to generate a relative branch value that involves an external location. The length indicates how many bytes to generate for the instruction, the offset indicates where the origin of the branch is relative to the current location counter, and the expression is evaluated to yeild the destination of the branch. For example, a `bne loc` instruction, where `loc` is external, generates this record. For the 6502 and 65816 microprocessors, the offset is 1.
LOCAL	$EF	This record contains the name of a local label followed by three attribute fields. The label is assigned the value of the current location counter. The first attribute field is two bytes long and gives the number of bytes generated by the line that defined the label. The second attribute field is one byte long and specifies the type of operation in the line that defined the label, as listed in the discussion of the **GLOBAL** record. The third attribute field is one byte long and is the private flag (1 = private). This

		flag is used to designate a code or data segment as private.
EQU	$F0	This record contains the name of a local label followed by three attribute fields and an expression. The label is given the value of the expression. The first attribute field is two bytes long and gives the number of bytes generated by the line that defined the label. The second attribute field is one byte long and specifies the type of operation in the line that defined the label, as listed in the discussion of the **GLOBAL** record. The third attribute field is one byte long and is the private flag (1 = private). This flag is used to designate a code or data segment as private.
DS	$F1	This record contains a long integer indicating how many bytes of 0's to insert at the current location counter.
LCONST	$F2	This record contains a 4-byte count followed by absolute code or data. The count indicates the number of bytes of data. The **LCONST** record is similar to **CONST** except that it allows for a much greater number of data bytes. Each relocatable load segment consists of **LCONST** records, **DS** records, and a relocation dictionary. See the discussions on **INTERSEG** records, **RELOC** records, and the relocation dictionary for more information.
LEXPR	$F3	This record contains a 1-byte count followed by an expression. The expression is evaluated, and its value is truncated to the number of bytes specified in the count. The order of the truncation is from most significant to least significant.

Because the **LEXPR** record generates an intersegment reference, only simple expressions are allowed in the expression field, as follows:

```
LABEL ± const
LABEL ± const
(LABEL ± const) | ± const
```

In addition, if the expression evaluates to a single label with a fixed, constant offset, and if the label is in another segment and that segment is a dynamic code segment, then the linker creates an entry for that label in the jump-table segment. (The jump-table segment provides a mechanism to allow dynamic loading of segments as they are needed).

ENTRY	$F4	This record is used in the run-time library entry dictionary; it contains a 2-byte number and an offset followed by a label. The number is the segment number. The label is a code-segment name or entry, and the offset is the relative location within the load segment of the label.

cRELOC	$F5	This record is the compressed version of the **RELOC** record. It is identical to the **RELOC** record, except that the offsets are two bytes long rather than four bytes. The **cRELOC** record can be used only if both offsets are less than $10000 (65,536). The following example compares a **RELOC** record and a **cRELOC** record for the same reference:

RELOC	**cRELOC**
$E2	$02
$02	$02
$00	$00
$00000401	$0401
$00000039	$0039
(11 bytes)	(7 bytes)

cINTERSEG	$F6	This record is the compressed version of the **INTERSEG** record. It is identical to the **INTERSEG** record, except that the offsets are two bytes long rather than four bytes, the segment number is one byte rather than two bytes, and this record does not inclue the 2-byte file number. The **cINTERSEG** record can be used only if both offsets are less than $10000 (65,536), the segment number is less than 256, and the file number associated with the reference is 1 (that is, the initial load file). References to segments in run-time library files must use **INTERSEG** records rather than **cINTERSEG** records.

The following example compares an **INTERSEG** record and a **cINTERSEG** record for the same reference:

INTERSEG	**cINTERSEG**
$E3	$F6
$03	$03
$00	$00
$00000720	$0720
$0001	
$000A	$0A
$00000345	$0345
(15 bytes)	(8 bytes)

SUPER	$F7	This is a supercompressed relocation-dicationary record. Each **SUPER** record is the equivalent of many **cRELOC**, **cINTERSEG**, and **INTERSEG** records. It contains a 4-byte length, a 1-byte record type, and one or more subrecords of variable size, as follows:

Opcode:	`$F7`
Length:	number of bytes in the rest of the record (4 bytes)
Type:	0–37(1 byte)
Subrecords:	(variable size)

When **SUPER** records are used, some of the relocation information is stored in the **LCONST** record at the address to be patched.

The length field indicates the number of bytes in the rest of the **SUPER** record (that is, the number of bytes exclusive of the opcode and the length field).

The type byte indicates the type of **SUPER** record. There are 38 types of **SUPER** records:

Value	SUPER record type
0	SUPER RELOC2
1	SUPER RELOC3
2–37	SUPER INTERSEG1 - SUPER INTERSEG36

SUPER RELOC2: This record can be used instead of **cRELOC** records that have a bit-shift count of zero and that relocate two bytes.

SUPER RELOC3: This record can be used instead of **cRELOC** records that have a bit-shift count of zero and that relocate three bytes.

SUPER INTERSEG1: This record can be used instead of **cINTERSEG** records that have a bit-shift count of zero and that relocate three bytes.

SUPER INTERSEG2 through SUPER INTERSEG12: The number in the name of the record referes to the file number of the file in which the record is used. For example, to relocate an address in file 6, use a SUPER INTERSEG6 record. These records can be used instead of INTERSEG records that meet the following criteria:

- Both offsets are less than $10000.
- The segment number is less than 256.
- The bit-shift count is 0.
- The record relocates 3 bytes.
- The file number is from 2 through 12.

SUPER INTERSEG13 through SUPER INTERSEG24: These records can be used instead of **cINTERSEG** records that have a bit-shift count of zero, that relocate two bytes, and that have a segment number of *n* minus 12, where *n* is a number from 13 to 24. For example, to replace a **cINTERSEG** record in segment 6, use a SUPER INTERSEG18 record.

SUPER INTERSEG25 through SUPER INTERSEG36: These records can be used instead of **cINTERSEG** records that have a bit-shift count of $F0 (-16), that relocate two bytes, and that have a segment number of *n* minus 24, where *n* is a number from 25 to 36. For example, to replace a **cINTERSEG** record in segment 6, use a SUPER INTERSEG30 record.

Each subrecord consists of either a 1-byte offset count followed by a list of 1-byte offsets, or a 1-byte skip count.

Each offset count indicates how many offsets are listed in this subrecord. The offsets are one byte each. Each offset corresponds to the low byte of the first (2-byte) offset in the equivalent **INTERSEG**, **cRELOC**, or **cINTERSEG** record. The high byte of the offset is indicated by the location of this offset count in the **SUPER** record: Each subsequent offset count indicates the next 256 bytes of the load segment. Each skip count indicates the number of 256-byte pages to skip; that is, a skip count indicates that there are no offsets within a certain number of 256-byte pages of the load segment.

For example, if patches must be made at offsets 0020, 0030, 0140, and 0550 in the load segment, the subrecords would include the following fields:

2 20 30	the first 256-byte page of the load segment has two patches: one at offset 20 and one at offset 30
1 40	the second 256-byte page has one patch at offset 40
skip-3	skip the next three 256-byte pages
1 50	the sixth 256-byte page has one patch at offset 50

In the actual **SUPER** record, the patch count byte is the number of offsets minus one, and the skip count byte has the high bit set. A SUPER INTERSEG1 record with the offsets in the preceding example would look like this:

`$F7`	opcode
`$00000009`	number of bytes in the rest of the record
`$02`	INTERSEG1-type **SUPER** record
`$01`	the first 256-byte page has two patches
`$20`	patch the load segment at offset $0020
`$30`	patch the segment at $0030
`$00`	the second page has one patch
`$40`	patch the segment at $0140
`$83`	skip the next three 256-byte pages
`$00`	the sixth page has one patch
`$50`	patch the segment at $0550

A comparison with the **RELOC** record shows that a SUPER RELOC record is missing the offset of the reference. Similarly, the SUPER INTERSEG1 through SUPER INTERSEG12 records are missing the segment number and offset of the subroutine referenced. The offsets (which are two bytes long) are stored in the **LCONST** record at the "to be patched" location. For the SUPER INTERSEG1 through SUPER INTERSEG12 records, the segment number is stored in the third byte of the "to be patched" location.

For example, if the example given in the discussion of the INTERSEG record were instead referenced through a SUPER INTERSEG1 record, the value $0345 (the offset of the subroutine referenced) would be stored at offset $0721 in the load segment (the offset of the instruction's operand). The segment number (`$0A`) would be stored at offset $0723, as follows:

```
4503 0A
```

Expressions

Several types of OMF records contain expressions. Expressions form an entremely flexible reverse-Polish stack language that can be evaluated by the linker to yeild numeric values such as addresses and labels. Each expression consists of a series of operators and operands together with the values on which they act.

An operator takes one or two values from the evaluation stack, performs some mathematical or logical operation on them, and places a new value onto the evaluation stack. The final value on the evaluation stack is used as if it were a single value in the record. Note that this evaluation stack is purely a programming concept and does not relate to any hardware stack in the computer. Each operation is stored in the object module file in postfix form; that is, the value or values come first, followed by the operator. For example, since a binary operation is stored as *`Value1 Value2 Operator`*, the operation *`Num1 minus Num2`* is stored as

`Num1Num2−`

The operators are as follows:

- **Binary math operators:** These operators take two numbers (as two's-complement signed integers) from the top of the evaluation stack, perform the specified operation, and place the single-integer result back on the evaluation stack. The binary math operators include:

\$01 addition	(+)
\$02 subtraction	(−)
\$03 multiplication	(∗)
\$04 division	(/, `DIV`)
\$05 integer remainder	(//, `MOD`)
\$06 bit shift	(<<, >>)

  The subtraction operator subtracts the second number from the first number. The division operator divides the first number by the second number. The integer-remainder operator divides the first number by the second number and returns the unsigned integer remainder to the stack. The bit-shift operator shifts the first number by the number of bit positions specified by the second number. If the second number is positive, the first number is shifted to the left, filling vacated bit positions with 0's (arithmetic shift left). If the second number is negative, the first number is shifted right, filling vacated bit positions with 0's (logical shift right).

- **Unary math operator:** A unary math operator takes a number as a two's-complement signed integer from the top of the evaluation stack, performs the operation on it, and places the integer result back on the evaluation stack. The only unary math operator currently available is

\$06 negation	(−)

- **Comparison operators:** These operators take two numbers as two's-complement signed integers from the top of the evaluation stack, perform the comparison, and place the single-integer result back on the evaluation stack. Each operator compares the second number in the stack (TOS - 1) with the number at the top of the stack (TOS). If the comparison is TRUE, a 1 is placed on the stack; if FALSE, a 0 is placed on the stack. The comparison operators include

\$0C less than or equal to	(<=, ≤)
\$0D greater than or equal to	(>=, ≥)
\$0E not equal	(<>, ≠, `!=`)
\$0F less than	(<)
\$10 greater than	(>)
\$11 equal to	(= or ==)

- **Binary logical operators:** These operators take two numbers as Boolean values from the top of the evaluation stack, perform the operation, and place the single Boolean result back on the stack. Boolean values are defined as being FALSE for the number 0 and TRUE for any other number. Logical operators always return a 1 for TRUE. The binary logical operators include

$08 AND	(**, `AND`)
$09 OR	(++, `OR`, |)
$0A EOR	(−−, `XOR`)

- **Unary logical operator:** A unary logical operator takes a number as a Boolean value from the top of the evaluation stack, performs the operation on it, and places the Boolean result back on the stack. The only unary logical operator currently available is

$0B NOT	(¬, `NOT`)

- **Binary bit operators:** These operators take two numbers as binary values from the top of the evaluation stack, perform the operation, and place the single binary result back on the stack. The operations are performed on a bit-by-bit basis. The binary bit operators include

$12 Bit AND	(logical AND)
$13 Bit OR	(inclusive OR)
$14 Bit EOR	(exclusive OR)

- **Unary bit operator:** This operator takes a number as a binary value from the top of the evaluation stack, performs the operation on it, and places the binary result back on the stack. The unary bit operator is

$15 Bit NOT	(complement)

- **Termination operator:** All expressions end with the termination operator $00.

An **operand** causes some value, such as a constant or a label, to be loaded onto the evaluation stack. The operands are as follows:

- **Location-counter operand ($80):** This operand loads the value of the current location counter onto the top of the stack. Because the location counter is loaded before the bytes from the expression are placed into the code stream, the value loaded is the value of the location counter before the expression is evaluated.
- **Constant operand ($81):** This operand is followed by a number that is loaded on the top of the stack. If the size of the number is not specified, its length is specified by the **numlen** field in the segment header.
- **Label-reference operands ($82-$86):** Each of these operand codes is followed by the name of a label and is acted on as follows:

  $82 Weak reference (see the note below).

  $83 The value assigned to the label is placed on the top of the stack.

  $84 The length attribute of the label is placed on the top of the stack.

  $85 The type attribute of the label is placed on the top of the stack. (Type attributes are listed in the discussion of the **GLOBAL** record in the section "**Segment Body**" earlier).

$86 The count attribute is placed on the top of the stack. The count attribute is 1 if the label is defined and 0 if it is not.

- **Relative offset operand ($87):** This operand is followed by a number that is treated as a displacement from the start of the segment. Its value is added to the value that the location counter had when the segment started, and the result is loaded on the top of the stack.

◇ *Note:* The operand code $82 is referred to as the weak reference. The weak reference is an instruction to the linker that asks for the value of a label, if it exists. It is not an error if the linker cannot find the label. However, the linker does not load a segment from a library if only weak references to it exist. If a label does not exist, a 0 is loaded onto the top of the stack. This operand is generally used for creating jump tables to library routines that may or may not be needed in a particular program.